REFLECTION FROM TOPIC 2

Conversion from NFA to DFA

An NFA can have zero, one or more than one move from a given state on a given input symbol. An NFA can also have NULL moves (moves without input symbol). On the other hand, DFA has one and only one move from a given state on a given input symbol.

Steps for converting NFA to DFA:

Step 1: Convert the given NFA to its equivalent transition table

To convert the NFA to its equivalent transition table, we need to list all the states, input symbols, and the transition rules. The transition rules are represented in the form of a matrix, where the rows represent the current state, the columns represent the input symbol, and the cells represent the next state.

Step 2: Create the DFA's start state

The DFA's start state is the set of all possible starting states in the NFA. This set is called the "epsilon closure" of the NFA's start state. The epsilon closure is the set of all states that can be reached from the start state by following epsilon (?) transitions.

Step 3: Create the DFA's transition table

The DFA's transition table is similar to the NFA's transition table, but instead of individual states, the rows and columns represent sets of states. For each input symbol, the corresponding cell in the transition table contains the epsilon closure of the set of states obtained by following the transition rules in the NFA's transition table.

Step 4: Create the DFA's final states
The DFA's final states are the sets of states that contain at least one final state from the NFA.

Step 5: Simplify the DFA
The DFA obtained in the previous steps may contain unnecessary states and transitions. To simplify the DFA, we can use the following techniques:

- Remove unreachable states: States that cannot be reached from the start state can be removed from the DFA.
- Remove dead states: States that cannot lead to a final state can be removed from the DFA.
- Merge equivalent states: States that have the same transition rules for all input symbols can be merged into a single state.

Step 6: Repeat steps 3-5 until no further simplification is possible
After simplifying the DFA, we repeat steps 3-5 until no further simplification is possible. The final DFA obtained is the minimized DFA equivalent to the given NFA.

Example: consider the NFA below

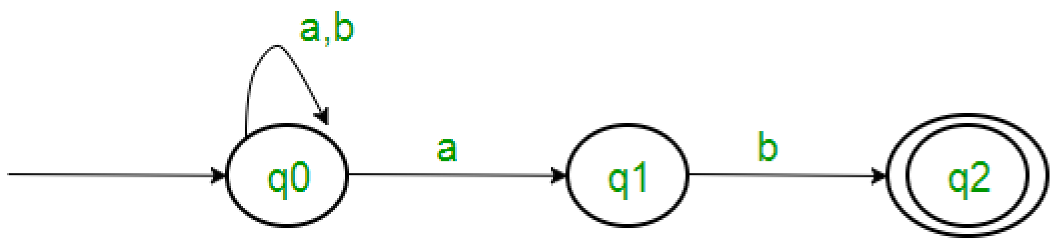
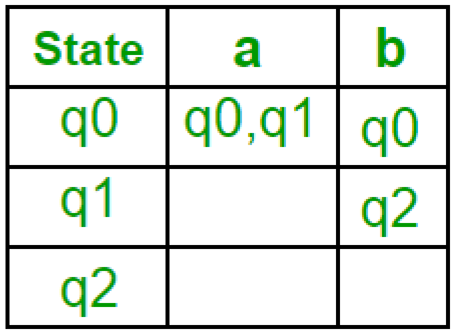


Figure 1

Following are the various parameters for NFA. Q = { q0, q1, q2 } ? = (a, b) F = { q2 } ? (Transition Function of NFA)

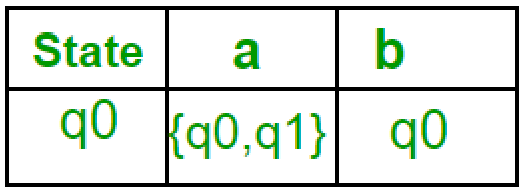


State	a	b
q0	q0,q1	q0
q1		q2
q2		

Step 1: Q' = ?

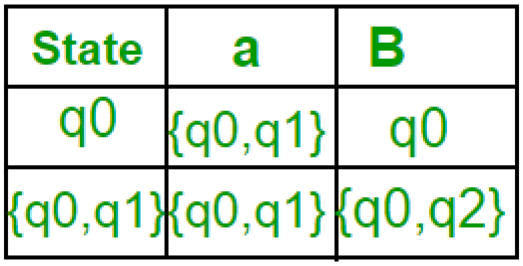
Step 2: Q' = {q0}

Step 3: For each state in Q', find the states for each input symbol. Currently, state in Q' is q0, find moves from q0 on input symbol a and b using transition function of NFA and update the transition table of DFA. ?' (Transition Function of DFA)



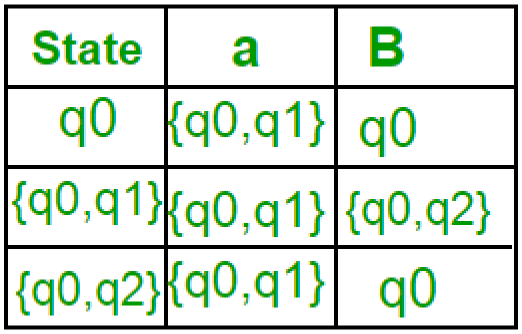
State	a	b
q0	{q0,q1}	q0

Now { q0, q1 } will be considered as a single state. As its entry is not in Q’, add it to Q’. So Q’ = { q0, { q0, q1 } } Now, moves from state { q0, q1 } on different input symbols are not present in transition table of DFA, we will calculate it like: ?’ ({ q0, q1 }, a) = ? (q0, a) ? ? (q1, a) = { q0, q1 } ?’ ({ q0, q1 }, b) = ? (q0, b) ? ? (q1, b) = { q0, q2 } Now we will update the transition table of DFA. ?’ (Transition Function of DFA)



State	a	B
q0	{q0,q1}	q0
{q0,q1}	{q0,q1}	{q0,q2}

Now { q0, q2 } will be considered as a single state. As its entry is not in Q’, add it to Q’. So Q’ = { q0, { q0, q1 }, { q0, q2 } } Now, moves from state {q0, q2} on different input symbols are not present in transition table of DFA, we will calculate it like: ?’ ({ q0, q2 }, a) = ? (q0, a) ? ? (q2, a) = { q0, q1 } ?’ ({ q0, q2 }, b) = ? (q0, b) ? ? (q2, b) = { q0 } Now we will update the transition table of DFA. ?’ (Transition Function of DFA)



State	a	B
q0	{q0,q1}	q0
{q0,q1}	{q0,q1}	{q0,q2}
{q0,q2}	{q0,q1}	q0

As there is no new state generated, we are done with the conversion. Final state of DFA will be state which has q2 as its component i.e., { q0, q2 } Following are the various parameters for DFA. Q’ = { q0, { q0, q1 }, { q0, q2 } } ? = (a, b) F = { { q0, q2 } } and transition function ?’ as shown above. The final DFA for above NFA has been shown in Figure 2.

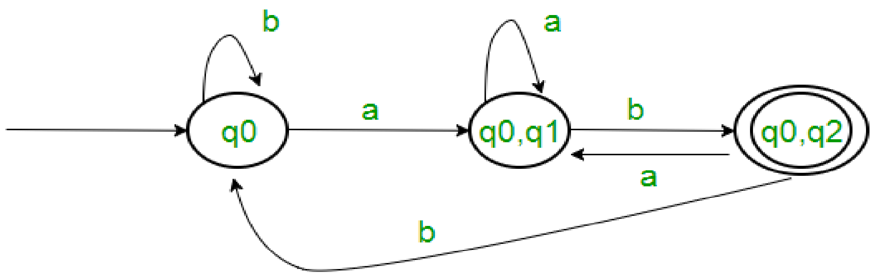


Figure 2

1. Regular Expressions

Regular Expressions (RegEx) are sequences of characters defining search patterns. They are used in search engines, text processing, and syntax validation. In the theory of computation, they describe **regular languages**, which can be recognized by **finite automata**.

2. Regular Languages

Regular languages are a class of languages that can be expressed using regular expressions. They are the simplest class in the Chomsky hierarchy of languages and can be recognized by deterministic finite automata (DFA) and nondeterministic finite automata (NFA).

3. Syntax of Regular Expressions

A regular expression is built using:

- **Literals**: Direct representation of characters.
  - Example: a, b, 1, 0
- **Concatenation**: Joining expressions sequentially.
  - Example: ab (matches "ab")
- **Alternation (Union)**: Choice between expressions.
  - Example: a|b (matches "a" or "b")
- **Kleene Star**: Zero or more repetitions.
  - Example: a* (matches "", "a", "aa", "aaa", etc.)
- **Parentheses**: Grouping of expressions.
  - Example: (ab)* (matches "", "ab", "abab", etc.)

4. Examples of Regular Expressions

- a* matches any number of 'a' characters including the empty string.

- (ab|cd) matches either "ab" or "cd".
- (a|b)* matches any combination of 'a' and 'b' including the empty string.
- a(b|c)*d matches an 'a' followed by any combination of 'b' and 'c', ending with a 'd'.

5. Finite Automata

Finite automata are abstract machines used to recognize patterns defined by regular expressions.

Deterministic Finite Automata (DFA)

A DFA has:

- A finite set of states.
- An alphabet of input symbols.
- A transition function mapping state-symbol pairs to states.
- A start state.
- A set of accepting states.

Example DFA for the regular expression a*:

- States: {q0, q1}
- Alphabet: {a}
- Transition function: δ(q0, a) = q1, δ(q1, a) = q1
- Start state: q0
- Accepting states: {q0, q1}

Nondeterministic Finite Automata (NFA)

An NFA has the same components as a DFA but allows:

- Multiple transitions for the same input symbol.
- Transitions without input symbols (ε-transitions).

Example NFA for the regular expression (a|b)*:

- States: {q0, q1}
- Alphabet: {a, b}
- Transition function:
    - δ(q0, a) = {q0}
    - δ(q0, b) = {q0}
- Start state: q0

- Accepting states: {q0}

6. Equivalence of DFA and NFA

- **NFA to DFA Conversion**: An NFA can be converted to an equivalent DFA using the subset construction method.
- **DFA to Regular Expression**: A DFA can be converted back to a regular expression using state elimination techniques.

7. Regular Expressions and Closure Properties

Regular languages are closed under several operations:

- **Union**: If L1 and L2 are regular, then L1 ∪ L2 is regular.
- **Concatenation**: If L1 and L2 are regular, then L1L2 is regular.
- **Kleene Star**: If L is regular, then L* is regular.
- **Intersection and Complement**: Regular languages are closed under intersection and complement.

8. Applications of Regular Expressions

- **Text Search**: Finding patterns within text i.e in searching algorithms, search engines, and text processing tools..
- **Lexical Analysis**: Tokenizing source code in compilers.
- **Pattern Matching/Input Validation**: Checking the structure of inputs (e.g., email validation).

9. Limitations of Regular Expressions

Regular expressions cannot describe all languages. They are limited to regular languages and cannot express languages that require memory of arbitrarily large input (e.g., balanced parentheses).

Non-Regular Languages

In the theory of computation, non-regular languages are languages that cannot be recognized by finite automata and cannot be described by regular expressions. These languages require more powerful computational models to be recognized, such as **pushdown automata or Turing machines**.

Properties of Non-Regular Languages

1. **Complexity**: Non-regular languages often involve patterns that cannot be captured by the simple memoryless nature of finite automata.
2. **Context-Sensitivity**: Non-regular languages can require context-sensitive recognition, meaning that the recognition of a particular string may depend on the entire context of the input rather than just a small, fixed portion of it.
3. **Stack or Tape Memory**: Recognition of non-regular languages often requires additional memory in the form of a stack (for context-free languages) or an unbounded tape (for recursively enumerable languages).

Examples of Non-Regular Languages

1. **Balanced Parentheses**: The language of balanced parentheses is a classic example of a non-regular language. For example, the language L={anbn∣n≥0}L = \{a^n b^n \mid n \geq 0\}L={anbn∣n≥0} is non-regular because it requires matching the number of as with the number of bs, which a finite automaton cannot do.
2. **Palindrome Language**: The language of palindromes over an alphabet Σ\SigmaΣ is non-regular. For example, L={w∣w=wR}L = \{w \mid w = w^R\}L={w∣w=wR}, where wRw^RwR is the reverse of www, is non-regular because checking for a palindrome requires storing and comparing the string, which is beyond the capability of finite automata.
3. **Copy Language**: The language L={ww∣w∈Σ∗}L = \{ww \mid w \in \Sigma^*\}L={ww∣w∈Σ∗} is non-regular because it requires ensuring that two consecutive substrings are identical, which requires more memory than finite automata can provide.

Pumping Lemma for Regular Languages

The Pumping Lemma provides a method to prove that certain languages are not regular by demonstrating that all strings in the language cannot satisfy the properties of regular languages.

Pumping Lemma Statement: If LLL is a regular language, then there exists a number ppp (the pumping length) such that any string sss in LLL with |s|≥p|s| \geq p|s|≥p can be split into three parts s=xyzs = xyzs=xyz, satisfying the following conditions:

1. |xy|≤p|xy| \leq p|xy|≤p
2. |y|>0|y| > 0|y|>0
3. xyiz∈Lxy^i z \in Lxyiz∈L for all i≥0i \geq 0i≥0

Using the Pumping Lemma: To prove that a language LLL is not regular:

1. Assume, for contradiction, that LLL is regular.
2. Let ppp be the pumping length given by the Pumping Lemma.
3. Choose a string sss in LLL with |s|≥p|s| \geq p|s|≥p.
4. Show that for all possible decompositions of s=xyzs = xyzs=xyz that meet the conditions of the lemma, at least one iii exists such that xyiz∉Lxy^i z \notin Lxyiz∈/L.
5. This contradiction implies that LLL is not regular.

Example: Proving L={anbn∣n≥0}L = \{a^n b^n \mid n \geq 0\}L={anbn∣n≥0} is not regular using the Pumping Lemma:

1. Assume LLL is regular.
2. Let ppp be the pumping length.
3. Choose s=apbps = a^p b^ps=apbp, which is clearly in LLL and has |s|=2p≥p|s| = 2p \geq p|s|=2p≥p.
4. Split sss into xyzxyzxyz with |xy|≤p|xy| \leq p|xy|≤p and |y|>0|y| > 0|y|>0.
5. Since |xy|≤p|xy| \leq p|xy|≤p, yyy consists only of as.
6. Pumping yyy, i.e., considering xy2zxy^2 zxy2z, results in more as than bs, specifically ap+|y|bpa^{p+|y|} b^pap+|y|bp, which is not in LLL.
7. This contradiction shows that LLL is not regular.