National University

Manila

College of Computing and Information Technologies

Computer Science Department

Automata Theory and Formal Languages (CCAUTOMA)

TERM 1 | AY2025-2026

Solo, Alvhin Carcueva

COM232

Ms. Susan Caluya

BUILD A DFA THAT ACCEPTS STRINGS WITH NO 'ABA' SUBSTRING

Instruction Analysis

The language L consists of all strings over the alphabet $\Sigma = \{a, b\}$ that do not contain 'aba' as a substring. This includes the empty string, single-character strings such as “a” and “b,” and any combination of symbols that never forms the consecutive sequence “aba.” In simpler terms, any string that avoids the pattern “aba” anywhere within it is accepted by the automaton. Examples of accepted strings include “ab,” “ba,” “aabb,” and “babb,” since they do not contain the forbidden sequence. On the other hand, strings like “aba,” “baba,” and “aabaa” are rejected because they contain the pattern “aba” at least once, violating the rule that defines the language.

The required automaton type is a Deterministic Finite Automaton (DFA). A DFA operates by reading an input string symbol by symbol, making transitions between a finite number of states based on the current symbol and its present state. It differs from non-deterministic automata in that each state and input combination leads to exactly one next state, ensuring a clear and predictable computation. This deterministic property makes DFAs efficient for tasks involving pattern recognition and validation, such as identifying or excluding substrings. Furthermore, the DFA’s ability to model regular languages through finite states allows it to handle problems that depend only on limited contextual information, such as tracking a short sequence of symbols like “aba.”

The key condition is that the DFA must accept any string without the pattern 'aba' and reject any string containing it, regardless of position. The automaton must track recent symbols to detect when 'a', 'b', 'a' occurs consecutively. Since DFAs have no memory beyond their current state, this tracking is accomplished by encoding progress through states. The pattern tracking strategy requires four states: one for no progress toward 'aba', one for seeing 'a', one for seeing 'ab', and a trap state for detecting complete 'aba'. Once the automaton transitions into the trap state, it remains there permanently, ensuring that any string containing “aba” is rejected. This systematic design allows the DFA to operate deterministically while effectively preventing the occurrence of the forbidden substring.

Research Summary

The study of Deterministic Finite Automata (DFA) remains one of the foundational concepts in automata theory and computation. DFAs are mathematical models used to represent and analyze systems with a finite number of states and deterministic behavior. Formally, a DFA is represented as a five-tuple $M = (Q, \Sigma, \delta, q_0, F)$, where:

- Q is the finite set of states
- Σ is the input alphabet
- δ is the transition function
- q_0 is the start state
- F is the set of accepting states.

This structure enables DFAs to process input strings symbol by symbol, making deterministic decisions based on well-defined transitions.

The theory behind DFAs emphasizes their role in recognizing regular languages, which can be described by regular expressions. Finite automata provide a foundation for understanding how machines can model computation with limited memory, particularly in pattern recognition and lexical analysis. Modern applications extend to compiler design, software verification, and digital communication protocols, where deterministic behavior ensures reliability and predictability.

In the context of constructing a DFA that rejects strings containing the forbidden substring “aba”, each state serves to represent the machine’s progress toward detecting this specific sequence. As highlighted by Sipser (2021), DFAs rely on state-based memory, where previous inputs influence the next transition without the need for external storage. Thus, to prevent “aba” from appearing in any part of the string, states must be carefully designed to track partial matches while ensuring immediate rejection once the pattern is completed.

To explore similar problems, several solved examples in automata theory demonstrate how DFAs can be constructed to avoid or detect forbidden substrings such as “aa,” “00,” or “abba.” These examples reveal a common design principle known as progressive pattern tracking, where each state represents a specific level of progress toward recognizing an undesired sequence. This method allows the automaton to simulate memory through its states, even without external storage. For instance, in designing a DFA that rejects strings containing “aa,” the machine transitions to a new state upon reading the first ‘a,’ indicating partial progress toward the forbidden pattern. If another ‘a’ immediately follows, it moves into a trap state, confirming detection and leading to rejection. Similarly, in the case of the substring “aba,” after reading “ab,” the DFA enters a “watch” state that anticipates whether the next symbol will complete the sequence. If the next input is ‘a,’ the automaton transitions to a trap state to ensure permanent rejection; however, if a ‘b’ follows, it resets safely to the earlier state, indicating the pattern has been broken. This strategy ensures that the DFA functions deterministically, efficiently, and without ambiguity while continuously monitoring input symbols for the formation of the restricted pattern.

As outlined in Linz (2022), this process ensures completeness (every input has a transition) and determinism (no ambiguities exist between states). Simulation strategies, both manual and digital, such as JavaScript- or Python-based DFA simulators, allow visualization of each state transition and verification of correctness through test strings.

The design of a DFA that rejects strings containing the substring “aba” synthesizes multiple theoretical and practical aspects of automata theory, from understanding regular language recognition to mastering systematic state-based design and validation through simulation.

DFA Design

The Deterministic Finite Automaton (DFA) for this problem is formally represented as:

$$M = (Q, \Sigma, \delta, q_0, F)$$

where each component defines the machine’s structure and behavior:

- $Q = \{q_0, q_1, q_2, q_3\}$
- $\Sigma = \{a, b\}$
- q_0 is the initial state
- $F = \{q_0, q_1, q_2\}$
- δ is the transition function that determines the next state based on the current state and the input symbol.

This DFA is designed to accept all strings that do not contain the substring “aba” and reject any string that includes it at any position. To achieve this, the automaton keeps track of the most recent symbols read, ensuring that whenever the sequence “a–b–a” is detected, the machine transitions to a trap (rejecting) state.

Each state in the DFA has a specific meaning and function:

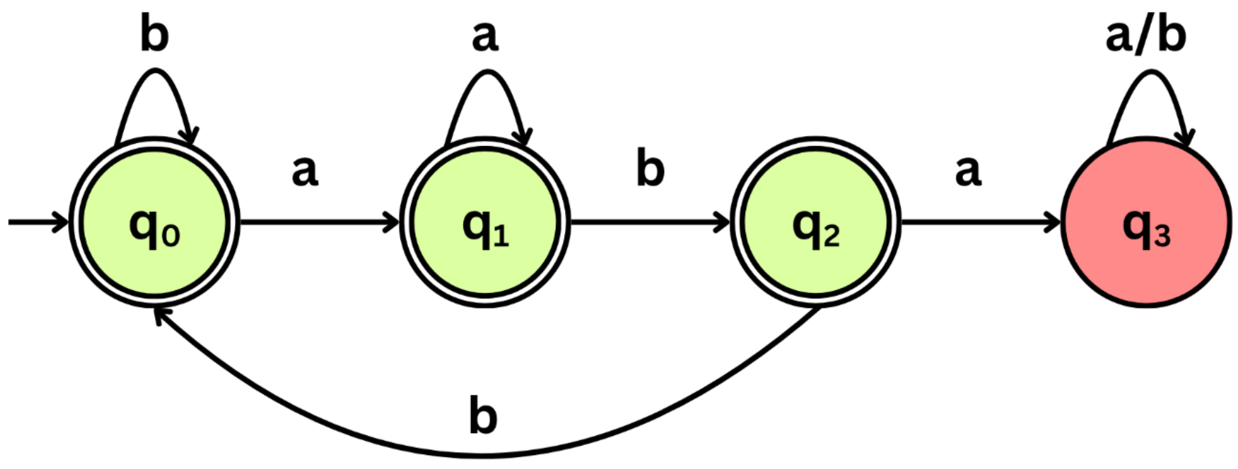
- q_0 represents no progress toward 'aba'
- q_1 represents having seen 'a'
- q_2 represents having seen 'ab'
- q_3 is the trap state where 'aba' has been detected.

The accepting states are $\{q_0, q_1, q_2\}$, as these represent all conditions where the DFA has not yet identified the forbidden pattern. Only q_3 is excluded, signifying the detection of “aba.”

The transition function δ is defined in the following table:

Current State	Input 'a'	Input 'b'
q_0	q_1	q_0
q_1	q_1	q_2
q_2	q_3	q_0
q_3	q_3	q_3

State Transition Diagram



- q_0 is the initial safe state reached when starting or after reading 'b' from q_2. On input 'a', it transitions to q_1 as this could start 'aba'. On input 'b', it stays in q_0 since there's no progress toward the pattern.
- q_1 represents having seen 'a' which could begin 'aba'. On another 'a', it stays in q_1 since the new 'a' might start a fresh pattern. On 'b', it moves to q_2 forming 'ab'.

- q_2 represents the dangerous 'ab' sequence, two-thirds complete. On 'a', it transitions to q_3 completing 'aba' and rejecting the string. On 'b', it returns to q_0 since 'abb' breaks the pattern.
- q_3 is the trap state indicating 'aba' was found. All transitions from q_3 lead back to itself, ensuring permanent rejection.

Simulation

Accepted Strings

Input String	Path Trace	Final State	Result	Explanation
ε (empty)	q_0	q_0	✓ ACCEPT	No 'aba' in empty string
a	$q_0 \to q_1$	q_1	✓ ACCEPT	Just 'a', incomplete pattern
b	$q_0 \to q_0$	q_0	✓ ACCEPT	No 'a' to start pattern
ab	$q_0 \to q_1 \to q_2$	q_2	✓ ACCEPT	'ab' is incomplete, missing second 'a'
ba	$q_0 \to q_0 \to q_1$	q_1	✓ ACCEPT	Pattern not formed
bb	$q_0 \to q_0 \to q_0$	q_0	✓ ACCEPT	No 'aba' present
aa	$q_0 \to q_1 \to q_1$	q_1	✓ ACCEPT	Only 'a's, no 'b' between
abb	$q_0 \to q_1 \to q_2 \to q_0$	q_0	✓ ACCEPT	Pattern broken by second 'b'
bab	$q_0 \to q_0 \to q_1 \to q_2$	q_2	✓ ACCEPT	Ends with 'ab', not complete 'aba'
aabb	$q_0 \to q_1 \to q_1 \to q_2 \to q_0$	q_0	✓ ACCEPT	No 'aba' substring
abbb	$q_0 \to q_1 \to q_2 \to q_0 \to q_0$	q_0	✓ ACCEPT	Multiple 'b's prevent pattern
bbab	$q_0 \to q_0 \to q_0 \to q_1 \to q_2$	q_2	✓ ACCEPT	Ends before completing 'aba'

Rejected Strings

Input String	Path Trace	Final State	Result	Explanation
aba	$q_0 \to q_1 \to q_2 \to q_3$	q_3	✗ REJECT	Exact match of forbidden pattern
abab	$q_0 \to q_1 \to q_2 \to q_3 \to q_3$	q_3	✗ REJECT	Contains 'aba' at start
baba	$q_0 \to q_0 \to q_1 \to q_2 \to q_3$	q_3	✗ REJECT	Contains 'aba' after 'b'

aaba	$q_0 \rightarrow q_1 \rightarrow q_1 \rightarrow q_2 \rightarrow q_3$	q_3	✗ REJECT	Contains 'aba' at end
abaa	$q_0 \rightarrow q_1 \rightarrow q_2 \rightarrow q_3 \rightarrow q_3$	q_3	✗ REJECT	Contains 'aba' at start
babab	$q_0 \rightarrow q_0 \rightarrow q_1 \rightarrow q_2 \rightarrow q_3 \rightarrow q_3$	q_3	✗ REJECT	Contains 'aba' in middle
aabaa	$q_0 \rightarrow q_1 \rightarrow q_1 \rightarrow q_2 \rightarrow q_3 \rightarrow q_3$	q_3	✗ REJECT	Contains 'aba' pattern
abbaba	$q_0 \rightarrow q_1 \rightarrow q_2 \rightarrow q_0 \rightarrow q_1 \rightarrow q_2 \rightarrow q_3$	q_3	✗ REJECT	Contains 'aba' at end
ababab	$q_0 \rightarrow q_1 \rightarrow q_2 \rightarrow q_3 \rightarrow q_3 \rightarrow q_3 \rightarrow q_3$	q_3	✗ REJECT	Multiple 'aba' patterns

Code Snippet

- Transition Table (Core Logic)

```
const transitions = {
    q0: { a: 'q1', b: 'q0' },
    q1: { a: 'q1', b: 'q2' },
    q2: { a: 'q3', b: 'q0' },
    q3: { a: 'q3', b: 'q3' }
};
```

Explanation: Defines the DFA's state transitions. For example, from q0, reading 'a' moves to q1, reading 'b' stays in q0. State q3 is the trap state (once reached, stays forever).

- State Transition Logic

```
let state = 'q0';
const char = input[i];
const prevState = state;
state = transitions[state][char];
```

Explanation: Reads each character and updates the current state using the transition table. This is the heart of DFA simulation.

- Input Validation

```
if (!/^[ab]*$/.test(input)) {
    result.textContent = '⚠️ Invalid Input! Please use only letters "a" and "b"';
    return;
}
```

Explanation: Uses regex to ensure input contains only 'a' and 'b' characters. Rejects invalid strings immediately.

- Accept/Reject Decision

```
const isRejected = state === 'q3';
result.textContent = isRejected ?
```

```
    `❌ STRING REJECTED! Found "aba" substring` :
    `✅ STRING ACCEPTED! No "aba" found`;
```

Explanation: After processing all input, checks if final state is q3 (trap). If yes, string contains "aba" and is rejected; otherwise accepted.

- Dynamic Path Highlighting

```
const transitionPaths = {
    'q0-a': 'trans-q0-q1',
    'q1-b': 'trans-q1-q2',
};

function highlightTransition(from, input) {
    const pathId = transitionPaths[`${from}-${input}`];
    const path = document.getElementById(pathId);
    path.classList.add('active-transition');
    path.setAttribute('marker-end', 'url(#arrow-active)');
}
```

Explanation: Maps state transitions to SVG path elements. When a transition occurs, finds the corresponding path and applies animation CSS class plus pink arrow marker.

- Step-by-Step Log

```
function addStep(stepNum, char, fromState, toState, description) {
    stepDiv.innerHTML = `
        <span class="step-number">${stepNum}</span>
        Reading '${char}': ${fromState} → ${toState}
        <br><small>${description}</small>
    `;
    stepsContainer.appendChild(stepDiv);
}
```

Explanation: Dynamically creates step cards showing each transition. Includes step number, character read, state change, and descriptive text.

Reflection

Designing a Deterministic Finite Automaton (DFA) that rejects strings containing the substring “aba” has been an enlightening and intellectually engaging experience. This activity helped me understand how abstract computational models translate into structured systems capable of logical decision-making. Through this process, I learned that even a simple automaton can represent complex behaviors through well-defined states and transitions.

Constructing this DFA required a clear understanding of how each state functions as a memory mechanism. Since DFAs operate with limited memory, every transition must represent meaningful progress or reset within the pattern. This became particularly evident in handling overlapping sequences such as “aab” or “abab,” where the machine must correctly interpret whether to continue tracking the forbidden substring or safely return to the starting state. Achieving this balance between progress and reset deepened my appreciation for the precision and logical structure that automata demand.

The most challenging aspect was ensuring that the automaton responded correctly to every possible input combination. It required analyzing how partial patterns could appear within longer strings and verifying that no valid input would mistakenly lead to rejection. This step-by-step reasoning and testing process improved my analytical thinking and attention to detail, qualities that are essential in both theoretical and applied computer science.

Beyond the technical aspect, this project allowed me to appreciate how automata theory serves as the foundation for various applications such as text recognition, data validation, and software verification. It demonstrated how formal logic and mathematical precision can be used to ensure correctness and predictability in computation.

Overall, this activity strengthened my understanding of computational logic and the disciplined approach required to design deterministic systems. It reinforced the idea that theoretical computer science is not just about symbols and equations, it is about building frameworks that enable consistent reasoning, structured problem-solving, and reliable system design.

References

Linz, P. (2022). *An Introduction to Formal Languages and Automata* (7th ed.). Jones & Bartlett Learning.

Sipser, M. (2021). *Introduction to the Theory of Computation* (4th ed.). Cengage Learning.

Automata Theory and Formal Language Course Materials. National University-Manila.