DFA Implementation

Sifeddine Akaaboune
Student No. : 2021326660027
Course : Compiler Principle

January 2024

1 Abstract

This paper presents the design and implementation of a Deterministic Finite Automaton (DFA) that can recognize the regular expression a*(bb—baa)*aa. The DFA is implemented in the C++ programming language and is capable of accepting strings and providing an error message for non-acceptable strings. The DFA will be presented using a transition table and a transition diagram. The report also includes instructions on running the implementation and provides sample executions of acceptable and non-acceptable strings, showcasing the state transition sequences on the DFA transition diagram and screenshots of the actual executions.

2 Introduction

Pattern matching is a fundamental operation in various applications, including compilers and text processing tools. To efficiently perform pattern matching, theoretical computer science provides elegant formulations such as Deterministic Finite Automata (DFA) and regular expressions. DFA and regular expressions offer structured ways to specify patterns and have equivalent expressive power. In this paper, we focus on the design and implementation of a DFA that recognizes the regular expression a*(bb—baa)*aa. In the case of this particular regular expression, there is no need to perform the NFA conversion process. This is because the regular expression itself is relatively simple and can be directly implemented as a DFA.

3 Constructing the DFA

The regular expression a*(bb|baa)*aa can be deconstructed into three pieces: a*, bb|baa, aa. With these smaller pieces we can construct simple DFAs for each one:

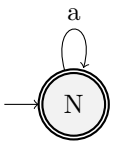


Figure 1: DFA for a*

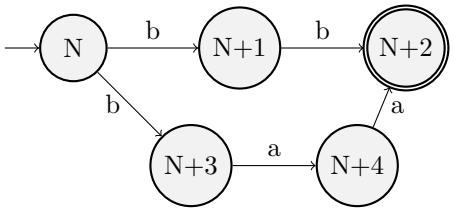


Figure 2: DFA for bb|baa

This DFA can be simplified as the starting will always choose 'b' as its first character; we can remove node N+3 and connect node N+1 to N+4.

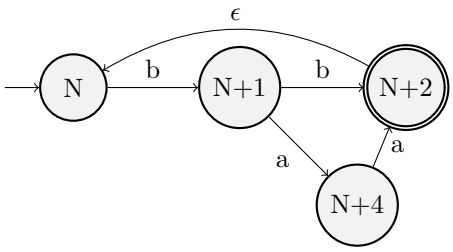


Figure 3: Minimised DFA for (bb|baa)*

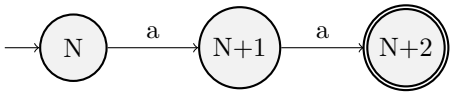


Figure 4: DFA for aa

The next step is to combine all three DFAs. The last DFA(for aa) should be prioritise as it is the constant in all acceptable strings. It will either appear at the end of a string or at the start; when the first two parts of the regular expression are omitted. Therefore the final DFA should account for both cases; this was done by fulfilling the condition at the start and at the end of the DFA, As shown below.

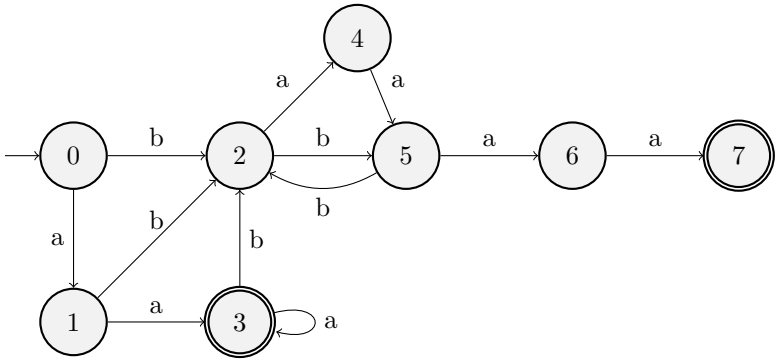


Figure 5: The complete DFA

4 Transition Table

The transition table is an essential tool in the study of automatons. It provides a systematic representation of the transitions between states in an automaton based on the input symbols. The transition table consists of rows representing different states, columns representing input symbols, and entries representing the next state. The initial state is denoted by the plus operator, and the final state is indicated by the subtraction operation. The transition table helps us understand and analyze the behavior of the automaton by providing a clear and organized overview of the state transitions for each input symbol. It allows us to easily determine the next state given the current state and input symbol, making it a valuable tool for designing, implementing, and analyzing DFAs. Below is the transition table of our DFA:

DFA State	Type	a	b
0^+		1	2
1		3	2
2		4	5
3^-	accept	3	2
4		5	
5		6	2
6		7	
7^-	accept		

5 Implementation in C++

The DFA implementation is structured as such:

```
class DFA {
private:
    int currentState;
```

```
public:
    DFA() {
        currentState = 0;
    }

    void Transition(char input) {
        switch (currentState) {
            // cases....
        }
    }

    bool isAcceptState() {
        return currentState == 3 || currentState == 7;
    }

}
```

The DFA class consists of one private variable: **currentState** which as its name implies it indicates the state the DFA is currently at, starting at state 0. The isAcceptState() will return a boolean to see whether the current state is in state 3 or 7; as shown in Figure 5 the accepting states include 3 and 7. The Transition() function is the main component of this DFA class, it checks the current state and current character and when paired with a loop function we can check for each character in a given string. Below is the complete Transition() function:

```
void Transition(char input) {
    switch (currentState) {
        case 0:
            cout << "State:-" << currentState << endl;
            if (input == 'a') {
                currentState = 1;
            } else if (input == 'b') {
                currentState = 2;
            } else {
                currentState = -1;
            }
            break;
        case 1:
            cout << "State:-" << currentState << endl;
            if (input == 'a') {
                currentState = 3;
            } else if (input == 'b') {
                currentState = 2;
            } else {
                currentState = -1;
            }
            break;
        case 2:
            cout << "State:-" << currentState << endl;
            if (input == 'a') {
                currentState = 4;
            } else if (input == 'b') {
                currentState = 5;
            } else {
                currentState = -1;
```

```
            }
            break;
        case 3:
            cout << "State:-" << currentState << endl;
            if (input == 'a') {
                currentState = 3;
            } else if (input == 'b') {
                currentState = 2;
            } else {
                currentState = -1;
            }
            break;
        case 4:
            cout << "State:-" << currentState << endl;
            if (input == 'a') {
                currentState = 5;
            } else {
                currentState = -1;
            }
            break;
        case 5:
            cout << "State:-" << currentState << endl;
            if (input == 'a') {
                currentState = 6;
            } else if (input == 'b') {
                currentState = 2;
            } else {
                currentState = -1;
            }
            break;
        case 6:
            cout << "State:-" << currentState << endl;
            if (input == 'a') {
                currentState = 7;
                cout << "State:-" << currentState << endl;
            } else {
                currentState = -1;
            }
            break;
        case -1:
            break;
    }
}
```

The function does a simple check and returns the respective state based on the input character, it also prints out the current state for a clearer understanding of the path it took.

```
int main() {
    // RegEx a*(bb|baa)*aa
    while(true) {
        DFA dfa;
        string input;
        cout << "Enter a string:-";
        cin >> input;
```

```
        if (input == "exit") return false;

        for (char c : input) {
            dfa.Transition(c);
        }

        if (dfa.isAcceptState())
            cout << "Accept" << endl;
        else
            cout << "Error: Non-acceptable string" << endl;
    }

    return 0;
}
```

The code above shows the implementation of a DFA object. Using a for loop we can iterate through each character from the string that was inputted by the user. After the loop is completed the program will check whether the the final state was a viable exit point which in this case will be either state 3 or 7, if not an error code is printed out.

6 Sample execution

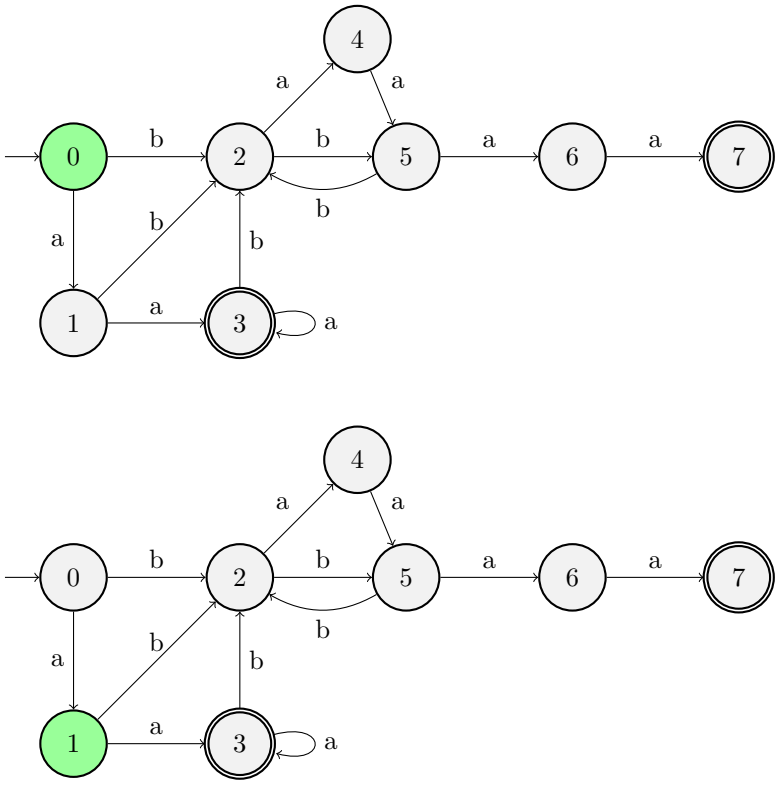
This section will go over the process during an execution of a string for both accepting strings and rejected one. The execution of this program is quite user-friendly, all the user need to do is to input a string such as abbaaaa with not spaces in between the characters and the program will return the "Accept" message or an error if the string was not a viable one for the regular expression a*(bb—baa)*aa.

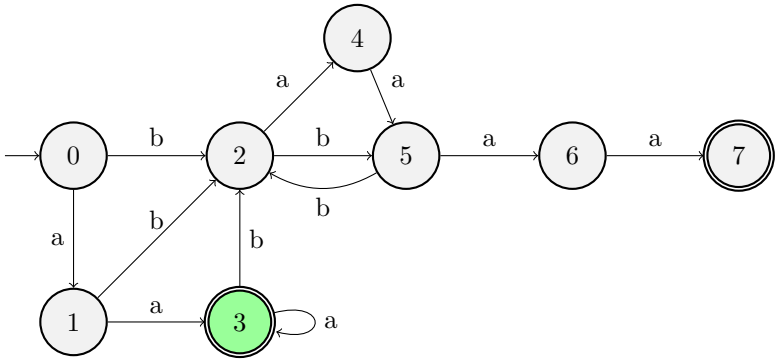
6.1 Accepting strings

Example 1: aaaaaa

```
Enter a string: aaaaaa
State: 0
State: 1
State: 3
State: 3
State: 3
State: 3
State: 3
Accept
```

Figure 6: Execution of string aaaaaa



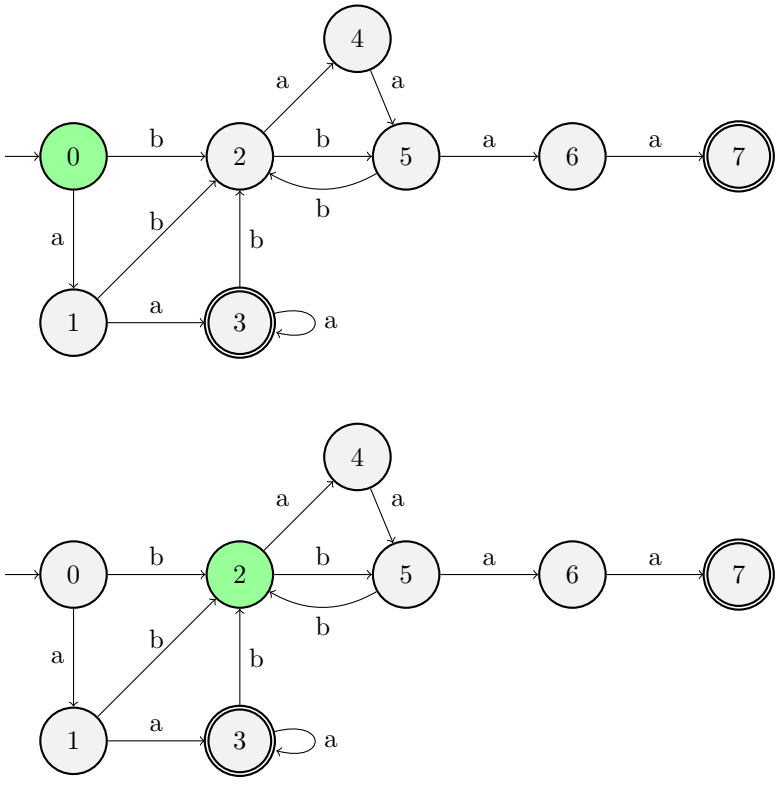


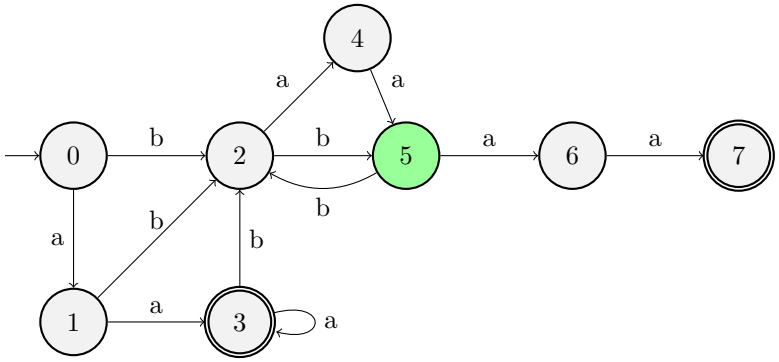
This part is repeated 3 more times before exiting.

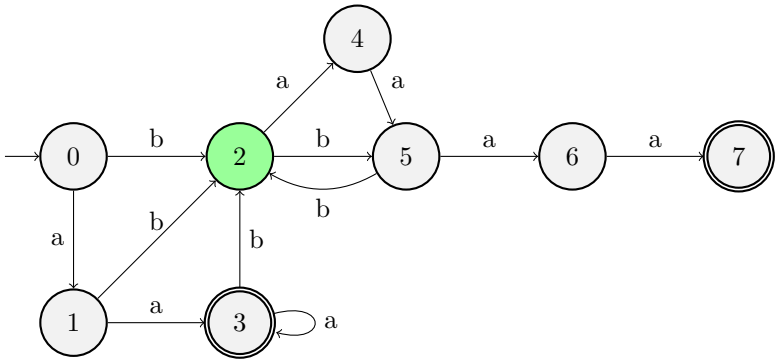
Example 2: bbbaabbaa

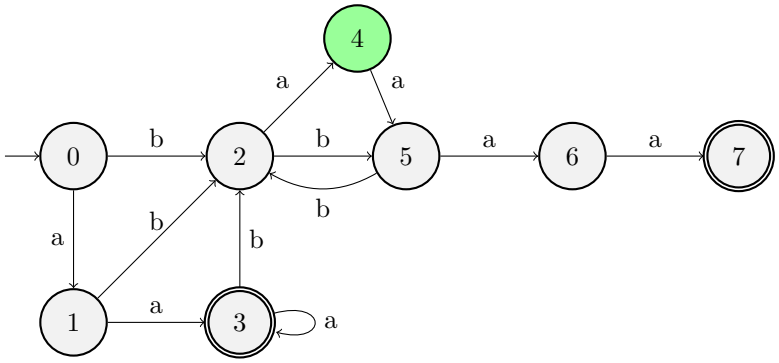
```
Enter a string: bbbaabbaa
State: 0
State: 2
State: 5
State: 2
State: 4
State: 5
State: 2
State: 5
State: 6
State: 7
Accept
```

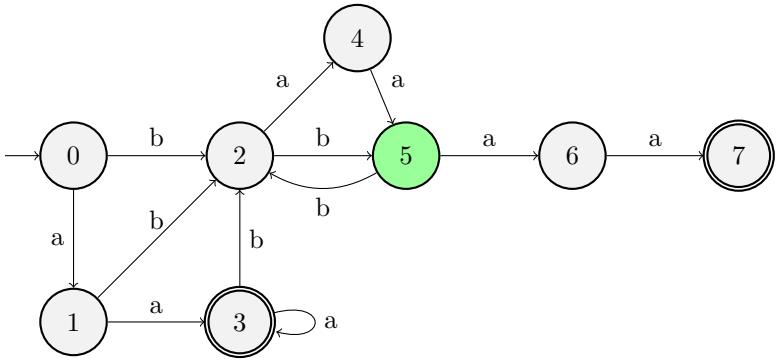
Figure 7: Execution of string bbbaabbaa

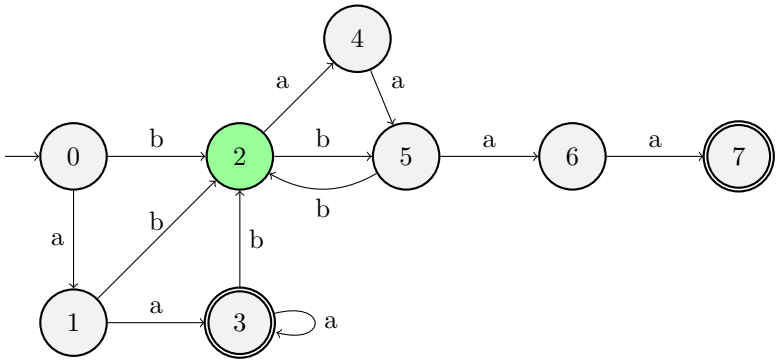


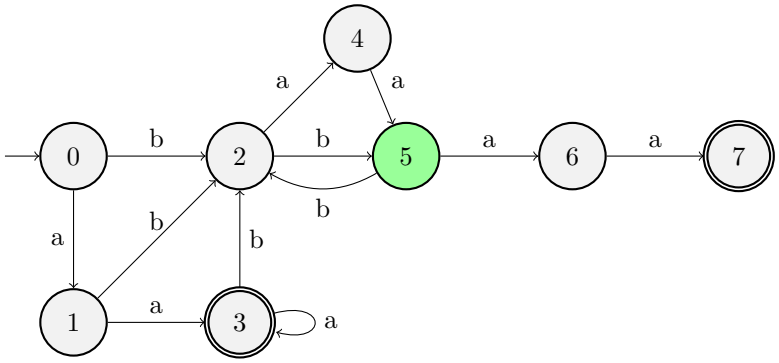

4
a
a
0
b
2
b
5
a
6
a
7
b
a
b
b
1
a
3
a

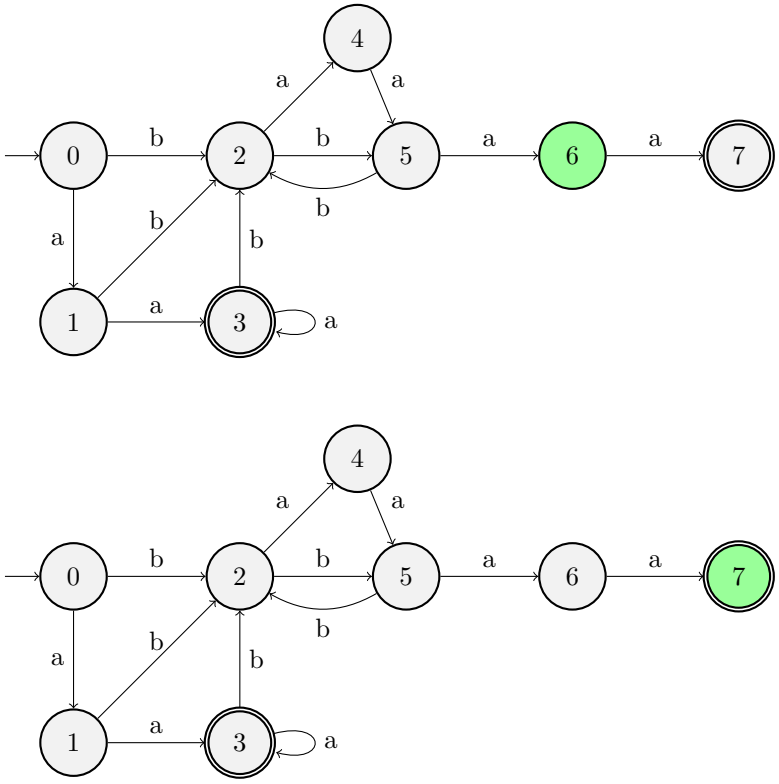

4
a
a
0
b
2
b
5
a
6
a
7
b
a
b
b
1
a
3
a


4
a
a
0
b
2
b
5
a
6
a
7
b
a
b
b
1
a
3
a


4
a
a
0
b
2
b
5
a
6
a
7
b
a
b
b
1
a
3
a


4
a
a
0
b
2
b
5
a
6
a
7
b
a
b
b
1
a
3
a


4
a
a
0
b
2
b
5
a
6
a
7
b
a
b
b
1
a
3
a

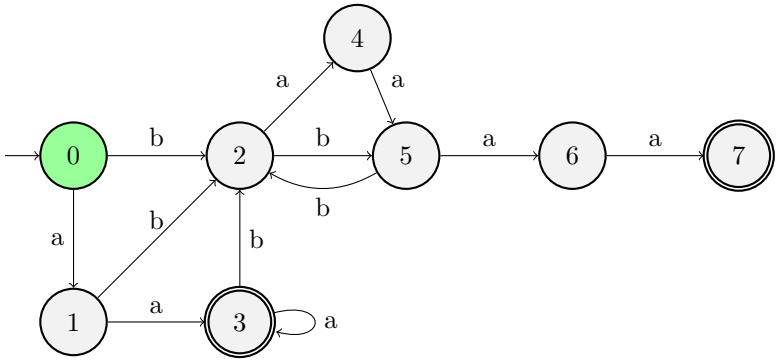


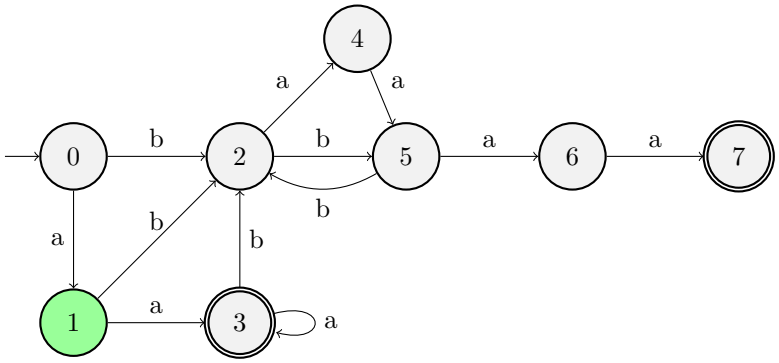
6.2 Rejected strings

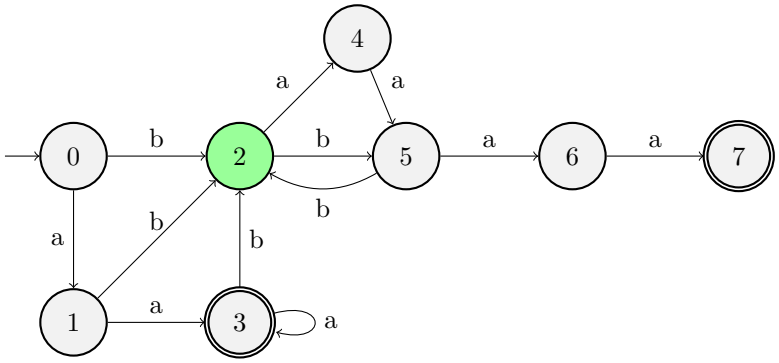
Example 1: abaab

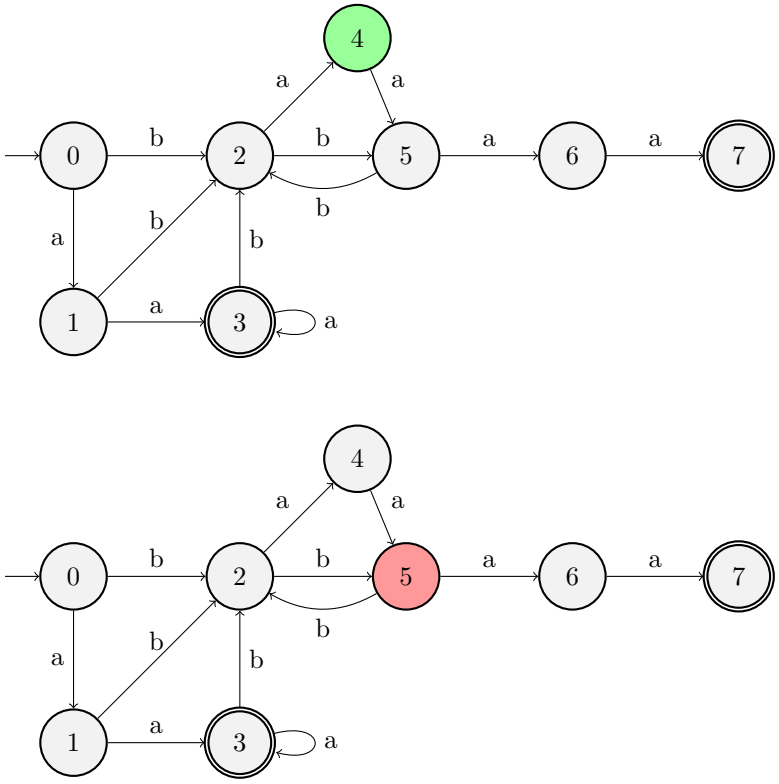
```
Enter a string: abaab
State: 0
State: 1
State: 2
State: 4
State: 5
Error: Non-acceptable string
```

Figure 8: Execution of string abaab


4
a
a
0
b
2
b
5
a
6
a
7
b
a
b
b
1
a
3
a


4
a
a
0
b
2
b
5
a
6
a
7
b
a
b
b
1
a
3
a

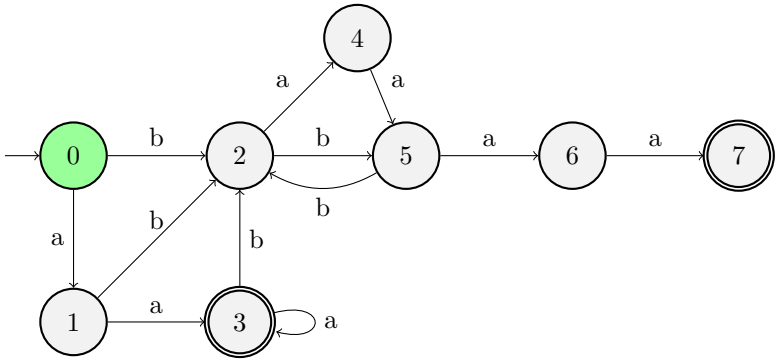

4
a
a
0
b
2
b
5
a
6
a
7
b
a
b
b
1
a
3
a

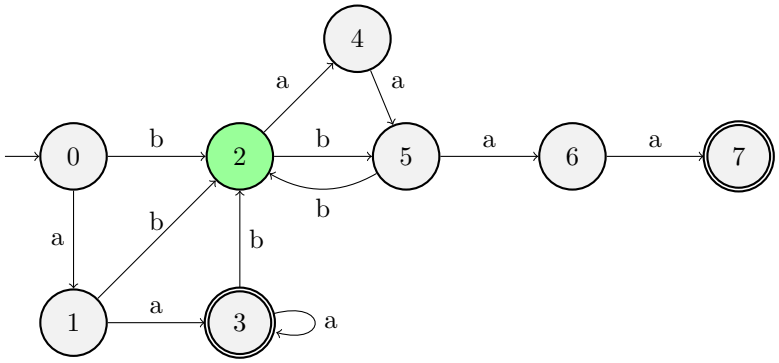


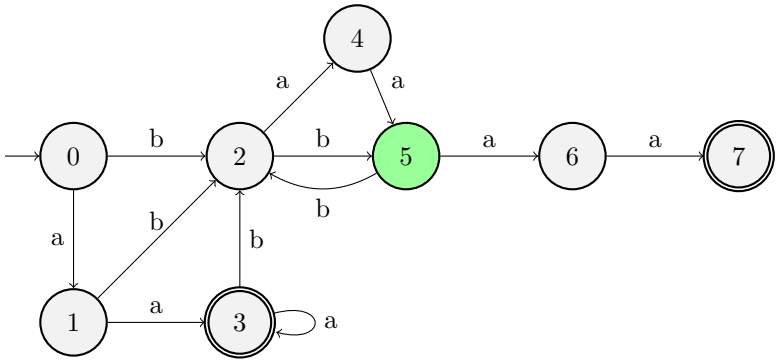
Example 2: bbabb

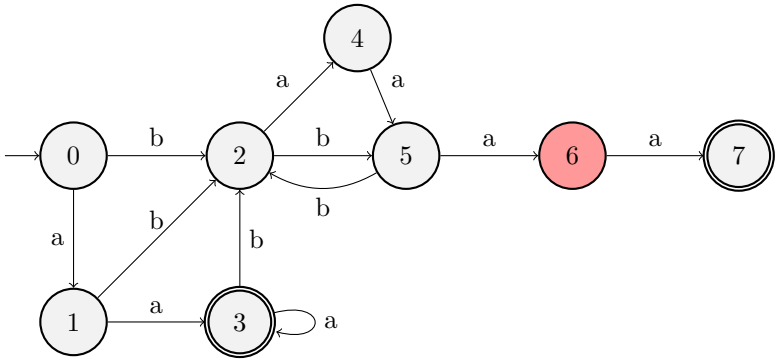
```
Enter a string: bbabb
State: 0
State: 2
State: 5
State: 6
Error: Non-acceptable string
```

Figure 9: Execution of string bbabb


4
a
a
0
b
2
b
5
a
6
a
7
b
a
b
b
1
a
3
a


4
a
a
0
b
2
b
5
a
6
a
7
b
a
b
b
1
a
3
a


4
a
a
0
b
2
b
5
a
6
a
7
b
a
b
b
1
a
3
a



7 Conclusion

This paper has successfully designed and implemented a Deterministic Finite Automaton (DFA) capable of recognizing the regular expression a*(bb—baa)*aa. The DFA's effectiveness was demonstrated through a clear transition table and a detailed transition diagram that provided a visual representation of the state transitions. The implementation in C++ offers a practical tool for string recognition, which is both robust and user-friendly, as evidenced by the instructions included for running the program and the sample executions provided.

The DFA's capacity to accurately accept valid strings and to issue a error message for invalid inputs was thoroughly tested, ensuring that the system behaves as expected for both cases. The inclusion of state transition sequences and screenshots of the program in action offers a transparent view into the DFA's operational processes, enhancing the educational value of this implementation.

Future work could focus on optimizing the DFA's performance, extending the implementation to handle a broader set of regular expressions, or integrating this DFA into larger systems requiring automated string processing. The principles demonstrated here can serve as a foundation for such complex systems, highlighting the versatility and power of automata theory in practical applications of computer science.

References

[1] Xiaoyuan Xie. "Lecture 2: Lexical Analysis." School of Computer Science, Wuhan University, 21 Sept. 2023. Lecture presentation.