Finite State Machine Regex Implementation

Roman Prokhorov

Contents

1 Introduction **2**
1.1 Project Overview . . . 2

2 Theoretical Background **2**
2.1 Finite State Machines . . . 2
2.2 NFAs vs DFAs . . . 3

3 Architecture Analysis **3**
3.1 Class Hierarchy . . . 3
3.1.1 Key Classes . . . 3
3.2 RegexFSM Class . . . 4

4 Implementation Analysis **4**
4.1 State Class . . . 4
4.2 State Implementations . . . 4
4.3 Regex Parsing and FSM Construction . . . 5
4.4 String Matching Algorithm . . . 6

5 Algorithmic Analysis **7**
5.1 Time Complexity . . . 7
5.2 Space Complexity . . . 7

6 Supported Features **8**

7 Limitations and Potential Improvements **8**
7.1 Current Limitations . . . 8
7.2 Potential Improvements . . . 8
7.2.1 Feature Enhancements . . . 8
7.2.2 Performance Optimizations . . . 9
7.2.3 Structural Improvements . . . 9

8 Conclusion **9**

1 Introduction

Regular expressions are powerful tools for pattern matching and text processing. While most common implementations use backtracking algorithms, this project takes an alternative approach by implementing regular expressions using finite state machines. This method provides more predictable performance characteristics and avoids the potential exponential complexity issues associated with backtracking.

1.1 Project Overview

The analyzed system implements a subset of regular expression syntax using the principles of finite state automata. The core functionality is embodied in several key components:

- A hierarchy of state classes representing different regex elements
- A regex parser that transforms pattern strings into state machines
- A pattern matching algorithm that evaluates input strings against the constructed automaton

This implementation is particularly notable for its use of object-oriented design principles to represent automata concepts, with each state type encapsulating specific matching behavior.

2 Theoretical Background

2.1 Finite State Machines

Finite state machines (FSMs) are abstract computational models consisting of:

- A finite set of states
- A set of transitions between states
- A designated start state
- One or more accepting (final) states

FSMs process input sequences by transitioning between states based on each input symbol. If the machine ends in an accepting state after consuming all input, the input is considered to match the pattern encoded by the machine.

2.2 NFAs vs DFAs

This implementation uses a Nondeterministic Finite Automaton (NFA) approach, which differs from Deterministic Finite Automata (DFAs) in several key ways:

- NFAs can have multiple active states simultaneously
- NFAs can include epsilon transitions (transitions without consuming input)
- NFAs can transition to multiple states for the same input symbol

While DFAs are generally more efficient for execution, NFAs are easier to construct directly from regex patterns, making them a suitable choice for this implementation.

3 Architecture Analysis

3.1 Class Hierarchy

The implementation follows an object-oriented approach with an inheritance hierarchy for different state types:

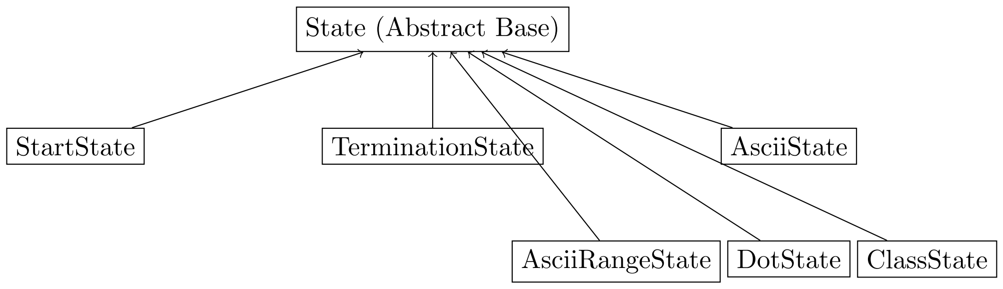

Figure 1: Class hierarchy of the state implementation

3.1.1 Key Classes

- **State**: Abstract base class defining the interface for all states
- **StartState**: Marks the entry point of the automaton
- **TerminationState**: Represents accepting/final states
- **AsciiState**: Matches a specific character
- **DotState**: Wildcard state that matches any single character
- **AsciiRangeState**: Handles character ranges (e.g., a-z)
- **ClassState**: Processes character classes with optional negation

3.2 RegexFSM Class

The `RegexFSM` class serves as the orchestrator for the entire system:

- It parses regex patterns and constructs corresponding state machines
- It manages the process of evaluating input strings against the constructed automaton
- It handles special operators like '*' and '+' by creating appropriate loops and connections between states

4 Implementation Analysis

4.1 State Class

The base `State` class provides the foundation for the state hierarchy:

```cpp
class State {
protected:
    std::vector<std::shared_ptr<State>> nextStates;
    bool isStar = false;
    bool isPlus = false;

public:
    virtual ~State() = default;
    virtual bool checkSelf(const char& symbol) const;
    std::vector<std::shared_ptr<State>> checkNext(const char symbol
    ) const;
    void addNextState(std::shared_ptr<State> state);
    std::vector<std::shared_ptr<State>> getNextStates() const;
    void markStar();
    void markPlus();
    bool getStar() const;
    bool getPlus() const;
};
```

Listing 1: State class interface

Key methods:

- `checkSelf`: Determines if a state accepts a particular symbol
- `checkNext`: Finds all valid next states for a given input symbol
- `addNextState`: Establishes a transition to another state
- `markStar/markPlus`: Flags for repetition operators

4.2 State Implementations

Each derived state class implements the `checkSelf` method to define its specific matching behavior:

```cpp
// DotState matches any character
bool DotState::checkSelf(const char& symbol) const {
    return true;
}

// AsciiState matches a specific character
bool AsciiState::checkSelf(const char& symbol) const {
    return this->symbol == symbol;
}

// AsciiRangeState matches characters within defined ranges
bool AsciiRangeState::checkSelf(const char& symbol) const {
    for (const auto& range : this->charRanges) {
        if (symbol >= range.first && symbol <= range.second) {
            return !this->isNegated;
        }
    }
    return this->isNegated;
}

// ClassState matches characters against a set of states
bool ClassState::checkSelf(const char& symbol) const {
    for (const auto& checkingState : this->checkingStates) {
        if (checkingState->checkSelf(symbol)) {
            return !this->isNegated;
        }
    }
    return this->isNegated;
}
```

Listing 2: Implementation of specific state types

4.3 Regex Parsing and FSM Construction

The `RegexFSM` class parses regex patterns and constructs the corresponding state machine:

```cpp
void RegexFSM::initializeRegex(const std::string& regex) {
    size_t regexSize = regex.size();
    this->startingState = std::make_shared<StartState>();
    std::vector<std::shared_ptr<State>> states = { startingState };

    for (size_t index = 0; index < regexSize; ++index) {
        char current = regex[index];

        if (current == '*') {
            if (states.size() < 2) throw std::runtime_error("
    Nothing to repeat with '*'");
            states[states.size() - 1]->markStar();
            continue;
        }
        else if (current == '+') {
            if (states.size() < 2) throw std::runtime_error("
    Nothing to repeat with '+'");
            states[states.size() - 1]->markPlus();
            continue;
        }

        if (current == '[') {
            states.push_back(this->parseClassState(regex, index));
        }
        else {
            states.push_back(this->parseNewState(current));
        }
    }

    std::shared_ptr<TerminationState> endingState = std::
    make_shared<TerminationState>();
    states.back()->addNextState(endingState);
    states.push_back(endingState);
    this->connectStates(states);
}
```

Listing 3: RegexFSM initialization

This method:

- Creates a starting state
- Iteratively processes each character in the regex pattern
- Handles special operators like '*' and '+'
- Parses character classes when encountered
- Adds a termination state at the end
- Connects all states according to the regex semantics

4.4 String Matching Algorithm

The matching algorithm is implemented in the `checkString` method:

```
bool RegexFSM::checkString(const std::string& input) const {
    std::vector<std::shared_ptr<State>> currentStates = { this->
    startingState };

    for (const char& symbol : input) {
        std::vector<std::shared_ptr<State>> nextStates;

        for (const auto& state : currentStates) {
            std::vector<std::shared_ptr<State>> validNext = state->
    checkNext(symbol);
            nextStates.insert(nextStates.end(), validNext.begin(),
    validNext.end());
        }

        if (nextStates.empty()) {
            return false;
        }

        currentStates = nextStates;
    }

    for (const auto& state : currentStates) {
        for (const auto& nextState : state->getNextStates()) {
            if (std::dynamic_pointer_cast<TerminationState>(
    nextState)) {
                return true;
            }
        }
    }

    return false;
}
```

Listing 4: String matching implementation

The algorithm:

- Starts with only the initial state active
- For each input character, finds all valid next states from all currently active states
- If no valid next states exist, the match fails
- After processing all input, checks if any current state can reach the termination state

5 Algorithmic Analysis

5.1 Time Complexity

The time complexity of the regex matching algorithm is $O(m \cdot n)$, where:

- m is the length of the input string
- n is the number of states in the FSM

This results from:

- Processing each character of the input string exactly once: $O(m)$
- For each character, potentially checking all states in the worst case: $O(n)$

This is generally more efficient than backtracking algorithms, which can have exponential complexity in the worst case.

5.2 Space Complexity

The space complexity is $O(n)$, where n is the number of states in the FSM. This accounts for:

- Storage for all state objects
- Storage for transitions between states
- The set of currently active states during matching

6 Supported Features

The implementation supports a subset of regular expression syntax:

- **Basic Character Matching**: Literals that match themselves
- **Wildcard (.)**: Matches any single character
- **Kleene Star (*)**: Matches zero or more occurrences of the preceding element
- **Plus Operator (+)**: Matches one or more occurrences of the preceding element
- **Character Classes**:
  - `[abc]`: Matches any character in the set
  - `[a-z]`: Matches any character in the specified range
  - `[^0-9]`: Negated classes match any character not in the set

7 Limitations and Potential Improvements

7.1 Current Limitations

The implementation lacks support for several common regex features:

- No grouping with parentheses `()`
- No alternation with the pipe operator `|`
- No precise quantifiers like {`n`}, {`n,`}, or {`n,m`}
- No support for anchors `^` and `$`
- No backreferences or lookahead/lookbehind assertions

7.2 Potential Improvements

Several enhancements could extend the functionality and performance:

7.2.1 Feature Enhancements

- Add support for grouping and alternation
- Implement anchors for start/end of line matching
- Add precise quantifiers for repetition
- Support for word boundaries and other zero-width assertions

7.2.2 Performance Optimizations

- Convert NFA to DFA for more efficient matching
- Optimize state transition lookup with more efficient data structures
- Add caching for frequently used patterns
- Implement parallel matching for large inputs

7.2.3 Structural Improvements

- Enhance error handling with more descriptive messages
- Add visualization tools for the generated state machines

8 Conclusion

This finite state machine regex implementation is a clear and practical use of automata theory for pattern matching. It uses an NFA-based approach, which offers consistent performance and supports a useful subset of regular expression features.

Although it doesn't support all the features of more advanced regex engines, the simple and well-structured design makes it easy to understand and extend.

Overall, it's a strong alternative to traditional backtracking regex engines, particularly in cases where stable performance and a focused feature set are important.