Regular Expression Finite State Machine

May 10, 2025

1 Introduction

This project implements a Finite State Machine (FSM) for processing regular expressions. The implementation creates a non-deterministic finite automaton (NFA) from a given regular expression pattern and provides functionality to check whether input strings match the pattern.

2 Supported regex patterns

The implementation supports the following regular expression features:

- Basic character matching (e.g., "abc" matches exactly "abc")
- Wildcard character "." (matches any single character)
- Repetition operators:
  - "*" (zero or more occurrences)
  - "+" (one or more occurrences)
- Character classes with the following capabilities:
  - Basic character sets: `[abc]` (matches any of a, b, or c)
  - Character ranges: `[a-z]` (matches any lowercase letter)
  - Negated character classes: `[^abc]` (matches any character except a, b, or c)

3 States of FSM

The FSM consists of the following state types:

- **StartState**: The initial state of the automaton
- **AsciiState**: Accepts a specific ASCII character
- **DotState**: Accepts any ASCII character (implements the "." wildcard)

- **CharClassState**: Accepts characters based on inclusion or exclusion from a set

Each state maintains:

- A unique identifier
- A set of normal transitions to other states
- A set of ϵ-transitions (transitions without consuming input)
- A flag indicating whether it is an accepting state

4 Algorithm of machine creation

Firstly an empty machine with only the StartState is initialized. Then the regex pattern is processed character by character:

- If the char is ".", a DotState is created.
- If the char is "[", the character-class parser is invoked. It returns a set of characters, a Boolean for negation, and an updated index. Then a CharClassState is created.
- If the char is any other ASCII character, an AsciiState with that char is created.
- After creating the state, if the next pattern char is "*" or "+", a loop transition is added:
  - For "*", an ϵ-transition back to the current state is added.
  - For "+", a normal transition consuming the state's symbol is added.
- If at end-of-pattern, the current state is marked accepting.
- The current state becomes the "previous" for the next iteration.

```python
def __init_machine(self, regex_expr: str) -> None:
    """
    function initializes FSM
    """
    prev_state = self.start_state

    i, state_id = 0, 1
    while i < len(regex_expr):
        char = regex_expr[i]

        if char in ("*", "+"):
            i += 1
            continue

        if char == ".":
            cur_state = DotState(state_id)
            state_id += 1
        elif char == "[":
            char_set, is_negated, i = self.__parse_char_class(regex_expr, i+1)
            cur_state = CharClassState(state_id, char_set, is_negated)
            state_id += 1
        elif char.isascii():
            cur_state = AsciiState(state_id, char)
            state_id += 1
        else:
            raise AttributeError(f"Character '{char}'' is not supported")

        next_char = regex_expr[i+1] if i+1 < len(regex_expr) else None

        match next_char:
            case "*":
                cur_state.add_loop()
                if prev_state is not None:
                    prev_state.epsilon_transition_states.add(cur_state)
            case "+":
                cur_state.add_loop()
                if prev_state is not None:
                    prev_state.next_states.add(cur_state)
            case _:
                if prev_state is not None:
                    prev_state.next_states.add(cur_state)

        if i == len(regex_expr)-1 or (i == len(regex_expr)-2 and next_char in ("+", "*")):
            cur_state.is_accept_state = True

        prev_state = cur_state
        i += 1
```

Figure 1: Code of function initializing machine

5 Algorithm of string checking

During matching, the machine tracks a set of current states—initially the ϵ-closure of the start state. For each input character:

- From each current state, follow transitions labeled with the input char.
- Collect all reachable states.
- Compute the ϵ-closure of that set.
- That becomes the new current-states set.

After the entire string is consumed, if any current state is accepting, the match succeeds. Then the machine resets its current-states to the ϵ-closure of the start state for the next query.

```
def __update_cur_states(self, char: str) -> None:
    """
    function updates current states
    """
    new_states = set()
    for state in self.__cur_states:
        next_states = state.get_next_states(char)

        for next_state in next_states:
            new_states.update(next_state.epsilon_closure())

    self.__cur_states = new_states

def __reset_machine(self) -> None:
    """
    function resets FSM
    """
    self.__cur_states = self.__start_states

def check_string(self, string: str) -> bool:
    """
    checks whether string is accepted by FSM
    """
    if self.start_state is None:
        return False

    for char in string:
        self.__update_cur_states(char)

    for state in self.__cur_states:
        if state.is_accept_state:
            self.__reset_machine()
            return True

    self.__reset_machine()
    return False
```

Figure 2: Functions for checking string match with regex

6 Visualization

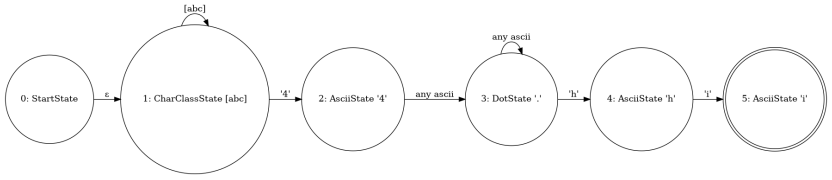

Figure 3: FSM for regular expression "[a–c]*4.+hi"

7 Project Structure

The project is organized into several key files:

- **regex.py**: Module with implemented `RegexFSM` to check match of strings with regex pattern.
- **visualization.py**: Module to create visualization of compiled machine (generated by AI).
- **test_regex.py**: Module with unit tests (generated by AI).