Звіт

Реалізація скінченного автомату для регулярних виразів

Мацелюх Максим

10.05.2025

1 Вступ

У цьому звіті розглянуто реалізацію недетермінованого скінченного автомату (НСА) для перевірки рядків на відповідність заданому регулярному виразу.

2 Реалізація

У цій роботі використано реалізацію на базі недетермінованого скінченного автомата без використання підкласів класу `State`, натомість був створений клас `NFAFragment`, з методами для обробки *, . і +.

У початковому шаблоні використовувалася складна ієрархія класів (`AsciiState`, `DotState`, `StarState`, `PlusState` тощо), кожен зі своєю логікою перевірки символів і списком `next_states`. Такий підхід швидко ускладнюється при комбінаціях операторів: наприклад, як правильно поєднати оператор `+` у підвиразі всередині дужок або як реалізувати `*` над кількома символами підряд. Оператор `*` і особливо `+` потребують «повернутися» до початку підвиразу або пропустити обробку без споживання символу. У початковій структурі довелося б заводити додаткові стани та розкидати штучні переходи по різних підкласах. За алгоритмом Томпсона ми чітко бачимо, що для `*` потрібно чотири ε-переходи (два на вхід, два на вихід), а для `+` — два ε-переходи (петля з кінця на початок плюс вихід). Це легко інкапсулювати у двох функціях `apply_star` та `apply_plus` замість розпорошення логіки по методах кожного стану.

Основними структурами є класи `State` (стан) і `NFA` (автомат). Кожен стан зберігає список переходів за конкретними символами та список епсілон-переходів. Наприклад:

```
class State:
    def __init__(self):
        self.transitions = {} # : , :
        self.epsilon = [] #    -
        self.is_final = False #
```

Автомат має початковий стан і набір фінальних станів:

```
class NFA:
    def __init__(self, start, accepts):
        self.start = start
        self.accepts = set(accepts)
```

Для побудови автомата з регулярного виразу використано варіант алгоритму Томпсона. Спочатку регулярний вираз переводиться у постфіксну

форму (для спрощення обробки операторів у правильному порядку). Потім кожен символ або оператор послідовно обробляється стеком:

```
stack = []
for token in postfix_regex:
    if token == '*':
        nfa = stack.pop()
        stack.append(apply_star(nfa))
    elif token == '.':
        nfa2 = stack.pop()
        nfa1 = stack.pop()
        stack.append(concat(nfa1, nfa2))
    elif token == '+':
        nfa2 = stack.pop()
        nfa1 = stack.pop()
        stack.append(union(nfa1, nfa2))
    else:
        #   ()
        stack.append(build_symbol_nfa(token))
nfa = stack.pop()
```

Тут функції `apply_star`, `apply_plus`, `build_nfa` створюють НСА для відповідних операцій. Такий підхід дозволяє зрозуміло описати роботу з регулярними виразами і побудову автоматів.

Ключовою зміною в порівнянні з вихідним шаблоном було відмовитися від складного патерну `State` з багатьма підкласами і перейти до більш простої архітектури. Використання НСА з епсілон-переходами робить код зрозумілішим і дозволяє легше реалізувати операції злиття (конкатенації, об'єднання, заміщення зіркою) без значних ускладнень.

3 Висновки

У цій роботі було реалізовано простий скінченний автомат (НСА) для перевірки відповідності рядків регулярним виразам. Побудова автомата базувалася на алгоритмі Томпсона. Основною перевагою вибраного підходу є зрозумілість коду та можливість одразу візуалізувати автомат. Обрана архітектура у вигляді NFA з епсілон-переходами працює достатньо ефективно.

Під час роботи виникали труднощі з коректною обробкою операторів і забезпеченням правильного порядку виконання дій (саме тому використовувався перехід до постфіксної форми виразу). Проте отриманий

автомат успішно пройшов усі тестові сценарії.