Лабораторна робота 3

Стратюк Арсеній

10 травня 2025 р.

Зміст

1 Вступ

Регулярні вирази є потужним інструментом для пошуку та маніпулювання текстом на основі шаблонів. Вони широко використовуються в програмуванні, обробці текстових даних та багатьох інших галузях. В основі механізму роботи регулярних виразів лежать скінченні автомати – математичні моделі обчислень, які можуть перебувати в одному з скінченної кількості станів.

Метою даної лабораторної роботи є розробка програмної реалізації скінченного автомату, здатного обробляти деяку підмножину синтаксису регулярних виразів. Це включає розбір регулярного виразу, побудову відповідного скінченного автомату та перевірку, чи відповідає заданий рядок цьому автомату.

У звіті детально описано теоретичні основи регулярних виразів та скінченних автоматів, процес проектування та реалізації програмного модуля на мові Python, а також наведено результати тестування розробленої системи.

2 Теоретичні відомості

2.1 Регулярні вирази

Регулярний вираз — це послідовність символів, що визначає шаблон пошуку. Регулярні вирази використовуються для перевірки відповідності рядків певному шаблону, а також для пошуку та заміни частин рядків.

Основні елементи регулярних виразів, що розглядаються в даній роботі:

- **Літерали**: Звичайні символи (наприклад, `a`, `b`, `1`, `2`), які відповідають самі собі.
- **Метасимвол крапки (`.`):** Відповідає будь-якому одному символу (окрім символу нового рядка, залежно від реалізації).
- **Символьні класи (`[]`):** Дозволяють вказати набір символів, будь-який з яких може збігатися. Наприклад, `[abc]` відповідає `a`, `b` або `c`. Діапазони символів, такі як `[a-z]`, відповідають будь-якій малій літері латинського алфавіту. Негативні символьні класи (`[^...]`), наприклад `[^0-9]`, відповідають будь-якому символу, що не є цифрою.
- **Квантифікатори**:
  - **Зірочка Кліні (`*`):** Відповідає нулю або більше повторень попереднього елемента. Наприклад, `a*` відповідає , `a`, `aa` і т.д.
  - **Плюс Кліні (`+`):** Відповідає одному або більше повторень попереднього елемента. Наприклад, `a+` відповідає `a`, `aa`, але не порожньому рядку.

2.2 Скінченні автомати

Скінченний автомат (СА) або машина зі скінченною кількістю станів (англ. Finite State Machine, FSM) — це абстрактна машина, яка може перебувати в одному зі скінченної множини станів. Машина може переходити з одного стану в інший у відповідь на вхідні дані (символи). Переходи між станами називаються транзиціями. Один зі станів позначається як початковий, а деякі стани можуть бути позначені як кінцеві (або приймаючі).

Формально, скінченний автомат можна визначити як п'ятірку $(Q, \Sigma, \delta, q_0, F)$, де:

- Q — скінченна множина станів.
- Σ — скінченна множина вхідних символів (алфавіт).
- δ — функція переходів: $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to P(Q)$ для НСА або $\delta : Q \times \Sigma \to Q$ для ДСА. ϵ позначає порожній рядок (епсилон-перехід). $P(Q)$ — булеан множини Q.
- $q_0 \in Q$ — початковий стан.
- $F \subseteq Q$ — множина приймаючих (кінцевих) станів.

2.2.1 Детерміновані скінченні автомати (ДСА)

У детермінованому скінченному автоматі (ДСА, англ. Deterministic Finite Automata, DFA) для кожного стану та кожного вхідного символу існує рівно один перехід до наступного стану.

2.2.2 Недетерміновані скінченні автомати (НСА)

У недетермінованому скінченному автоматі (НСА, англ. Nondeterministic Finite Automata, NFA) для пари (стан, вхідний символ) може існувати декілька можливих наступних станів. Будь-який НСА може бути перетворений на еквівалентний ДСА.

2.2.3 Перетворення регулярних виразів у скінченні автомати

Існує тісний зв'язок між регулярними виразами та скінченними автоматами: для будь-якого регулярного виразу можна побудувати скінченний автомат (зазвичай НСА), який розпізнає ту саму мову, і навпаки. У даній роботі реалізовано прямий підхід до побудови НСА на основі структури регулярного виразу.

2.3 Операції над регулярними виразами та їх представлення в СА

2.3.1 Конкатенація

Регулярний вираз R_1R_2 (конкатенація) відповідає послідовності, де спочатку йде рядок, що відповідає R_1, а за ним — рядок, що відповідає R_2. В СА це реалізується шляхом з'єднання кінцевого стану автомата для R_1 з початковим станом автомата для R_2.

2.3.2 Зірочка Кліні (*)

Регулярний вираз R^* відповідає нулю або більше повторень рядка, що відповідає R. Для побудови СА для R^* додаються епсилон-переходи, що дозволяють "обійти"автомат для R (нуль повторень) або повернутися до його початку після успішного розпізнавання R (одне або більше повторень).

2.3.3 Плюс Кліні (+)

Регулярний вираз R^+ відповідає одному або більше повторень рядка, що відповідає R. Це еквівалентно RR^*. В СА це реалізується так, що автомат для R повинен бути пройдений хоча б один раз, після чого можливі повторні проходи.

2.3.4 Символьні класи ([...])

Символьний клас, наприклад `[a-z]`, відповідає одному символу з указаного набору. В СА це може бути представлено станом, який переходить у наступний стан при отриманні будь-якого символу з цього класу.

2.3.5 Метасимвол крапки (.)

Метасимвол . відповідає будь-якому одному символу. В СА це стан, який переходить у наступний при отриманні будь-якого символу з алфавіту.

3 Опис реалізації

Програмна реалізація скінченного автомату для обробки регулярних виразів виконана на мові Python. Основна ідея полягає у створенні структури станів, які з'єднуються між собою відповідно до правил регулярного виразу.

3.1 Загальна архітектура

Система складається з кількох ключових класів:

- `State`: Абстрактний базовий клас для всіх типів станів.
- `StartState`: Спеціальний стан, що позначає початок автомата.
- `TerminationState`: Спеціальний стан, що позначає успішне завершення (приймаючий стан).
- `AsciiState`: Стан, що відповідає конкретному ASCII символу.
- `DotState`: Стан, що відповідає метасимволу . (будь-який символ).
- `CharacterClassState`: Стан, що відповідає символьному класу (наприклад, `[a-z]`).
- `RegexFSM`: Основний клас, що інкапсулює логіку побудови скінченного автомата з регулярного виразу та перевірки рядків.

3.2 Класи станів

3.2.1 Абстрактний клас `State`

Клас `State` є базовим для всіх інших станів. Він визначає загальний інтерфейс:

- `__init__()`: Ініціалізує стан порожнім списком наступних станів (`next_states`).
- `check_self(char)`: Абстрактний метод, який має бути реалізований у похідних класах. Перевіряє, чи приймає поточний стан заданий символ `char`.

- `check_next(next_char)`: Знаходить наступний стан зі списку `next_states`, який приймає символ `next_char`. Якщо такого стану немає, генерує виняток.

```python
from abc import ABC, abstractmethod

class State(ABC):
    def __init__(self) -> None:
        self.next_states = []

    @abstractmethod
    def check_self(self, char: str) -> bool:
        pass

    def check_next(self, next_char: str):
        for state in self.next_states:
            if state.check_self(next_char):
                return state
        raise Exception("Rejected string")
```

Лістинг 1: Абстрактний клас State (фрагмент)

3.2.2 Клас `StartState`

Представляє початковий стан автомата. Не споживає символів. Метод `check_self` завжди повертає `False`.

3.2.3 Клас `TerminationState`

Представляє кінцевий (приймаючий) стан. Не споживає символів. Метод `check_self` завжди повертає `False`. Метод `check_next` завжди генерує виняток, оскільки з кінцевого стану немає переходів.

3.2.4 Клас `AsciiState`

Стан, що відповідає одному конкретному символу.

```python
class AsciiState(State):
    def __init__(self, symbol: str) -> None:
        super().__init__()
        self.curr_sym = symbol

    def check_self(self, curr_char: str) -> bool:
        return self.curr_sym == curr_char
```

Лістинг 2: Клас AsciiState (фрагмент)

3.2.5 Клас `DotState`

Стан для метасимволу `.` (крапка). Приймає будь-який одиночний символ. Метод `check_self` завжди повертає `True`.

3.2.6 Клас `CharacterClassState`

Стан для символьних класів, таких як `[a-z0-9]` або `[^0-9]`. Під час ініціалізації розбирає визначення класу (наприклад, `a-z0-9`) і формує множину дозволених символів. Враховує можливість негації класу (символ `^` на початку).

```python
class CharacterClassState(State):
    def __init__(self, class_definition: str) -> None:
        super().__init__()
        self.allowed_chars = set()
        self.negated = False

        if class_definition and class_definition[0] == '^':
            self.negated = True
            class_definition = class_definition[1:]
        # ... (processing ranges and individual characters) ...

```

```
    def check_self(self, char: str) -> bool:
        if self.negated:
            return char not in self.allowed_chars
        return char in self.allowed_chars
```

Лістинг 3: Клас CharacterClassState (фрагмент)

3.3 Клас `RegexFSM`

Цей клас відповідає за компіляцію регулярного виразу в скінченний автомат та перевірку рядків на відповідність.

3.3.1 Метод `_build_fsm(regex)`

Цей приватний метод розбирає вхідний рядок регулярного виразу `regex` і створює послідовність станів.

- Ітеративно обробляє символи регулярного виразу.
- Для `'['`: знаходить відповідну закриваючу дужку `']'`, виділяє визначення класу і створює `CharacterClassState`.
- Для `'.'`: створює `DotState`.
- Для `'*'` або `'+'`: модифікує попередній стан, позначаючи його для квантифікації. Зберігає пару (стан, квантифікатор).
- Для звичайних символів: створює `AsciiState`.
- В кінці додає `TerminationState`.
- Викликає `_connect_states` для з'єднання створених станів.

Обробка помилок включає перевірку на незакриті дужки та квантифікатори без попереднього елемента.

3.3.2 Метод `_connect_states(start, states)`

Цей приватний метод з'єднує стани, передані у списку `states`, починаючи зі стану `start`. Логіка з'єднання враховує квантифікатори:

- **Для `*` (зірочка Кліні)**:
  1. Додається шлях для "обходу"стану з квантифікатором (з поточного стану `current` до наступного стану після квантифікованого).
  2. Додається петля на самому квантифікованому стані.
  3. Додається шлях для входу в квантифікований стан з поточного.
  4. Поточний стан оновлюється на квантифікований стан.
- **Для `+` (плюс Кліні)**:
  1. Додається петля на самому квантифікованому стані.
  2. Додається шлях для входу в квантифікований стан з поточного.
  3. Поточний стан оновлюється на квантифікований стан.
- **Для звичайних станів**: поточний стан з'єднується з наступним, і наступний стає поточним.

3.4 Приклади побудови автоматів (схеми)

Нижче наведено схеми скінченних автоматів для деяких простих регулярних виразів, як вони реалізовані в коді. S_0 — початковий стан, S_T — кінцевий (приймаючий) стан. Переходи без міток є ϵ-переходами (в реалізації це означає, що стан додається до `next_states` і перевірка символу відбувається вже в ньому).

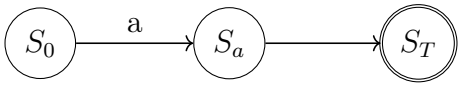

Рис. 1: СА для регулярного виразу `a`

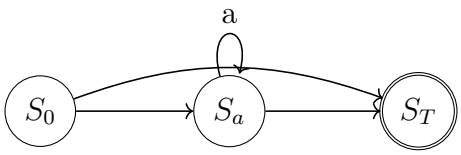

Рис. 2: СА для регулярного виразу `a*` (спрощена схема логіки зв'язків)

Примітка до рис. 2: Схема показує логічні зв'язки. S_0 може перейти в S_a (якщо наступний символ 'a') або безпосередньо в S_T (якщо рядок порожній або 'a' не зустрічається). S_a при символі 'a' може залишитися в S_a або перейти в S_T.

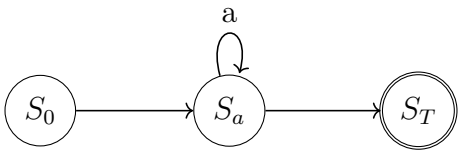

Рис. 3: СА для регулярного виразу `a+` (спрощена схема логіки зв'язків)

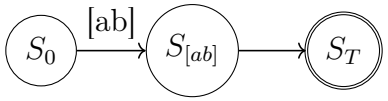

Рис. 4: СА для регулярного виразу `[ab]`

4 Тестування та результати

Тестування реалізованого скінченного автомату проводилося за допомогою набору тестових сценаріїв, що охоплюють різні аспекти підтримуваного синтаксису регулярних виразів. Для автоматизації тестування використовувався модуль `unittest` мови Python.

4.1 Тестові сценарії

Були розроблені тести для перевірки:

- **Окремих класів станів**: `DotState`, `AsciiState`, `CharacterClassState` (включаючи діапазони, окремі символи та негацію), `StartState`, `TerminationState`.

- **Побудови FSM**: коректність обробки валідних регулярних виразів та генерація помилок для невалідних (наприклад, `*`, `+`, `[a-z`).
- **Функції** `check_string`:
  - Прості літерали: `a`, `abc`.
  - Метасимвол крапки: `.`, `a.c`.
  - Символьні класи: `[a-z]`, `[0-9]`, `[^0-9]`.
  - Квантифікатори: `a*`, `a+`, `a*b`, `a+b`.
  - Комбіновані складні шаблони: `a*4.+hi`, `[a-z]+[0-9]`.

Файл з тестами `test_regex.py` містить детальні перевірки.

4.2 Результати виконання

Основний скрипт `regex.py` містить приклад використання реалізованого FSM:

```
# ... (код класів) ...

if __name__ == "__main__":
    print("Original tests:")
    regex_pattern = "a*4.+hi"
    regex_compiled = RegexFSM(regex_pattern)
    print(f"Regex: '{regex_pattern}', String: 'aaaaaa4uhi', Matches: {regex_compiled.check_string('aaaaaa4uhi')}") #
     True
    print(f"Regex: '{regex_pattern}', String: '4uhi', Matches: {regex_compiled.check_string('4uhi')}")           # True
    print(f"Regex: '{regex_pattern}', String: 'meow', Matches: {regex_compiled.check_string('meow')}")           #
     False

    print("\nCharacter class tests:")
    regex_with_class = "[a-z]+[0-9]"
    class_regex = RegexFSM(regex_with_class)
    print(f"Regex: '{regex_with_class}', String: 'abc1', Matches: {class_regex.check_string('abc1')}")     # True
    print(f"Regex: '{regex_with_class}', String: 'xyz9', Matches: {class_regex.check_string('xyz9')}")       # True
    print(f"Regex: '{regex_with_class}', String: 'ABC1', Matches: {class_regex.check_string('ABC1')}")     # False
    print(f"Regex: '{regex_with_class}', String: '1', Matches: {class_regex.check_string('1')}")          # False

    print("\nNegated character class test:")
    negated_class = "[^0-9]+"
    not_digit_regex = RegexFSM(negated_class)
    print(f"Regex: '{negated_class}', String: 'abc', Matches: {not_digit_regex.check_string('abc')}")     # True
    print(f"Regex: '{negated_class}', String: '123', Matches: {not_digit_regex.check_string('123')}")     # False
```

Лістинг 4: Приклад використання з regex.py

Усі тести, визначені в `test_regex.py`, також успішно проходять, що підтверджує коректність реалізації для підтримуваної підмножини регулярних виразів.

5 Висновки

В ході виконання лабораторної роботи було розроблено програмну реалізацію скінченного автомату для обробки регулярних виразів на мові Python. Реалізація підтримує основні конструкції: літерали, метасимвол крапки, символьні класи (включаючи діапазони та негацію), а також квантифікатори "зірочка Кліні"(*) та "плюс Кліні"(+).

Була спроектована ієрархія класів для представлення різних станів автомата, що дозволило створити гнучку та розширювану архітектуру. Механізм побудови автомата (`_build_fsm` та `_connect_states`) коректно трансформує рядок регулярного виразу у відповідну структуру станів та переходів. Алгоритм перевірки рядка (`check_string`) ефективно симулює роботу недетермінованого скінченного автомата.

Розроблена система була ретельно протестована за допомогою модульних тестів, які підтвердили її працездатність та відповідність поставленим вимогам для обраної підмножини функціональності регулярних виразів.

Дана робота дозволила поглибити розуміння теоретичних основ регулярних виразів та скінченних автоматів, а також отримати практичний досвід їх реалізації.

6 Можливі напрямки для покращення

Хоча поточна реалізація успішно справляється з поставленими завданнями, існують напрямки для її подальшого розвитку та вдосконалення:

- **Розширення підтримки синтаксису регулярних виразів**:
  - Квантифікатор `?` (нуль або одне входження).
  - Квантифікатори з точною кількістю повторень (`{n}`, `{n,}`, `{n,m}`).
  - Групування за допомогою дужок `()` та зворотні посилання.
  - Оператор альтернативи `|` (логічне АБО).
  - Якорі `^` (початок рядка) та `$` (кінець рядка).
  - Підтримка спеціальних символьних класів (`\d`, `\w`, `\s` тощо).
  - Екранування метасимволів за допомогою `\`.
- **Оптимізація продуктивності**: Для дуже складних регулярних виразів або довгих рядків поточна реалізація НСА може бути не найшвидшою.