Лабораторна робота №3

Шевчук Дарина

11 травня 2025 р.

Мета: Реалізація скінченного автомату для обробки спрощених регулярних виразів

1 Мета та тема

У даній лабораторній роботі я реалізувала скінченний автомат (finite-state machine, FSM) для обробки спрощених регулярних виразів. Представлена реалізація підтримує наступні можливості регулярних виразів:

- ASCII символи (літери, цифри, спеціальні символи)
- Оператор .” (відповідає будь-якому символу)
- Оператор *” (зірка Кліні: нуль або більше повторень)
- Оператор “+” (один або більше повторень)

2 Теорія

2.1 Скінченні автомати

Скінченний автомат (FSM) - це математична модель обчислень, яка може знаходитися в одному з кінцевого числа станів у будь-який момент часу. FSM може переходити з одного стану в інший у відповідь на деякі входи; переходи зі стану в стан називаються переходами. Формально, детермінований скінченний автомат можна представити як п'ятірку $(Q, \Sigma, \delta, q_0, F)$, де:

- Q - кінцева множина станів
- Σ - вхідний алфавіт
- $\delta : Q \times \Sigma \to Q$ - функція переходу
- $q_0 \in Q$ - початковий стан
- $F \subseteq Q$ - множина заключних (приймаючих) станів

У випадку недетермінованих скінченних автоматів (NFA), які використовуються в даній реалізації, функція переходу має вигляд $\delta : Q \times \Sigma \to \mathcal{P}(Q)$, тобто переводить пару (стан, символ) у множину станів.

2.2 Регулярні вирази та їх зв'язок з FSM

Регулярні вирази - це формальна мова для опису шаблонів в текстових даних. Основні оператори в регулярних виразах:

- Конкатенація: послідовне розташування символів
- Зірка Кліні (*): нуль або більше повторень
- Плюс (+): один або більше повторень

Для побудови FSM за регулярним виразом використовується алгоритм компіляції регулярного виразу в недетермінований скінченний автомат (NFA).

3 Реалізація

3.1 Загальна архітектура

Реалізація складається з абстрактного базового класу `State`, його конкретних підкласів для різних типів станів та класу `RegexFSM`, який представляє скінченний автомат. Ієрархія класів:

- `State` (абстрактний базовий клас)
  - `StartState` (початковий стан)
  - `TerminationState` (кінцевий стан)
  - `DotState` (стан для оператора ".")
  - `AsciiState` (стан для конкретного символу)
  - `StarState` (стан для оператора "*")
  - `PlusState` (стан для оператора "+")
- `RegexFSM` (головний клас скінченного автомату)

3.2 Аналіз коду

3.2.1 Імпорти та базовий абстрактний клас

Лістинг 1: Імпорти та базовий абстрактний клас

```
from future import annotations
from abc import ABC, abstractmethod
class State(ABC):

def __init__(self) -> None:
    self.next_states: list[State] = []
```

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

- Використовується `from __future__ import annotations` для підтримки відкладеного виведення типів.
- Імпорт `ABC` та `abstractmethod` з модуля `abc` для створення абстрактного класу.
- Оголошення абстрактного класу `State`.
- Конструктор класу, який створює порожній список `next_states`.
- Метод `check_self` визначається як абстрактний.

3.2.2 Конкретні підкласи станів

Лістинг 2: StartState - початковий стан

```
class StartState(State):

def check_self(self, char: str) -> bool:
    return True
```

- Визначення класу `StartState`, який успадковується від `State` і представляє початковий стан автомату.
- Перевизначення методу `check_self`, який завжди повертає `True`, оскільки початковий стан не має умов переходу (функціонує як епсилон-перехід).

Лістинг 3: TerminationState - кінцевий стан

```
class TerminationState(State):

def check_self(self, char: str) -> bool:
    return False
```

- Визначення класу `TerminationState`, який представляє кінцевий (приймаючий) стан автомату.

- Перевизначення методу `check_self`, який завжди повертає `False`, оскільки кінцевий стан не приймає жодних символів і використовується для сигналізації кінця вводу.

Лістинг 4: DotState - стан для оператора "."

```
class DotState(State):

def check_self(self, char: str) -> bool:
    return True
```

- Визначення класу `DotState`, який представляє стан для оператора `"."` в регулярних виразах.
- Перевизначення методу `check_self`, який завжди повертає `True`, оскільки оператор `"."` відповідає будь-якому символу.

Лістинг 5: AsciiState - стан для конкретного символу

```
class AsciiState(State):

def __init__(self, symbol: str) -> None:
    super().__init__()
    self.curr_sym = symbol

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

- Визначення класу `AsciiState`, який представляє стан для конкретного ASCII-символу.
- Конструктор класу, який викликає конструктор базового класу і зберігає переданий символ у атрибуті `curr_sym`.
- Перевизначення методу `check_self`, який повертає `True`, якщо переданий символ збігається з символом стану, і `False` в іншому випадку.

Лістинг 6: StarState - стан для оператора

```
class StarState(State):
```

```
def __init__(self, checking_state: State):
    super().__init__()
    self.checking_state = checking_state
    self.next_states.append(self)

def check_self(self, char: str) -> bool:
    return self.checking_state.check_self(char)
```

- Визначення класу `StarState`, який реалізує оператор зірки Кліні `*` (нуль або більше повторень).
- Конструктор класу, який приймає стан `checking_state`, на який діє оператор `*`. Важливо відзначити, що він додає сам себе у список `next_states`, що дозволяє реалізувати множинні повторення.
- Перевизначення методу `check_self`, який делегує перевірку стану `checking_state`, тобто повертає те ж значення, що і `check_self` стану, на який діє оператор `*`.

Лістинг 7: PlusState - стан для оператора "+"

```
class PlusState(State):

def __init__(self, checking_state: State):
    super().__init__()
    self.checking_state = checking_state
    self.next_states.append(self)

def check_self(self, char: str) -> bool:
    return self.checking_state.check_self(char)
```

- Визначення класу `PlusState`, який реалізує оператор `"+"` (один або більше повторень).
- Конструктор класу, аналогічний конструктору `StarState`. Додає себе до списку `next_states` для підтримки повторень.
- Перевизначення методу `check_self`, аналогічно `StarState`. Це підкреслює схожість між операторами `"*"` та `"+"`. Основна відмінність полягає в тому, як ці стани інтегруються в автомат у класі `RegexFSM`.

3.2.3 Основний клас RegexFSM

Лістинг 8: RegexFSM - основна частина

```
class RegexFSM:

def __init__(self, regex_expr: str) -> None:
    self.start_state = StartState()
    self.states: list[State] = [self.start_state]

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

        if char.isascii() and char not in "*+.":
            new_state = AsciiState(char)
            self.states[-1].next_states.append(new_state)
            self.states.append(new_state)
            i += 1

        elif char == ".":
            new_state = DotState()
            self.states[-1].next_states.append(new_state)
            self.states.append(new_state)
            i += 1

        elif char == "*":
            base_state = self.states[-1]
            star_state = StarState(base_state)
            star_state.next_states.extend(base_state.next_states)
            base_state.next_states.extend(star_state.next_states)
            self.states[-2].next_states.append(star_state)
            self.states.append(star_state)
            i += 1

        elif char == "+":
            base_state = self.states[-1]
            plus_state = PlusState(base_state)
            plus_state.next_states.extend(base_state.next_states)
            self.states[-2].next_states.append(plus_state)
            self.states.append(plus_state)
```

```
            i += 1

        else:
            raise ValueError("Invalid character")

    termination = TerminationState()
    self.states[-1].next_states.append(termination)
```

- Визначення документації класу `RegexFSM`, яка описує підтримувані функції та наводить приклади використання.
- Конструктор класу, який ініціалізує початковий стан `start_state` та список всіх станів `states`.
- Цикл, який обробляє кожен символ регулярного виразу і будує відповідний скінченний автомат.
- Обробка звичайних ASCII-символів. Створюється новий стан `AsciiState` для символу, додається до списку наступних станів попереднього стану та до загального списку станів.
- Обробка символу `"."`. Створюється новий стан `DotState`, який додається аналогічно.
- Обробка оператора `"*"`. Створюється новий стан `StarState` для попереднього стану. Важливо відзначити наступні кроки:
  - Додавання наступних станів базового стану до `next_states` стану зірки Кліні.
  - Додавання наступних станів стану зірки Кліні до `next_states` базового стану.
  - Додавання стану зірки Кліні до `next_states` передостаннього стану.
- Обробка оператора `"+"`. Створюється новий стан `PlusState` для попереднього стану. На відміну від `StarState`, наступні стани базового стану не додаються до `next_states` базового стану. Це є ключовою відмінністю між операторами `"*"` і `"+"`.
- Обробка невідомих символів – викидається виняток `ValueError`.
- Додавання кінцевого стану `TerminationState` до списку наступних станів останнього стану.

Лістинг 9: RegexFSM - метод перевірки рядка

```
def check_string(self, input_string: str) -> bool:

visited = set()

def dfs(state: State, idx: int) -> bool:
    if (id(state), idx) in visited:
        return False
    visited.add((id(state), idx))

    if idx == len(input_string):
        return any(isinstance(s, TerminationState) for s in state.next

    for next_state in state.next_states:
        if next_state.check_self(input_string[idx]):
            if dfs(next_state, idx + 1):
                return True
        elif dfs(next_state, idx):
            return True
    return False

return dfs(self.start_state, 0)
```

- Метод `check_string`, який перевіряє, чи відповідає вхідний рядок скомпільованому регулярному виразу.
- Ініціалізація множини `visited` для відстеження вже відвіданих пар (стан, індекс), щоб уникнути нескінченної рекурсії.
- Визначення вкладеної функції `dfs` (depth-first search), яка реалізує обхід у глибину для пошуку шляху в графі автомату.
- Перевірка, чи була вже відвідана пара (стан, індекс). Якщо так — повертається `False`, щоб уникнути циклів.
- Якщо досягнуто кінця вхідного рядка, перевіряється, чи є серед наступних станів поточного стану термінальний стан. Якщо так — це означає, що рядок відповідає регулярному виразу.
- Для кожного наступного стану:
  - Якщо стан приймає поточний символ, рекурсивно викликається `dfs` для цього стану з наступним індексом.

- Якщо стан не приймає поточний символ, рекурсивно викликається `dfs` для цього стану з тим самим індексом (це дозволяє обробляти епсилон-переходи).
- Якщо будь-який з цих викликів повертає `True`, повертається `True`.

- Запуск `dfs` з початкового стану та індексу 0.

Лістинг 10: Блок запуску

```
if name == "main":
import doctest
doctest.testmod(verbose=True)
regex_pattern = "a*4.+hi"
regex_compiled = RegexFSM(regex_pattern)

print(regex_compiled.check_string("aaaaaa4uhi"))   #True
print(regex_compiled.check_string("4uhi"))         #True
print(regex_compiled.check_string("meow"))         #False
```

- Блок коду, який виконується, якщо файл запускається безпосередньо (а не імпортується як модуль).
- Імпорт модуля `doctest` та запуск всіх тестів, визначених у документації класів.
- Приклад використання класу `RegexFSM` з конкретним регулярним виразом та перевірка декількох рядків.

4 Приклади роботи

Розглянемо регулярний вираз `a4.+hi` та проаналізуємо, як він обробляється:

1. `a*`: Символ "a"нуль чи більше разів
2. `4`: Символ "4"точно один раз
3. `.+`: Будь-який символ один чи більше разів
4. `hi`: Послідовність символів "h"та "i"

При компіляції цього регулярного виразу створюється наступна послідовність станів:

1. `StartState` - початковий стан
2. `AsciiState("a")` - стан для символу "a"
3. `StarState(AsciiState("a"))` - стан для оператора "* що діє на символ "a"
4. `AsciiState("4")` - стан для символу "4"
5. `DotState` - стан для оператора "."
6. `PlusState(DotState)` - стан для оператора "+ що діє на оператор "."
7. `AsciiState("h")` - стан для символу "h"
8. `AsciiState("i")` - стан для символу "i"
9. `TerminationState` - кінцевий стан

Перевірка рядка `aaaaaa4uhi`:

1. Символи "aaaaaa"відповідають `a*` (нуль або більше повторень "a")
2. Символ "4"відповідає `4` (точно один символ "4")
3. Символ "u"відповідає `.+` (один або більше будь-яких символів)
4. Символи "hi"відповідають `hi` (послідовність символів "h"та "i")

Тому рядок `aaaaaa4uhi` відповідає регулярному виразу `a*4.+hi`. Перевірка рядка `4uhi`:

1. Відсутність символів "a"також відповідає `a*` (нуль повторень "a")
2. Символ "4"відповідає `4`
3. Символ "u"відповідає `.+`
4. Символи "hi"відповідають `hi`

Тому рядок `4uhi` також відповідає регулярному виразу `a*4.+hi`. Перевірка рядка `meow`:

1. Символ "m"не відповідає `a*` (очікується "a"або відсутність символів)
2. Символ "e"не відповідає `4` (очікується "4")

Тому рядок `meow` не відповідає регулярному виразу `a*4.+hi`.

5 Висновки

У даній лабораторній роботі була реалізована система обробки спрощених регулярних виразів за допомогою недетермінованого скінченного автомату (NFA). Ця реалізація підтримує основні оператори регулярних виразів, такі як "."(будь-який символ), "*"(нуль або більше повторень) та "+"(один або більше повторень). Основні компоненти реалізації:

- Абстрактний базовий клас `State` для представлення станів автомату
- Спеціалізовані підкласи для різних типів станів (`StartState`, `TerminationState`, `DotState`, `AsciiState`, `StarState`, `PlusState`)
- Клас `RegexFSM` для компіляції регулярних виразів та перевірки рядків

Алгоритм перевірки відповідності рядка регулярному виразу базується на обході в глибину (DFS) графа автомату з відстеженням вже відвіданих станів для уникнення циклів. Дана реалізація демонструє важливі концепції теорії автоматів та їх застосування для вирішення практичних задач обробки тексту. Розроблений скінченний автомат має наступні переваги:

- Модульна архітектура, що дозволяє легко розширювати функціональність
- Ефективний алгоритм перевірки рядків з використанням обходу в глибину
- Підтримка основних операторів регулярних виразів
- Чистий та добре документований код

Можливі напрямки подальшого розвитку:

- Додавання підтримки групування виразів за допомогою дужок
- Реалізація операторів альтернативи (|)
- Додавання квантифікаторів (?, n, n,m)
- Оптимізація автомата шляхом детермінізації та мінімізації

Ця лабораторна робота демонструє практичне застосування теоретичних знань з теорії формальних мов та автоматів для створення корисного інструменту для обробки текстових даних.