Звіт щодо виконання лабораторної роботи по реалізації кінцевого автомата для обробки регулярних виразів

Зміст

1 Вступ

Представлена реалізація впроваджує кінцевий автомат (Finite State Machine, FSM) для обробки регулярних виразів. Це класичний підхід до розпізнавання шаблонів у тексті, заснований на теорії формальних мов. Реалізація включає базовий абстрактний клас стану, спеціалізовані стани для різних конструкцій регулярних виразів та головний клас для компіляції та обробки шаблонів.

2 Загальна архітектура

Реалізація використовує об'єктно-орієнтовану архітектуру з наступними компонентами:

1. Абстрактний базовий клас `State` визначає інтерфейс для усіх типів станів
2. Спеціалізовані класи станів для обробки різних елементів регулярних виразів
3. Головний клас `RegexFSM`, який керує компіляцією шаблону в автомат та виконанням пошуку

Такий підхід обрано для забезпечення розширюваності та зручності додавання нових типів станів або операторів. Ця архітектура є класичною для реалізації скінченних автоматів і дозволяє чітко відокремити відповідальність між обробкою окремих символів та керуванням автоматом у цілому.

3 Базовий клас `State`

3.1 Опис та призначення

```python
class State(ABC):
    @abstractmethod
    def __init__(self) -> None:
        pass

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

    def check_next(self, next_char: str) -> State | Exception:
        for state in self.next_states:
            if state.check_self(next_char):
                return state
        raise NotImplementedError("rejected string")
```

Цей абстрактний базовий клас визначає два основних методи:

1. `check_self(char)` – абстрактний метод, який перевіряє, чи обробляє даний стан вказаний символ.
2. `check_next(next_char)` – метод, який передає символ наступним станам і повертає стан, що може його обробити.

3.2 Обґрунтування реалізації

Такий підхід використовує принцип єдиної відповідальності: кожен стан відповідає лише за власну логіку обробки, а методи `check_self` і `check_next` забезпечують послідовну перевірку переходів. Це дозволяє реалізувати недетермінований скінченний автомат (NFA), що є природним для регулярних виразів.

Зберігання посилань на наступні стани в атрибуті `next_states` спрощує роботу з переходами між станами без необхідності централізованої таблиці переходів.

4 Типи станів

4.1 StartState

```
class StartState(State):
    next_states: List[State] = []
    def __init__(self):
        self.next_states = []
    def check_self(self, char):
        return False
```

Стартовий стан не обробляє жодних символів (`check_self` завжди повертає `False`), він слугує лише як точка входу в автомат. Ця реалізація підкреслює, що старт регулярного виразу не споживає жодних символів.

4.2 TerminationState

```
class TerminationState(State):
    next_states: List[State] = []

    def __init__(self):
        self.next_states = []

    def check_self(self, char):
        return False
```

Термінальний стан також не обробляє символи, він лише позначає успішне завершення розбору. Цей підхід дозволяє чітко визначити момент, коли регулярний вираз було повністю розпізнано.

4.3 DotState

```
class DotState(State):
    next_states: List[State] = []

    def __init__(self):
        self.next_states = []

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

Стан `.` приймає будь-який символ, тому `check_self` завжди повертає `True`. Це безпосередньо відображає семантику оператора `.` в регулярних виразах.

4.4 AsciiState

```
class AsciiState(State):
    next_states: List[State] = []
    curr_sym = ""

    def __init__(self, symbol: str) -> None:
        self.next_states = []
        self.curr_sym = symbol

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

Стан обробляє конкретний символ із ASCII-таблиці. Метод `check_self` порівнює вхідний символ з очікуваним символом `curr_sym`. Даний підхід дозволяє створити окремі стани для кожного літерального символу в регулярному виразі.

4.5 StarState

```
class StarState(State):
    next_states: List[State] = []

    def __init__(self, checking_state: State):
        self.next_states = []
        self.checking_state = checking_state

    def check_self(self, char):
        if self.checking_state.check_self(char):
            return True
        for state in self.next_states:
            if state.check_self(char):
                return True
        return False
```

Стан `*` реалізує квантифікатор “нуль або більше” повторень. Він містить посилання на стан, який повинен повторюватися (`checking_state`). Метод `check_self` перевіряє, чи підходить символ для базового стану або для будь-якого з наступних станів.

Така реалізація відображає недетермінованість автомата для операції `*`, де можуть бути активні одночасно кілька станів.

4.6 PlusState

```
class PlusState(State):
    next_states: List[State] = []

    def __init__(self, checking_state: State):
```

```
        self.next_states = []
        self.checking_state = checking_state
        self.matched_at_least_one = False

    def check_self(self, char):
        if self.checking_state.check_self(char):
            self.matched_at_least_one = True
            return True
        if self.matched_at_least_one:
            for state in self.next_states:
                if state.check_self(char):
                    return True
        return False
```

Стан + реалізує квантифікатор "один або більше" повторень. Він відрізняється від `StarState` наявністю прапорця `matched_at_least_one`, що гарантує принаймні одне співпадіння перед переходом до наступних станів. Це безпосередньо відображає семантичну різницю між операторами * і +.

4.7 CharacterClass

```
class CharacterClass(State):
    next_states: List[State] = []

    def __init__(self, class_definition: str):
        self.next_states = []
        self.chars = set()
        self._parse_class(class_definition)

    def _parse_class(self, definition: str):
        i = 0
        while i < len(definition):
            if i + 2 < len(definition) and definition[i+1] == '-':
                start_char = definition[i]
                end_char = definition[i+2]
                for char_code in range(ord(start_char), ord(
    end_char) + 1):
                    self.chars.add(chr(char_code))
                i += 3
            else:
                self.chars.add(definition[i])
                i += 1

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

Стан для класу символів `[]` обробляє діапазони символів, як-от `[a-z0-9]`. Метод `_parse_class` розбирає визначення класу, обробляючи як окремі символи, так і діапазони, та зберігає їх у множині `chars`. Метод `check_self` просто перевіряє наявність символу в множині.

Використання множини (`set`) забезпечує швидкий пошук символу O(1), що важливо для ефективної роботи з класами символів, які можуть містити багато еле-

ментів.

5 Клас RegexFSM

Головний клас для компіляції регулярних виразів і виконання пошуку:

```
class RegexFSM:
    def __init__(self, regex_expr: str) -> None:
        self.pattern = regex_expr
        self.curr_state = StartState()
        self.start_state = self.curr_state
        self.final_state = TerminationState()
        self.states_map = {}
        self.states_map[self.start_state] = {"transitions": {}, "
   epsilon": set()}
        self.states_map[self.final_state] = {"transitions": {}, "
   epsilon": set()}
        self._compile(regex_expr)
```

Конструктор ініціалізує:

1. Початковий стан (`StartState`)
2. Кінцевий стан (`TerminationState`)
3. Словник-мапу станів і переходів (`states_map`)
4. Компілює регулярний вираз у граф станів через виклик `_compile`

5.1 Метод _compile

```
def _compile(self, pattern: str):
    if not pattern:
        self._add_epsilon_transition(self.start_state, self.
   final_state)
        return

    current_state = self.start_state
    i = 0
    while i < len(pattern):
        char = pattern[i]
        # (                                                    )
    self._add_epsilon_transition(current_state, self.final_state)
```

Цей метод є ключовим для компіляції регулярного виразу в граф станів. Він обробляє патерн посимвольно, створюючи відповідні стани та переходи. Метод використовує алгоритм прямого аналізу (лексичний аналіз) регулярного виразу, що забезпечує простоту реалізації та підтримки.

Основні принципи реалізації:

1. Обробка спеціальних конструкцій (класи символів `[]`, квантифікатори `*`, `+`)
2. Створення станів для кожного символу та додавання переходів між ними
3. З'єднання останнього стану з кінцевим через епсилон-перехід

5.2 Метод _handle_repetition

```
def _handle_repetition(self, current_state, base_state, repeat_type
    ):
    self._add_state(base_state)
    if repeat_type == '*':
        loop_state = StarState(base_state)
        self._add_state(loop_state)
        self._add_epsilon_transition(current_state, loop_state)
        self._add_transition(current_state, base_state, '')
        self._add_epsilon_transition(base_state, loop_state)
        self._add_epsilon_transition(loop_state, base_state)
        return loop_state
    else:
        loop_state = PlusState(base_state)
        self._add_state(loop_state)
        self._add_transition(current_state, base_state, '')
        self._add_epsilon_transition(base_state, loop_state)
        self._add_epsilon_transition(loop_state, base_state)
        return loop_state
```

Цей метод обробляє квантифікатори * і +. Структура переходів відрізняється для операторів:

- Для * додається епсилон-перехід в обхід базового стану (відображає можливість нуля повторень)
- Для + прямий перехід до базового стану (потрібне щонайменше одне повторення)

В обох випадках додаються епсилон-переходи для циклу між базовим станом і квантифікатором.

5.3 Методи керування станами та переходами

```
def _add_state(self, state):
    if state not in self.states_map:
        self.states_map[state] = {"transitions": {}, "epsilon": set
    ()}

def _add_transition(self, from_state, to_state, char):
    if char not in self.states_map[from_state]["transitions"]:
        self.states_map[from_state]["transitions"][char] = set()
    self.states_map[from_state]["transitions"][char].add(to_state)
    from_state.next_states.append(to_state)

def _add_epsilon_transition(self, from_state, to_state):
    self.states_map[from_state]["epsilon"].add(to_state)
    from_state.next_states.append(to_state)
```

Ці допоміжні методи спрощують керування станами та переходами:

1. _add_state додає новий стан до загальної мапи станів

2. `_add_transition` додає символьний перехід між станами

3. `_add_epsilon_transition` додає епсилон-перехід (перехід без споживання символу)

Використання множин (`set`) для переходів важливе, оскільки воно забезпечує відсутність дублювання станів і покращує пошук O(1).

5.4 Метод _epsilon_closure

```
def _epsilon_closure(self, states):
    """Find all states reachable through epsilon transitions"""
    closure = set(states)
    stack = list(states)

    while stack:
        state = stack.pop()
        for next_state in self.states_map[state]["epsilon"]:
            if next_state not in closure:
                closure.add(next_state)
                stack.append(next_state)
    return closure
```

Цей метод реалізує пошук епсилон-замикання для множини станів. Він використовує алгоритм пошуку в глибину (DFS) для знаходження всіх станів, досяжних через епсилон-переходи. Цей метод є критичним для реалізації недетермінованого скінченного автомата (NFA), оскільки він дозволяє одночасно перебувати в декількох станах.

5.5 Метод check_string

```
def check_string(self, text: str) -> bool:
    """
    Check if the input string contains the regex pattern.
    """
    if self.is_full_match(text):
        return True
    for start_pos in range(len(text)):
        current_states = self._epsilon_closure({self.start_state})
        for i in range(start_pos, len(text)):
            char = text[i]
            next_states = set()
            for state in current_states:
                for c, destinations in self.states_map[state]["
    transitions"].items():
                    state_obj = next(iter([s for s in [state] if
    isinstance(s,
                    (AsciiState, DotState, CharacterClass,
    StarState, PlusState))]), None)
                    if c == '.' or (state_obj and isinstance(
    state_obj, DotState)):
```

```
                        next_states.update(destinations)
                    elif (c == char or
                          (state_obj and isinstance(state_obj,
    AsciiState)
                    and state_obj.curr_sym == char) or
                          (state_obj and isinstance(state_obj,
    CharacterClass)
                    and char in state_obj.chars)):
                        next_states.update(destinations)
                    elif state_obj and isinstance(state_obj,
                    (StarState, PlusState)) and state_obj.
    check_self(char):
                        next_states.update(destinations)
            current_states = self._epsilon_closure(next_states)
            if not current_states:
                break
            if self.final_state in current_states:
                return True
     return False
```

Цей метод перевіряє, чи містить вхідний рядок підрядок, що відповідає шаблону регулярного виразу. Він використовує алгоритм моделювання роботи NFA:

1. Спочатку перевіряє, чи весь рядок відповідає шаблону
2. Для кожної можливої стартової позиції в тексті:
   - Ініціалізує множину поточних станів початковим станом з епсилон-замиканням
   - Для кожного символу з цієї позиції:
     - Знаходить усі переходи, що відповідають поточному символу
     - Додає цільові стани до множини наступних станів
     - Виконує епсилон-замикання для нових станів
     - Перевіряє, чи досягнуто кінцевого стану

Такий підхід є класичним алгоритмом симуляції NFA і забезпечує коректну роботу з недетермінованими переходами.

5.6 Метод `is_full_match`

```
def is_full_match(self, text: str) -> bool:
    """
    Check if the entire input string matches the regex pattern.
    """
    current_states = self._epsilon_closure({self.start_state})
    for char in text:
        next_states = set()
        for state in current_states:
            for c, destinations in self.states_map[state]["
   transitions"].items():
                if isinstance(state, DotState):
```

```
                    next_states.update(destinations)
                elif isinstance(state, AsciiState) and state.
curr_sym == char:
                    next_states.update(destinations)
                elif isinstance(state, CharacterClass) and char in
state.chars:
                    next_states.update(destinations)
                elif c == char:
                    next_states.update(destinations)
                elif c == '.':
                    next_states.update(destinations)
            for next_state in state.next_states:
                if next_state.check_self(char):
                    next_states.add(next_state)
        current_states = self._epsilon_closure(next_states)
        if not current_states:
            return False
    return self.final_state in current_states or any(state.is_final
    if
    hasattr(state, 'is_final') else False for state in
current_states)
```

Цей метод перевіряє, чи повністю відповідає вхідний рядок шаблону регулярного виразу. Він також використовує симуляцію NFA, але з однією стартовою позицією і вимогою, щоб фінальний стан був досягнутий в кінці рядка.

Алгоритм схожий на `check_string`, але не перевіряє всі можливі стартові позиції, а лише проходить по тексту з початку до кінця.

6 Алгоритми та методи

6.1 Алгоритм побудови НСА (Недетермінованого Скінченного Автомата)

Для побудови автомата використовується алгоритм Томпсона (Thompson's construction), що переводить регулярний вираз в еквівалентний NFA. Цей алгоритм обрано через його простоту реалізації та зрозумілість. Основні кроки:

1. Розбиття регулярного виразу на базові компоненти (символи, класи символів, оператори)
2. Побудова станів для кожного компоненту
3. З'єднання станів відповідними переходами
4. Обробка квантифікаторів через епсилон-переходи

Алгоритм має часову складність O(n) для побудови автомата, де n – довжина регулярного виразу.

6.2 Алгоритм симуляції NFA

Для перевірки відповідності рядка регулярному виразу використовується алгоритм симуляції NFA, який має такі переваги:

1. Прямо відображає недетермінований характер автомата
2. Простіше реалізується, ніж перетворення NFA в DFA
3. Може бути більш ефективним для коротких рядків або простих шаблонів

Однак цей метод має гіршу асимптотичну складність O(n*m), де n – довжина рядка, m – кількість станів автомата.

6.3 Використання епсилон-замикань

Для епсилон-замикань використовується алгоритм пошуку в глибину (DFS), що забезпечує коректну обробку епсилон-переходів. Цей підхід обрано через його простоту і ефективність (складність O(V+E), де V – кількість станів, E – кількість переходів).

7 Аналіз ефективності

7.1 Просторова складність

- **Побудова автомата**: O(n), де n – довжина регулярного виразу
- **Зберігання автомата**: O(m), де m – кількість станів (пропорційна n)

7.2 Часова складність

- **Компіляція шаблону**: O(n)
- **Перевірка рядка**:
  - Найгірший випадок: O(n*m), де n – довжина рядка, m – кількість станів
  - Це відбувається, коли потрібно перевірити багато шляхів через автомат

8 Висновки

Реалізація регулярних виразів через кінцевий автомат є класичним і теоретично обґрунтованим підходом. Обрана архітектура забезпечує:

1. **Розширюваність**: Легко додавати нові типи станів або оператори
2. **Зрозумілість**: Кожен компонент має чітку відповідальність
3. **Коректність**: Реалізація безпосередньо відповідає теоретичному визначенню регулярних виразів

Використання NFA (недетермінованого скінченного автомата) замість DFA зроблено для спрощення реалізації та покращення підтримуваності коду, хоча це впливає на ефективність роботи з великими текстами.

Загалом, дана реалізація представляє збалансований підхід між ефективністю, зрозумілістю, і можливістю подальшого розширення функціоналу.