Леник Нікіта

Скінченний автомат для регулярних виразів

1 Скінченний автомат

1.1 Принцип роботи автомата

Скінченний автомат (FSM - Finite State Machine) є основою для розбору та перевірки відповідності рядків регулярним виразам. Ключовий принцип полягає в тому, що автомат приймає стрічку тільки тоді, коли після обробки всіх символів він переходить у фінальний стан. Якщо ж по завершенню обробки автомат не перебуває у фінальному стані, стрічка вважається такою, що не відповідає заданому регулярному виразу.

1.2 Реалізація автомата в проєкті

Автомат реалізовано як набір взаємопов'язаних станів, кожен з яких відповідає певному компоненту регулярного виразу. Коли автомат обробляє вхідну стрічку, він переходить між станами в залежності від символів, які зчитує.

Основний алгоритм роботи автомата:

1. Автомат починається зі стартового стану.
2. Для кожного символу у вхідній стрічці:
   - Визначаються можливі переходи з поточного стану, що приймають поточний символ.
   - Якщо такі переходи існують, автомат переміщається до нових станів.
   - Якщо жодних переходів немає, обробка завершується невдачею.
3. Після обробки всієї стрічки перевіряється, чи є хоча б один із поточних станів фінальним.

1.3 Використані рішення

У проєкті було застосовано такі архітектурні рішення:

- **Ієрархія класів станів** - дозволяє ефективно моделювати різні типи елементів регулярних виразів, такі як окремі символи, класи символів, квантифікатори (*, +, [], .).
- **Метакласи** - використані для динамічного розширення функціональності класів станів, зокрема для реалізації оператора зірочки замикання Кліні.
- **Композиція станів** - складні регулярні вирази будуються шляхом композиції простіших станів, утворюючи направлений граф. Для прикладу

$$a+ = aa*$$

2 Технічна реалізація автомата

2.1 Використання метакласу

Метаклас `StateMeta` використовується для розширення функціональності базового класу `State` додаванням методу `__mul__`, що дозволяє використовувати оператор * для створення станів типу `StarState`:

```
class StateMeta(ABCMeta):
    def __new__(mcs: type, name: str, bases: tuple, namespace: dict):
        def __mul__(self, _other: "State") -> "State":
            return StarState(self)

        namespace["__mul__"] = __mul__
        return super().__new__(mcs, name, bases, namespace)
```

Це дозволяє використовувати нотацію `state * None` для створення стану, що представляє конструкцію "нуль або більше повторень"(оператор * в регулярних виразах).

2.2 Класи станів

- **State** - абстрактний базовий клас для всіх станів із методами `check_self` та `check_next`.
- **StartState** - початковий стан автомата, з якого починається обробка стрічки.

  ```
  class StartState(State):
      def check_self(self, _: str) -> bool:
          return False
  ```

- **TerminationState** - кінцевий стан, що позначає успішне завершення обробки.

  ```
  class TerminationState(State):
      def __init__(self) -> None:
          super().__init__()
          self.is_final = True

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

- **AsciiState** - стан, який приймає конкретний символ.

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

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

- **DotState** - стан, що приймає будь-який символ (відповідає "."у регулярних виразах).

```
class DotState(State):
    def check_self(self, _: str) -> bool:
        return True
```

- **ClassState** - стан для класів символів (відповідає [...] у регулярних виразах).

```
class ClassState(State):
    def __init__(self, chars: set[str], ignore: bool = False) -> None:
        super().__init__()
        self.chars = chars
        self.ignore = ignore

    def check_self(self, char: str) -> bool:
        return char not in self.chars if self.ignore else char in self.cha
```

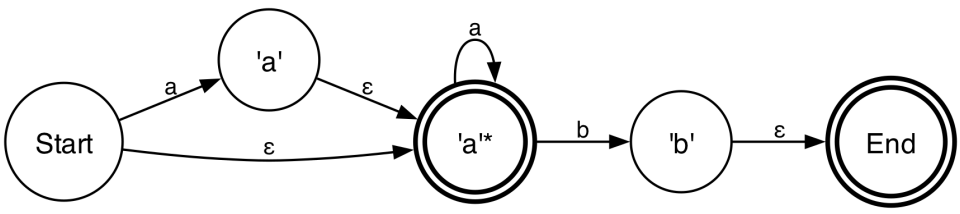
- **StarState** - стан для оператора замикання Кліні (*), що приймає нуль або більше повторень базового стану.

```
class StarState(State):
    def __init__(self, base: State) -> None:
        super().__init__()
        self.base = base
        self.is_final = True

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

    def check_next(self, next_char: str) -> "State":
        if self.check_self(next_char):
            return self
        return super().check_next(next_char)
```

2.3 Побудова та використання автомата

Клас `RegexFSM` будує автомат на основі рядка регулярного виразу. Він аналізує вираз символ за символом і створює відповідні стани та переходи. Метод `check_string` використовується для перевірки, чи відповідає вхідний рядок заданому регулярному виразу, відстежуючи всі можливі стани, в яких може перебувати автомат після обробки кожного символу.

3 Аналіз автомата на зображенні



Скінченний автомат для регулярного виразу "a*b"

На зображенні представлено автомат для регулярного виразу `a*b`, який складається з наступних станів:

- **Start** - початковий стан
- **'a'** - стан, що приймає символ 'a'
- **'a'*** - стан, що відповідає конструкції Кліні для символа 'a' (фінальний стан)
- **'b'** - стан, що приймає символ 'b'
- **End** - кінцевий стан (фінальний)

Переходи між станами:

- **Start** $\xrightarrow{a}$ **'a'** - при зчитуванні символа 'a' зі стартового стану
- **Start** $\xrightarrow{\varepsilon}$ **'a'*** - епсілон-перехід (без зчитування символа)
- **'a'** $\xrightarrow{\varepsilon}$ **'a'*** - епсілон-перехід після зчитування 'a'
- **'a'*** $\xrightarrow{a}$ **'a'*** - петля, що дозволяє автомату зчитувати багато символів 'a'
- **'a'*** $\xrightarrow{b}$ **'b'** - перехід при зчитуванні 'b' після будь-якої кількості 'a'
- **'b'** $\xrightarrow{\varepsilon}$ **End** - епсілон-перехід у кінцевий стан

3.1 Приклади роботи автомата

3.1.1 Обробка стрічки "b"

1. Початковий стан: **Start**
2. Через епсілон-перехід автомат одночасно перебуває у стані **'a'***
3. Обробка символу 'b':
   - У стані **'a'*** зчитується 'b' і автомат переходить до стану **'b'**
4. По завершенню обробки через епсілон-перехід автомат переходить до стану **End**
5. Оскільки **End** є фінальним станом, стрічка "b"приймається

3.1.2 Обробка стрічки "aab"

1. Початковий стан: **Start**
2. Обробка першого символу 'a':
   - Зі стану **Start** автомат переходить до стану **'a'**
   - Через епсілон-перехід також переходить до стану **'a'***
3. Обробка другого символу 'a':

- У стані **'a'*** є петля для символу 'a', тому автомат залишається в цьому ж стані

4. Обробка символу 'b':
   - У стані **'a'*** зчитується 'b' і автомат переходить до стану **'b'**
5. По завершенню обробки через епсілон-перехід автомат переходить до стану **End**
6. Оскільки **End** є фінальним станом, стрічка "aab"приймається

3.2 Пояснення епсілон-переходів (λ-переходів)

На діаграмі автомата для регулярного виразу "a*b"можна побачити переходи, позначені як ε (епсілон-переходи, λ-переходи). Ці переходи мають особливе значення: вони дозволяють автомату переходити з одного стану в інший без зчитування символу з вхідної стрічки.

Причини використання епсілон-переходів:

- **Підтримка операцій Кліні (*)** - епсілон-переходи дозволяють автомату "перескакувати"частину шаблону, що відповідає за повторення (нуль або більше разів).
- **Спрощення структури автомата** - вони дозволяють з'єднувати різні частини автомата без додаткових перевірок символів.

Реалізація епсілон-переходів у коді:

```
current |= {
    next_state
    for state in current
    for next_state in state.next_states
    if isinstance(next_state, StarState) and next_state.is_final
}
```

Цей код додає можливі стани, досяжні через епсілон-переходи від поточного стану до станів типу StarState. Подібним чином, наприкінці обробки стрічки автомат перевіряє фінальні стани:

```
current |= {
    next_state
    for state in current
    for next_state in state.next_states
    if next_state.is_final
}
return any(st.is_final for st in current)
```

Ця конструкція дозволяє врахувати фінальні стани, досяжні через епсілон-переходи, що є ключовим елементом недетермінованого скінченного автомата для обробки регулярних виразів.