Regex та скінченні автомати

Маргарита Падучак

Анотація—Реалізовано простий інтерпретатор регулярних виразів (regex). Ми будуємо автомат (NFA), який перевіряє, чи підходить вхідний рядок під заданий шаблон. Імплементовано розпізнавання літер англійської абетки (верхнього та нижнього регістру), числа, знаки * та +.

I. Пояснення (як для мами)

Ми будуємо уявну машину, яка читає букви в слові по черзі і перевіряє, чи це слово підходить під даний нам шаблон (regex).

Як ця машина працює?

Приклад шаблону: a*4.+hi
Уяви собі коробочки (стани), з'єднані стрілками.
Кожна стрілка каже: "якщо побачиш ось такий символ — іди далі".

- Є коробочка, яка каже: "можеш бачити скільки завгодно літер 'a' — хоч 0, хоч 100".
- Потім стрілка веде до коробочки, яка чекає цифру 4.
- Потім ще одна каже: "тепер будь-який символ".
- Потім дві коробочки, які чекають букви 'h' і 'i' підряд.

Коли ти вводиш слово, програма ніби запускає кульку в цю машину, і та стрибає з коробочки в коробочку, залежно від символів, які ти набрав.

Якщо кулька в кінці доходить до фінальної коробочки, це означає, що слово — підходить під шаблон.

II. Структура коду

Базовий клас State

State — це базовий клас, що є основою для всіх станів автомата.

Властивості:

- related_key_state — переходи за конкретними символами, наприклад, перехід з символу a до іншого стану: a $\rightarrow$ стан.
- e_transitions — ε-переходи (перехід без символу, автомат сам переходить у наступний стан).

Методи:

- check_self(char) — абстрактний метод. Має бути реалізований у нащадках. Перевіряє, чи символ підходить для цього стану.

Похідні стани:

- StartState, TerminationState — спеціальні стани для початку та завершення роботи автомата.
- DotState — завжди повертає True; підходить під будь-який символ ("."у регулярних виразах).
- AsciiState — представляє стан, який очікує конкретний ASCII-символ.
- StarState, PlusState — службові допоміжні вузли, які створюють структуру для операторів "*"(нуль або більше) та "+"(один або більше).
- TempState — допоміжний стан, який використовується для побудови переходів між символами під час побудови автомата.

Побудова автомата (RegexFSM)

Клас RegexFSM отримує регулярний вираз у вигляді рядка (тільки символи a-z, A-Z, 0-9, ., *, +) і будує недетермінований скінченний автомат (NFA).

- Він покроково обробляє кожен символ шаблону:
  - Якщо це символ або ".", створюється відповідний фрагмент автомата (пара станів з переходом).
  - Якщо після символу йде * або +, то додаються петлі:
    - "*"— відповідає за "нуль або більше" повторень.
    - "+"— відповідає за "один або більше" повторень.
- Потім усі частини поєднуються в єдину конструкцію через ε-переходи.

Перевірка рядків (check_string)

Метод check_string імітує "проходження рядка" через автомат:

- Починає з початкового стану та всіх станів, які можна досягти через ε-переходи.
- Для кожного символу в рядку:
  - Шукає всі можливі переходи з поточних станів.
  - Переходить у нові стани та знову виконує ε-переходи.
- Якщо після обробки всіх символів хоча б один із досягнутих станів — це фінальний (TerminationState), то рядок задовольняє шаблон.

III. Пояснення коду

Клас State

- add_related_key_state(symbol, state) — додає перехід із поточного стану до вказаного стану за певним символом (наприклад, 'a' → S_1).
- add_e_transitions(state) — додає ε-перехід (тобто перехід без символу) до іншого стану. Автомат може перейти в цей стан "автоматично".

Типи станів у автоматі

- Стани, які перевіряють символи.
  - AsciiState — перевіряє, чи символ збігається з конкретним (a, b, 4 тощо).
  - DotState — приймає будь-який символ, тому завжди повертає True.
- Службові стани.
  - StartState
  - TerminationState
  - StarState
  - PlusState
  - TempState

  Ці стани — не для перевірки символів, а для структури автомата:
  - Вони з'єднують частини автомата.
  - Дають можливість робити переходи без символів (ε-переходи).
  - Дають можливість повторювати фрагмент (як у * і +).

Розбір конструктора класу RegexFSM

Метод __init__ будує недетермінований автомат з ε-переходами (ε-NFA) на основі регулярного виразу.

```
self.start_state = StartState()
self.termination_state = TerminationState()
```

Створюються службові стани: початковий і кінцевий.

```
curr_start, curr_end = None, None
i = 0
```

Ініціалізація змінних для з'єднання фрагментів автомата.

```
while i < len(messege):
    el = messege[i]
```

Основний цикл: проходження по символах регулярного виразу.

```
if el == "." or el.isascii():
    fragment_start, fragment_end = self.make_start_end(el)
```

Створення фрагменту автомата для ASCII-символу або крапки.

```
if i + 1 < len(messege) and messege[i+1] in ("*", "+"):
    i += 1
```

Перевірка, чи після символу стоїть квантифікатор * або +.

Для * (нуль або більше повторень):

```
temp_start, temp_end = StarState(), StarState()
temp_start.add_e_transitions(fragment_start)
temp_start.add_e_transitions(temp_end)
fragment_end.add_e_transitions(fragment_start)
fragment_end.add_e_transitions(temp_end)
fragment_start, fragment_end = temp_start, temp_end
```

Для + (одне або більше повторень):

```
temp_start, temp_end = PlusState(), PlusState()
temp_start.add_e_transitions(fragment_start)
fragment_end.add_e_transitions(fragment_start)
fragment_end.add_e_transitions(temp_end)
fragment_start, fragment_end = temp_start, temp_end
```

Додавання фрагментів до основного автомата:

```
if curr_end is None:
    curr_start, curr_end = fragment_start, fragment_end
else:
    curr_end.add_e_transitions(fragment_start)
    curr_end = fragment_end
```

Завершення побудови автомата:

```
if curr_end is None:
    self.start_state.add_e_transitions(self.termination_state)
else:
    self.start_state.add_e_transitions(curr_start)
    curr_end.add_e_transitions(self.termination_state)
```

Функції класу RegexFSM

Метод make_start_end:

Створює базовий фрагмент автомата, який розпізнає один символ.

```
    start_ = TempState()
    end_ = TempState()
    start_.add_related_key_state(key, end_)
    return start_, end_
```

- Створює два тимчасові стани.
- Додає між ними перехід по символу.
- Повертає пару цих станів як фрагмент автомата.

Метод e_transitions_closure:

Знаходить усі стани, які можна досягти з початкової множини станів за ε-переходами.

```
    stack_ = list(states)
    res = set(states)
    while stack_:
        state = stack_.pop()
        for way_e in state.e_transitions:
            if way_e not in res:
                res.add(way_e)
                stack_.append(way_e)
    return res
```

- Використовує стек для обходу всіх ε-переходів.
- Повертає множину всіх досяжних станів.

Література

[1] Hasti, R. Scanning: Regular Expressions and Lexical Analysis. University of Wisconsin-Madison. URL: https://pages.cs.wisc.edu/ hasti-/cs536/readings/scanning.html, доступ 2023.

[2] EECS York University. Finite State Automata (FSA) Overview. URL: https://wiki.eecs.yorku.ca/course_archive/2014-15/W/6339/_media/fsa.pdf, доступ 2023.