Звіт про реалізацію скінченного автомата для перевірки регулярних виразів

Кальмук Ярополк

1. Мета роботи

Метою є створення скінченного автомата, який будується на основі регулярного виразу та перевіряє, чи заданий рядок відповідає цьому виразу. Автомат підтримує основні конструкції регулярних виразів: символи ASCII, крапку (.), зірочку (*) та плюс (+).

2. Опис реалізації

Основною ідеєю є створення абстрактного класу `State`, від якого наслідуються конкретні типи станів. Кожен стан реалізує логіку перевірки окремого символу. Спочатку при заданні regex'у створюються стани та можливі переходи між ними. Коли користувач вводить рядок, то програма послідовно рухається по станах, у які може зайти. Щоб вивелось True, автомат має дійти до термінального стану.

2.1 Класи станів

- **State** — абстрактний базовий клас. Визначає методи `check_self()` та `check_next()`. `check_self()` перевіряє, чи при вводі користувачем рядка конкретний символ підтримується цим станом. `check_next()` в свою чергу дивиться який з можливих станів переходу підтримує символ, що ввів користувач і повертає його, якщо такий є або False в іншому випадку. Також зберігає список усіх доступних станів `self.next_states`.

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

    @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
        return False
```

- **StartState** — початковий стан. Не виконує ніякої роботи, окрім як зберігання можливих переходів до інших станів в успадкованим `self.next_states`.
- **TerminationState** — завершальний стан, який сигналізує про успішне завершення. Сам по собі нічого не виконує.
- **AsciiState** — перевіряє відповідність конкретному символу. У собі зберігає конкретний символ з regex'у, і при перевірці у `check_self` дивиться чи введений користувачем символ відповідає тому що запам'ятали.
- **DotState** — відповідає символу `.`, який дозволяє будь-який символ. Тому при перевірці `check_self` завжди повертає True.
- **StarState** — відповідає символу `*`, що означає "нуль або більше" повторень. Цей стан обгортає інший, тому має в собі інформацію `self.checking_state`. При перевірці `check_self(self, char)` цей стан дивиться, чи виконується `self.checking_state.check_self(char)`
- **PlusState** — відповідає символу `+`, що означає "один або більше" повторень. Цей стан також обгортає інший, тому має в собі інформацію `self.checking_state`. При перевірці `check_self(self, char)` цей стан дивиться, чи виконується `self.checking_state.check_self(char)`

2.2 Побудова автомата

Клас `RegexFSM` приймає рядок регулярного виразу та будує граф станів. При обробці виразу:

- Кожен символ створює відповідний стан.
- Якщо після символу йде `*` або `+`, то створюється обгортка (`StarState` або `PlusState`), яка додається у граф.
- Між попереднім та новоствореним станом додається перехід.
- Проходимся ще раз по графу і додаєм усі можливі шляхи оскільки `StarState` дозволяє нуль повторень символу, тобто між попереднім станом має бути шлях до стану після `StarState`. Для цього використовуються дві функції - `add_empty_transitions(self)` та `recursive_star_connections(self, next_candidates, result=None)`. Перша проходиться по всіх шляхах бектрекінгом і додає нові, а друга - допоміжна, шукає всі шляхи через стан зірочки рекурсивно. Тобто враховує всі можливі переходи, наприклад у випадку `AsciiState(1)->StarState(2)->StarState(3)->AsciiState(4)` у `AsciiState(1)` буде можливість перейти зразу до `StarState(2)`, `StarState(3)` і до `AsciiState(4)`
- Додається завершальний стан.

2.3 Перевірка відповідності рядка

Метод `check_string()` по черзі проходить символи рядка, переходячи з одного стану до іншого. Якщо для символу не знайдено відповідного переходу або не вдалось дійти до термінального стану — повертається `False`. Якщо вдалося дійти до термінального стану — повертається `True`.

3. Приклад використання

Щоб запустити програму, треба запустити файл `regex.py`, попередньо ввівши свій regex_pattern та стрічку для перевірки, як показано знизу.

```
if __name__ == "__main__":
    regex_pattern = "ab*cd+ee."
    regex_compiled = RegexFSM(regex_pattern)
    print(regex_compiled.check_string("abbbbcddee0"))  # True
```

Цей приклад перевіряє, чи рядок `abbbbcd9ee0` відповідає виразу `ab*cd+ee.`, що означає:

- `a` — один символ a;
- `b*` — нуль або більше b;
- `c` — символ c;
- `d+` — один або більше d;
- `e` — символ e (двічі);
- `.` — будь-який символ.

Проте даний автомат не вміє зчитувати рядки з .* та .+, бо я не додумався як це зробити і ще не починав готуватись до сесії(