МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХЕРСОНСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ
Факультет комп'ютерних наук, фізики та математики
Кафедра комп'ютерних наук та програмної інженерії

Скінчені автомати, регулярні мови та їх використання у програмуванні

Кваліфікаційна робота (проєкт)
на здобуття ступеня вищої освіти «бакалавр»

Виконала: здобувачка 4 курсу 12-431 групи
Спеціальність: 122 Комп'ютерні науки
Освітньо-професійна програма: Комп'ютерні науки
Чередніченко Анна Сергіївна

Керівник: Доктор фізико-математичних наук, професор
Львов М. С.
Рецензент: к. т. н., доц., Огнєва О.Є.

Івано-Франківськ – 2024

ЗМІСТ

ВСТУП

Актуальність: Скінченні автомати та регулярні мови займають фундаментальне місце в інформатиці. Їхній вплив пронизує декілька напрямків, зокрема:

- Побудова компіляторів: Лексичний аналіз мов програмування часто значною мірою спирається на принципи скінченних автоматів.
- Обробка тексту та перевірка даних: Регулярні вирази діють як потужні інструменти для зіставлення шаблонів, пошуку та перевірки структурованих текстових даних.
- Мережева безпека: Регулярні вирази підсилюють аналіз на основі шаблонів і систем правил, що застосовуються в механізмах виявлення та запобігання вторгнень.

Мета дипломної робота, закріпити теоретичні основи скінченних автоматів та регулярних мов, одночасно демонструючи їх широке застосування за допомогою програмних конструкцій в універсальному середовищі Python. Дослідити теоретичні основи та практичні реалізації скінченних автоматів та регулярних мов в контексті програмування на мові Python. Забезпечити всебічне розуміння їх обчислювальних можливостей, обмежень та алгоритмічного представлення, одночасно підкреслюючи їх значення в різних прикладних областях.

Об’єкт: Ключовим об’єктом аналізу в цій дисертації є фундаментальна концепція регулярних мов. В роботі ретельно вивчимо їх формальні властивості, дослідимо складний взаємозв’язок між скінченними автоматами (як детермінованими, так і недетермінованими) та мовою регулярних виразів.

Предмет:

- Теоретичні конструкції скінченних автоматів (стани, переходи, формальні визначення).

- Вивчення детермінованих скінченних автоматів (DFA) та недетермінованих скінченних автоматів (NFA), включаючи їх еквівалентність з регулярними виразами.
- Вивчення теоретичних обмежень, що накладаються регулярними мовами.
- Алгоритмічне проектування та реалізація симуляторів скінченних автоматів на основі Python.
- Функціональність модуля re та міркування щодо оптимізації.

РОЗДІЛ 1. ТЕОРЕТИЧНІ ЗАСАДИ

1.1 Формальні мови та регулярні мови

Формальна мова в контексті комп'ютерних наук слугує точно визначеним набором рядків, побудованих з певного алфавіту символів. Цей суворий формалізм контрастує з природними мовами, такими як англійська, яким притаманні двозначності та неправильності. Ключовим компонентом у розумінні формальних мов є ієрархія Хомського, запропонована Ноамом Хомським у 1950-х роках ([1]). Ця ієрархія пропонує засіб класифікації різних формальних мов відповідно до складності обчислювального апарату, необхідного для їх розпізнавання [2, с.13-15].

1. Тип 0: Рекурсивно перечислювані мови: на вершині ієрархії Хомського знаходяться рекурсивно перечислювані мови. Ці мови охоплюють усі можливі мови, які може розпізнати машина Тюрінга, теоретична модель обчислень. Мови цієї категорії можуть бути надзвичайно складними.
2. Тип 1: Контекстно-залежні мови: контекстно-залежні мови вимагають для свого визначення контекстно-залежних граматик. Хоча вони менш складні, ніж необмежені мови, контекстно-залежні мови знаходять застосування у таких сферах, як обробка природної мови, оскільки вони враховують певний ступінь контекстної залежності.
3. Тип 2: Контекстно-вільні мови: далі за ієрархією йдуть контекстно-вільні мови. Ці мови піддаються опису за допомогою контекстно-вільних граматик і можуть бути ефективно розібрані за допомогою детермінованих автоматів. Їх важливість виникає в таких областях, як проектування мов програмування, де синтаксичні структури часто дотримуються контекстно-вільних принципів [2, с.13-15].

4. Тип 3: Регулярні мови: в основі ієрархії Хомського лежать регулярні мови. Саме цей клас мов є основним об’єктом дослідження в цій дисертації.

Регулярні мови володіють наступними ключовими властивостями:

- Розв’язність: Визначити, чи належить певний рядок до регулярної мови, завжди можна за допомогою скінченного автомата.
- Властивості замикання: Регулярні мови відображають замикання при виконанні ряду операцій. Ці операції включають
- Об’єднання: Об’єднання двох регулярних мов створює нову регулярну мову.
- Перетин: Аналогічно, перетин двох звичайних мов також є звичайною мовою.
- Доповнення: Доповнення регулярної мови (всі рядки, яких немає у мові) саме по собі є регулярною мовою.
- Конкатенація: Конкатенація двох регулярних мов дає регулярну мову.
- Зірка Клені: Застосування операції зірки Клені (нуль або більше повторень) до регулярної мови призводить до отримання регулярної мови [2, с.13-15].

Теоретична сила регулярних мов випливає з їх еквівалентності як детермінованим скінченним автоматам (ДСА), так і недетермінованим скінченним автоматам (НСА). Скінченний автомат – це концептуальна машина, що характеризується набором станів, алфавітом вхідних символів і функцією переходу, яка визначає, як автомат реагує на подачу символу в певному стані. Якщо автомат має однозначний перехід для кожного символа в кожному стані, він є детермінованим скінченним автоматом (ДСА). Недетермінований скінченний автомат (НСА) може допускати кілька можливих переходів на символ або переходи на порожньому рядку (епсилон-переходи).

Регулярні вирази пропонують компактну і практичну текстову нотацію для визначення регулярних мов. Цей зв’язок між регулярними виразами, скінченними автоматами і регулярними мовами є дуже важливим. Кожна регулярна мова може бути переведена у скінченне автоматне представлення і виражена у вигляді регулярного виразу. І навпаки, як скінченні автомати, так і регулярні вирази мають здатність породжувати рядки, що відповідають відповідній регулярній мові.

Відносна простота регулярних мов, порівняно з іншими класами формальних мов, має далекосяжні наслідки. Хоча регулярні мови не можуть повністю відобразити хитросплетіння природних мов або моделювати обчислення довільної складності, їхня розв’язність і добре вивчені властивості роблять їх винятково придатними для різноманітних практичних застосувань. Ми розглянемо такі застосування в наступних розділах цієї дисертації.

Регулярні мови, як формальна модель в теорії обчислень, мають декілька важливих властивостей, які мають глибоке значення як для теоретичного аналізу, так і для практичних застосувань. Однією з найважливіших з цих властивостей є їхня замкненість. Клас мов вважається замкнутим за певною операцією, якщо застосування цієї операції до членів цього класу завжди призводить до того, що мова все ще належить до того ж класу [2, с.13-15].

Розглянемо ключові властивості замикання регулярних мов:

- Об’єднання: якщо задано дві регулярні мови, L1 та L2, то їх об’єднання (L1 ∪ L2) також є регулярною мовою. Інтуїтивно це означає, що якщо у нас є два механізми зіставлення шаблонів (наприклад, скінченні автомати або регулярні вирази), ми можемо побудувати новий механізм, який об’єднає шаблони, розпізнані обома механізмами.
- Перетин: подібно до об’єднання, перетин двох регулярних мов (L1 ∩ L2) сам по собі є регулярною мовою. Мова, отримана в

результаті перетину, містить саме ті рядки, які розпізнаються обома вихідними мовами.

- Доповнення: доповнення регулярної мови L (позначається як L’) складається з усіх рядків того самого алфавіту, які не є членами вихідної мови L. Важливо, що регулярні мови є замкненими за операцією доповнення.
- Конкатенація: конкатенація двох регулярних мов L1 і L2 (позначається L1 · L2) утворює регулярну мову, яка містить рядки, що складаються з послідовності з L1, за якою слідує послідовність з L2. По суті, це означає нашу здатність з’єднувати в ланцюжок впізнавані патерни [6, с.113-114].
- Зірка Клені: замикання зірки Клені (L*) регулярної мови L дає регулярну мову, що складається з нуля або більше конкатенацій рядків, взятих з вихідної мови L. Ця властивість представляє можливість нуля або більше повторень шаблону.

Щоб проілюструвати вплив властивостей замикання, розглянемо спрощений приклад. Нехай у нас є наступні регулярні вирази:

- $L1 = ab*$ (відповідає букві "a", за якою слідує нуль або більше "b")
- $L2 = c+$ (відповідає одній або декільком буквам ‘c’)

Тепер, завдяки властивостям замикання, ми можемо встановити наступне:

- $L1 \cup L2$ – регулярна мова (відповідає шаблону в L1 або L2).
- $L1 \cdot L2$ – регулярна мова (відповідність "a" з нулем або більше "b", за яким слідує одне або більше "c").
- $(L1)'$ – регулярна мова (відповідає всьому, що не відповідає шаблону в L1) [6, с.113-114].

Властивості замикання надають потужний теоретичний інструментарій для маніпулювання та аналізу регулярних мов. На практиці ці

властивості формують основу для побудови складних регулярних виразів для вирішення складних проблем зіставлення шаблонів. Наприклад, функція пошуку і заміни у текстовому редакторі, яка часто підтримує регулярні вирази, неявно використовує ці властивості замикання.

Крім того, розуміння властивостей замикання дає уявлення про обмеження звичайних мов. Певні типи шаблонів, зокрема паліндроми та вирази, що вимагають збалансованих дужок, лежать за межами виражальних можливостей звичайних мов. Усвідомлення таких обмежень стає вирішальним при виборі відповідного формалізму для різноманітних обчислювальних задач [6, с.113-114].

Докази, що демонструють замкненість регулярних мов за допомогою цих операцій, часто включають конструктивні процеси – показують, як побудувати скінченні автомати або регулярні вирази для результуючих мов, маючи автомати або вирази для початкових мов. Детальне вивчення таких доведень може стати темою для подальших досліджень.

1.2 Скінченні автомати

Скінченні автомати є абстрактними обчислювальними моделями, які відіграють фундаментальну роль у теорії формальних мов і мають широке практичне застосування. За своєю суттю скінченні автомати втілюють надзвичайно просту, але потужну концепцію. Розглянемо ключові компоненти та їхнє функціонування:

Скінченний автомат формально визначається як 5-кортеж $(Q, \Sigma, \delta, q0, F)$, де

- Q – скінченна множина станів. Кожен стан автомата відповідає певній точці у процесі аналізу вхідного рядка.
- Σ: Скінченний алфавіт вхідних символів.

- δ: Функція переходу. Ця функція визначає, як автомат переходить з одного стану в інший при обробці вхідного символу. А саме, $\delta: Q \times \Sigma \rightarrow Q$.
- q0: Визначений початковий стан. Це стан, з якого автомат починає обробляти вхідні дані [1, с.22-23].
- F: Множина допустимих (або кінцевих) станів (де $F \subseteq Q$). Якщо після обробки всього вхідного рядка автомат переходить у стан, що належить F, то рядок вважається прийнятим.

Детерміновані скінченні автомати (DFA):

У детермінованому скінченному автоматі функція переходу (δ) має точно один вихідний стан для будь-якої комбінації поточного стану і вхідного символу. Простіше кажучи, існує єдиний, передбачуваний і однозначний шлях через автомат для будь-якого вхідного рядка. Автомат детерміновано обирає свій наступний стан.

Недетерміновані скінченні автомати (NFA):

Недетерміновані скінченні автомати вводять важливу відмінність. Функція переходу НСА може створювати декілька можливих вихідних станів для заданої пари вхідних символів. Крім того, NFA може здійснювати переходи на порожньому рядку (епсилон-переходи, часто позначаються символом ε), не споживаючи вхідних даних. Концептуально, NFA "вгадує" або досліджує декілька обчислювальних шляхів одночасно при зустрічі з недетермінованістю. Рядок вважається прийнятим, якщо хоча б один з цих обчислювальних шляхів приводить до допустимого стану [1, с.22-23].

Еквівалентність регулярним виразам:

Ключовим результатом теорії автоматів є те, що будь-який регулярний вираз можна перетворити в еквівалентний НФА, а будь-який НФА можна перевести в еквівалентний ДФА. Більше того, регулярні мови - це саме ті мови, які визначаються ДФА, НФА або регулярними виразами. Ця триєдність представлень утворює потужний міст між

теоретичними конструкціями та практичними інструментами зіставлення шаблонів [1, с.22-23].

Розглянемо простий DFA, призначений для розпізнавання рядків з алфавіту {0, 1}, які закінчуються парною кількістю 0. Цей DFA може мати три стани:

- q0 (початковий стан): Парна кількість 0, що зустрілися на даний момент.
- q1: Непарна кількість зустрінутих 0.
- q2: Неприйнятний стан (ми хочемо закінчити на парній кількості).

Перехідна функція δ буде визначена для перехода між цими станами відповідно до вхідних даних (або 0, або 1). Важливо, що q0 буде нашим прийнятним станом.

DFA та NFA:

Незважаючи на їх теоретичну еквівалентність, практичні міркування часто надають перевагу одному типу автоматів над іншим. DFA, завдяки своїй детермінованій природі, можуть бути реалізовані з більшою ефективністю, ніж NFA. Однак НФА часто дозволяють більш стисло і безпосередньо представляти певні закономірності. Цей компроміс між складністю представлення та виконання часто зустрічається в обчислювальних галузях [10, с.39-40].

Скінченні автомати лежать в основі численних інструментів і технологій. Лексичні аналізатори у компіляторах використовують автомати для розрізнення мовних лексем. Системи виявлення мережевих вторгнень використовують скінченні автомати для розпізнавання сигнатур атак. Движки регулярних виразів, які широко використовуються в задачах програмування, мають коріння в теорії скінченних автоматів.

При проектуванні DFA часто можна побудувати автомат, який розпізнає задану мову з меншою кількістю станів, ніж наївна або початкова конструкція. Володіння мінімізованим DFA має кілька переваг:

1. Зменшення складності: Спрощений DFA з меншою кількістю станів покращує розуміння і робить подальший аналіз більш керованим.
2. Оптимізація: Реалізація DFA зазвичай виграє від мінімізації станів. Менша кількість станів може призвести до зменшення вимог до пам’яті та потенційно швидшого часу виконання.

Тоді центральним питанням стає: як нам систематично виявляти та об’єднувати надлишкові стани в DFA, щоб отримати мінімальний еквівалентний автомат? На щастя, існують добре відпрацьовані алгоритми для виконання цього процесу мінімізації станів. Розглянемо два основні методи:

Алгоритм розбиття працює шляхом послідовного уточнення розбиття (поділу) станів DFA. Він працює наступним чином:

- Початкове розбиття: Починається з розділення станів DFA на дві очевидні частини - стани, що приймають, і стани, що не приймають.
- Уточнення: Ітеративно досліджуємо кожне існуюче розбиття. Для заданого розбиття "P" та вхідного символу "a" подивіться, яких станів досягають всі стани у "P" на цьому вхідному символі "a". Якщо ці стани потрапляють у декілька існуючих розділів, розбийте розділ ‘P’ відповідно.
- Завершення: Процес зупиняється, коли ітерація уточнення не призводить до подальшого розбиття жодного розділу [10, с.39-40].
- Мінімальна побудова ДФА: З останнього розділу будується мінімізований DFA. Кожен розділ стає єдиним станом у мінімізованому DFA, а переходи визначаються відповідно до поведінки початкового DFA.

Алгоритм заповнення таблиці надає альтернативний, і часто візуально інтуїтивно зрозумілий, підхід до мінімізації станів. Основні кроки включають

- Побудова таблиці: Створіть таблицю, де кожна клітинка (x, y) відповідає парі станів з вихідного DFA.
- Маркування: Спочатку позначте всі клітинки (x, y), де стан "x" є прийнятним, а стан "y" - неприйнятним (або навпаки). Ці пари станів чітко розрізняються.
- Поширення: Ітеративно розширюємо позначені клітинки. Якщо існує вхідний символ ‘a’ такий, що перехід зі стану ‘x’ на ‘a’ призводить до стану ‘p’, а перехід зі стану ‘y’ на ‘a’ призводить до стану ‘q’, і клітинка (p, q) вже позначена, то також позначаємо клітинку (x, y). По суті, ми визначаємо пари станів, майбутню поведінку яких можна передбачити за певними вхідними даними [10, с.39-40].
- Мінімізація: Після того, як всі можливі клітинки були позначені, непозначені пари станів вважаються еквівалентними і можуть бути об’єднані для отримання мінімізованого ДСА.

Хоча повний розгляд прикладу DFA може бути дуже довгим, розглянемо ідею, що якщо два стани в DFA завжди переходять в ідентичні наступні стани для кожного вхідного символу, і обидва з них є або прийнятними, або неприйнятними, то ці стани є фактично нерозрізнюваними з точки зору мови і кандидатів на об’єднання.

Методи мінімізації станів слугують не лише практичним інструментом для спрощення автоматів, але й мають теоретичне значення. Існування добре визначеного алгоритму мінімізації посилює уявлення про те, що для кожної регулярної мови існує унікальний (аж до ізоморфізму) мінімальний ДФА, який виступає її стандартним представленням [10, с.39-40].

Обидва алгоритми, описані тут, піддаються ручному виконанню на менших ДФА і можуть бути легко реалізовані в середовищі кодування для автоматичної мінімізації та маніпулювання скінченними автоматами.

Одне з найважливіших використань скінченних автоматів є розпізнавання або подання мов, що має фундаментальне значення у дослідженні та створенні компіляторів для мов програмування.

Скінченні автомати без виходу мають множину заключних станів. Автомат приймає ланцюжок, якщо він переводить автомат із початкового стану в один із заключних.

Означення:

Скінченним автоматом без виходу називають систему $M = (S, I, f, s_0, F)$, у якій S-скінченна множина станів, I – скінченний вхідний алфавіт, $f : S * I \rightarrow S$ - функція переходів, визначена на декартовому добутку $S * I$, $s_0 \in S$ – початковий стан, $F \subset S$ – множина заключних (або приймаючих) станів.

Елементи вхідного алфавіту, так само, як і раніше, іменуються вхідними символами чи просто входами.

Скінченні автомати без виходу можна описувати за допомогою таблиць станів або діаграм станів. Заключні стани на діаграмах позначаються подвійними колами. Оскільки в таких автоматах є тільки вхідні символи (елементи вхідного алфавіту I), то на дугах діаграми зазначаються тільки вони.

1.3 Автомати Мілі та Мура

Автомати Мілі та Мура – це два типи кінцевих автоматів, які використовуються для моделювання роботи електронних схем. Відмінність між ними полягає у визначенні вихідного сигналу:

- Автомат Мілі: вихідний сигнал залежить лише від поточного стану автомата та вхідного сигналу.
- Автомат Мура: вихідний сигнал залежить від поточного стану автомата та попереднього вхідного сигналу.

Це прості моделі, які використовуються для описання роботи електронних схем. Існують й інші більш складні моделі, які можуть краще відповідати реальним схемам.

Електронні цифрові схеми можна класифікувати за їхнім режимом роботи:

1. Комбінаційні схеми не мають пам’яті, тобто вихідний сигнал залежить тільки від поточних вхідних даних. Прикладами є такі схеми, як керована цифрова шина, мультиплексори, дешифратори і т. д.
2. Схеми з пам’яттю – їхнє функціонування залежить від стану входів і пам’яті про попередні вхідні стани. Ці схеми можуть бути описані за допомогою теорії кінцевих автоматів.

Інакше висловлюючись, комбінаційні схеми це логічні пристрої, які просто обробляють вхідний сигнал, тобто їхні вихідні стани залежать від поточних вхідних сигналів. А схеми з пам’яттю це пристрої, які, крім реакції на вхідні сигнали, також враховують збережені в них дані. Таким чином, їхні вихідні стани залежать від попередніх вхідних сигналів та збережених у них даних.

Абстрактні автомати призначені для виконання певних функцій, які задаються розробниками. Вони можуть виконувати різноманітні завдання, такі як операції сумування, виконання мікрокоманд процесора, вибір потрібних даних з пам’яті або синтаксичний аналіз виразів. В загальному смислі, абстрактний автомат можна описати як пристрій, який виконує певні дії без урахування конкретних технічних деталей, та може бути представлений таким чином:

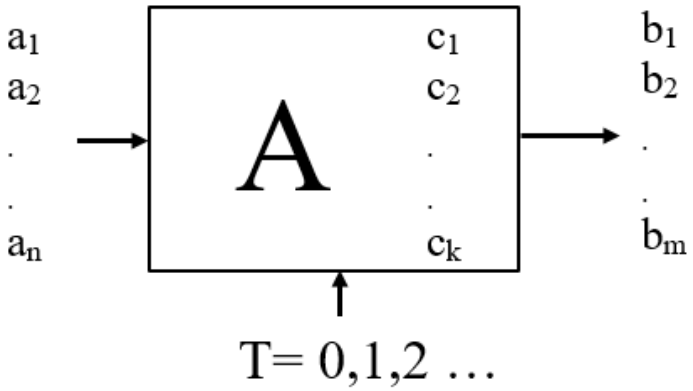

Рисунок 1.1 – Абстрактний автомат

Математично, ілюстрація схеми абстрактного автомата може бути представлена у термінах категорійної теорії, де автомат моделюється як об’єкт, а його функціонування описується морфізмами або стрілками між об’єктами категорії:

$$A =< A, B, C, \delta, \lambda >$$

Рисунок 1.2 – Формула

В цьому описі використовуються наступні позначення:

- Множина A складається зі значень, що подаються на фізичні входи автомата. У даному випадку на вході має бути встановлена певна послідовність рівнів напруги – високих і низьких, які використовуються для кодування логічних нулів і одиниць.
- Множина B включає в себе значення, які видаються на фізичних виходах автомата.
- Множина C описує внутрішній стан автомата, який відповідає його пам’яті. Важливо зауважити, що C0 позначає початковий стан автомата.
- $\delta = X * Z \rightarrow Z$ функція переходу автомата, визначає однозначно новий стан, до якого автомат повинен перейти зі стану aj.

- $\lambda = X * Z \to Y$ функція виходу автомата, визначає вихід автомата в залежності від сигналів на входах та внутрішнього стану.

Функції переходів - δ та виходів - λ не зображені на схемі для спрощення. Такий автомат працює дискретно в часі, тобто значення входів, виходів та внутрішнього стану змінюються лише в конкретні моменти часу. Прикладами таких абстрактних автоматів можуть бути тригери, регістри комп'ютерів або суматори.

Існують два основних типи абстрактних автоматів:

- Автомати Мілі.
- Автомати Мура.

Система рівнянь для опису автоматів Мілі включає:

1. Оновлення стану автомата: $c\ (t) = \ \delta(a(t), c(t-1))$
2. Визначення виходу: $b(t) = \ \lambda\ (a(t), c(t-1))$

Для автоматів Мура:

1. Оновлення стану автомата: $c(t) = \ \delta(a(t), c(t-1))$
2. Визначення виходу: $b(t) = \ \lambda\ (a(t), c(t))$

Обидва типи автоматів використовують оновлення стану залежно від вхідного сигналу та попереднього стану, проте у автоматів Мілі визначення виходу залежить від попереднього стану, тоді як у автоматів Мура – від поточного. Ці формули показують, що стан автомата $c(t)$ у поточний момент часу є результатом функції, що залежить від стану автомата у попередній момент часу та вхідного сигналу. Основна відмінність між автоматами Мілі та Мура полягає у тому, як визначається вихідний сигнал. У випадку автоматів Мілі вихідний сигнал визначається за допомогою вхідного сигналу $a(t)$ та стану автомата у попередній момент часу $c(t-1)$. У випадку автоматів Мура вихідний сигнал визначається вхідним сигналом $a(t)$ та поточним станом автомата $c(t)$.

Також важливо зазначити, що можливий перехід від одного типу автомата до іншого і навпаки. При переході від автомата Мілі до автомата Мура кількість внутрішніх станів автомата залишається незмінною, але при зворотньому переході кількість внутрішніх станів може збільшитися.

Отже, автомат типу Мілі генерує вихідний сигнал лише в той момент, коли вхідний сигнал змінюється, залежно від його попереднього стану. При цьому тривалість вихідного сигналу не залежить від тривалості вхідного сигналу, а відображається лише на його наявності.

В автоматах типу Мура вихідний сигнал залежить від стану автомата у поточний момент часу, тобто автомат продовжує генерувати певний вихідний сигнал до тих пір, поки не змінить свій стан.

Наступним кроком буде розгляд методів задання автоматів. Як було зазначено раніше, абстрактний автомат представляє собою комбінацію вхідного та вихідного алфавітів, множини внутрішніх станів і функцій, які визначають переходи та виходи. Проте, функції виходу (функції b) зазвичай не є визначеними, і поведінка автомата повинна бути описана іншими способами. Основними методами задання абстрактних автоматів є використання графів та таблиць переходів і виходів.

Граф автомата – це зв’язний орієнтований граф, де вершини відображають внутрішні стани автомата, а ребра представляють переходи між цими станами. На малюнку наведено граф автомата Мілі.

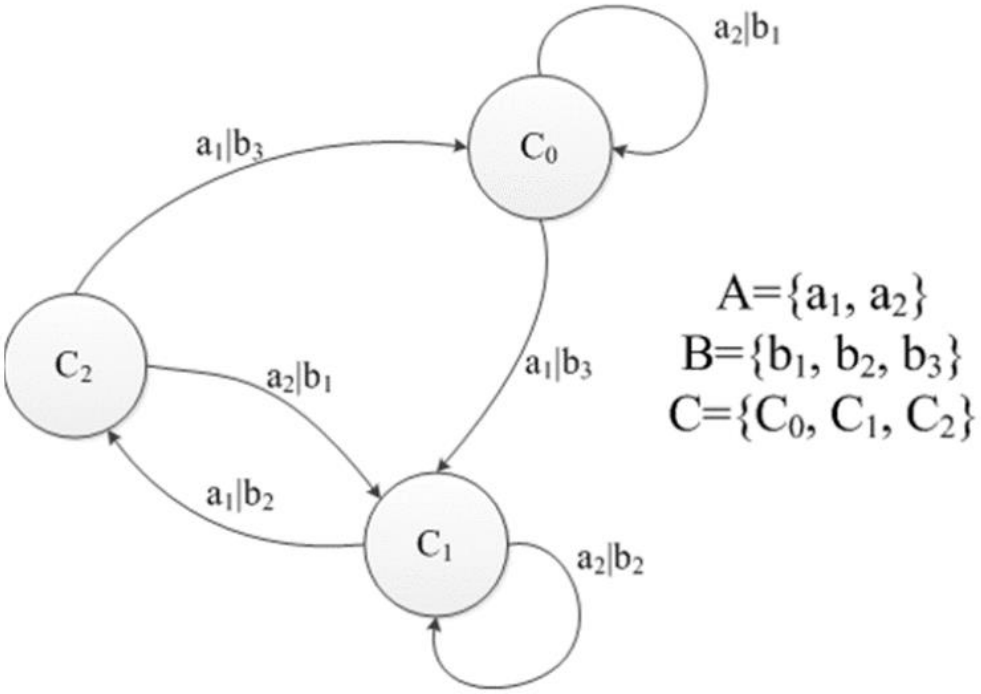

Рисунок 1.3 – Граф автомата Мілі

Для графа Мілі на стрілках потрібно вказати вхідні та вихідні символи. Вихідні символи мають бути написані над стрілками, щоб підкреслити залежність вихідного стану від попереднього стану автомата.

У графі автомата Мура на стрілках мають бути вказані лише вхідні символи, а вихідні символи повинні бути показані біля вершин, як показано на малюнку.

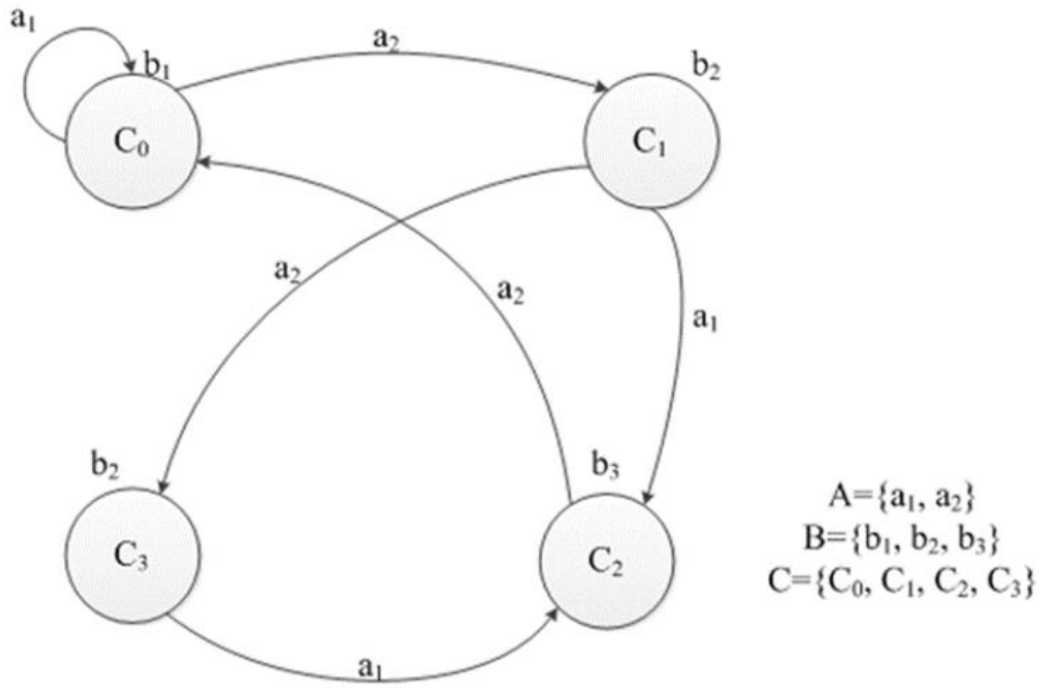

Рисунок 1.4 – Граф автомата Мура

Важно відзначити, що якщо кожна вершина має стільки вихідних ребер, скільки є вхідних символів, то автомат вважається повним. Іншими словами, якщо для кожної вершини передбачені переходи для всіх можливих вхідних символів, то автомат вважається повним. В наведених прикладах автомат Мілі може бути повним, у той час як автомат Мура є частково повним.

1.4 Регулярні вирази та регулярні множини. Теорема Кліні про збіг класів автоматних множин та регулярних множин

Регулярні вирази – це стисла, декларативна мова для визначення шаблонів у тексті. Їхнє походження лежить у площині формальної теорії мови та теоретичної моделі скінченних автоматів. В основі регулярних виразів лежить механізм визначення регулярних мов. Розглянемо основні будівельні блоки і те, як вони складаються.

Стандартна мова регулярних виразів включає наступні ключові елементи:

Літерали: звичайні символи точно відповідають самим собі. *Наприклад*, літера ‘a’ відповідає входженню ‘a’ у рядку.

Метасимволи: спеціальні символи мають значення для зіставлення зі зразком. До них належать:

1. .: Крапка відповідає будь-якому символу, крім нового рядка.
2. *: "Біла зірка" вказує на нуль або більше повторень попереднього елемента.
3. +: Символ "плюс" відповідає одному або більше повторенням попереднього елемента.
4. ?: Робить попередній елемент необов’язковим (нуль або одне входження).
5. []: Класи символів відповідають одному символу з набору. *Приклад*: [a-z] відповідає будь-якій малій літері.
6. ^: У регулярному виразі відповідає початку рядка.
7. $: Відповідає кінцю рядка.
8. |: Вказує на чергування ("або"). *Наприклад*, ‘abc|xyz’ відповідає або ‘abc’, або ‘xyz’.
9. Групування: Дужки () можна використовувати для групування елементів для застосування кванторів або визначення пріоритету [12, с.139-140].

Семантичний приклад:

Розберемо регулярний вираз:

$$/\text{^}[a-z0-9.\_\%+-]+@[a-z0-9.-]+\.[a-z]\{2,\}\$/$$

Рисунок 1.5 – Регулярний вираз

Цей шаблон зазвичай використовується для базової перевірки email-адрес:

1. ^: Стверджує, що шаблон повинен починатися з початку рядка.

2. [a-z0-9._%+-]+: Відповідає одній або декільком малим літерам, цифрам або вказаним спеціальним символам.
3. @ : Відповідає буквеному символу ‘@’.
4. [a-z0-9.-]+: Подібна структура, що дозволяє компоненти піддоменів.
5. \.: Відповідає буквальній крапці.
6. [a-z]{2,}: Відповідає домену верхнього рівня, що складається з двох або більше малих літер[12, с.139-140].
7. $: Забезпечує відповідність шаблону до всього рядка.

Визначення регулярних виразів над кінцевим алфавітом A за допомогою індукції:

Базис індукції:

1. Порожня множина (∅): регулярне вираження, яке не відповідає жодному рядку.
2. Символ алфавіту (a, де a ∈ A): регулярне вираження, яке відповідає рядку, що складається з одного символу "a".
3. Порожнє слово (Λ, де Λ ∈ A): * регулярне вираження, яке відповідає пустому рядку.

Індуктивний перехід: нехай s та t – регулярні вирази. Тоді:

1. (s + t): Якщо s і t – регулярні вираження, то s + t – це також регулярне вираження, яке відповідає будь-якому рядку, що складається з конкатенації рядків, які відповідають s і t відповідно.
2. (st): Якщо s і t – регулярні вираження, то st – це також регулярне вираження, яке відповідає будь-якому рядку, що складається з конкатенації рядків, які відповідають s і t відповідно.
3. (s):* Якщо s – регулярне вираження, то s* – це також регулярне вираження, яке відповідає будь-якому рядку, що складається з будь-якої кількості повторень рядків, які відповідають s.

Індуктивно задаємо множину регулярних виразів над кінцевим алфавітом A. Базис складають порожня множина, символи алфавіту та

порожнє слово. З двох даних регулярних виразів s і t можна отримати нові регулярні вираження s + t, st і s*. Жодних інших регулярних виразів не існує.

Кожне регулярне вираження над алфавітом A визначає певну множину (мову) $L \subseteq A*$.

Базис індукції:

1. Регулярне вираження ∅ визначає порожню множину ∅.
2. Регулярне вираження a, де $a \in A$, визначає множину {a}, що складається з одного символу "a".
3. Регулярне вираження Λ, де $\Lambda \in A*$, визначає множину {Λ}, що складається з порожнього рядка.

Індуктивний перехід: нехай регулярні вирази s, t визначають відповідно множини S, $T \subseteq A*$. Тоді:

1. Якщо регулярні вираження s і t визначають множини S і T відповідно, то регулярне вираження s + t визначає об'єднання $S \cup T$ цих множин.
2. Якщо s і t - регулярні вираження, то st визначає добуток ST цих множин.
3. Якщо s - регулярне вираження, то s* визначає ітерацію $S*$ множини S.

Множина L слів у кінцевому алфавіті A називається регулярною множиною (мовою), якщо вона визначається деяким регулярним виразом. Існує багато різних регулярних виразів, які можуть описувати одне й те ж регулярне множество. Не всі множини слів є регулярними. Наприклад, множина слів, які містять парну кількість символів "a", не є регулярною.

Існують алгоритми перетворення регулярних виразів в скінченні автомати і навпаки. Скінченні автомати – це модель для обчислення, яка може розпізнавати регулярні мови. Регулярні множини (мови) мають

багато застосувань в комп’ютерних науках, наприклад, в компіляторах, текстових редакторах і біоінформатиці.

Сучасні механізми регулярних виразів, які можна знайти у більшості мов програмування, часто виходять за межі основних теоретичних можливостей регулярних виразів. Ось деякі з найпоширеніших розширень:

- Зворотні посилання: зворотні посилання, що позначаються \n (де "n" – число), надають можливість знайти той самий текст, що був раніше знайдений групою перехоплення. Ця можливість дозволяє виявляти повторювані шаблони.
- Прямі та зворотні твердження: вони дають змогу стверджувати умови, пов’язані зі збігом, не витрачаючи ці символи. Наприклад, позитивне випередження (?=...) перевіряє, чи слідує за шаблоном щось, але не включає його до збігу.

Важливо зазначити, що деякі розширені функції надають можливості, які порушують теоретичну еквівалентність між регулярними виразами та скінченними автоматами. Такі можливості, як зворотні посилання, можуть фактично дозволити описувати певні нерегулярні мови [12, с.139-140].

Регулярні вирази є незамінним інструментом у багатьох сферах:

- Пошук і маніпулювання текстом: Пошук шаблонів, вилучення підрядків, операції пошуку та заміни у текстових редакторах і середовищах програмування.
- Перевірка вхідних даних: Використовується для перевірки відповідності даних, таких як адреси електронної пошти, номери телефонів або певні формати, очікуваним шаблонам.
- Синтаксичний аналіз: Регулярні вирази часто використовуються для лексичного аналізу в компіляторах, вилучення структур з файлів журналів або обробки структурованих даних.

Незважаючи на те, що регулярні вирази є надзвичайно потужним інструментом, їх слід використовувати з обережністю. Надто складні регулярні вирази можуть стати сумнозвісними для читання і налагодження. Для завдань, що включають сильно вкладені або рекурсивні структури (наприклад, розбір повних мов програмування), регулярні вирази можуть досягти своїх обмежень, що вимагає більш потужних підходів, заснованих на синтаксичному аналізаторі.

Виразна сила класичних регулярних виразів безпосередньо випливає з їх еквівалентності скінченним автоматам. Оскільки скінченні автомати працюють з обмеженою пам’яттю і не здатні "рахувати" як завгодно складні повторення, це обмеження переноситься і на область регулярних виразів. Розглянемо деякі конкретні приклади [12, с.139-140].

Принципово неможливо побудувати регулярний вираз, який бездоганно співпадає з рядками зі збалансованими круглими дужками довільної глибини вкладеності. Розглянемо спрощений приклад – мову, що допускає послідовності збалансованих дужок ‘()’. Жоден регулярний вираз не може гарантувати, що для кожного відкриваючого ‘(‘ існує відповідний закриваючий ‘)’, який коректно закриває правильний набір дужок. Скінченний автомат не може "запам’ятати", скільки відкритих дужок він зустрів під час обробки рядка.

Паліндроми - це слова або фрази, які читаються однаково в прямому і зворотному напрямку (наприклад, "рівень", "радар"). За винятком тривіальних паліндромів фіксованої довжини, визначити регулярний вираз для правильного розпізнавання довільного паліндрома неможливо. Основна проблема полягає у визначенні центру рядка з дзеркальною поведінкою навколо нього – те, з чим не може впоратися обмежена пам’ять регулярних мов [12, с.139-140].

Уявімо спробу представити мову, яка складається з унарного кодування простих чисел (наприклад, ‘111’ для числа 3, ‘11111’ для 5). Жоден регулярний вираз не може визначити цю мову. Ідентифікація простих

чисел пов’язана з властивостями подільності, які виходять за межі можливостей регулярних виразів.

Формальним інструментом, який часто використовується для виявлення обмежень регулярних мов, є лема накачування для регулярних мов. По суті, ця лема стверджує, що будь-який достатньо довгий рядок, що належить регулярній мові, можна "прокачати". Під "прокачуванням" мається на увазі знаходження середньої частини рядка, яку можна повторити будь-яку кількість разів, і при цьому отримати рядок, що належить цій мові. Можливість знаходити рядки, які не мають цієї властивості, дає змогу довести, що певні мови є нерегулярними.

Важливо пам’ятати, що багато сучасних рушіїв регулярних виразів включають розширення, такі як зворотні посилання. Хоча такі можливості і корисні, вони виходять за рамки чистої регулярної мови. За допомогою зворотних посилань можна описувати деякі нерегулярні шаблони (наприклад, співставлення "www" з наступним таким самим словом вимагає концепції зберігання або "запам’ятовування" першого слова для співставлення) [12, с.139-140]. Окрім цих теоретичних обмежень, при використанні регулярних виразів важливо пам’ятати про практичні міркування:

- Катастрофічне відкочування: Погано побудовані регулярні вирази, особливо ті, що містять вкладені квантори, можуть призвести до того, що механізм регулярних виразів витрачатиме експоненціально багато часу на пошук збігів у зворотному напрямку. Це може суттєво вплинути на продуктивність.
- Читабельність: Зі збільшенням складності регулярних виразів, вони часто стають менш зручними для супроводу і складнішими для розуміння іншими (або навіть для вас самих у майбутньому). Належною практикою є широке використання коментарів у складних виразах, щоб пояснити їхній зміст.

Коли проблеми зіставлення шаблонів або синтаксичного аналізу виходять за межі того, що можуть представити регулярні вирази, зазвичай звертаються до більш потужних формалізмів. Контекстно-вільні граматики та пов’язані з ними методи синтаксичного аналізу дозволяють обробляти вкладені структури, такі як збалансовані дужки або навіть синтаксис мови програмування [12, с.139-140].

Головна теза теореми Кліні: «Класи регулярних множин та автоматних мов збігаються».

Твердження: Мова $L \subseteq A$ є елементом півкільця регулярних мов R(A) в алфавіті A тоді і тільки тоді, коли вона допускається деяким кінцевим автоматом.

Існує еквівалентність між регулярними мовами та кінцевими автоматами. Будь-яку мову, яку можна розпізнати кінцевим автоматом, можна описати регулярним виразом, і навпаки.

Доведення:

Зліва направо: Доводиться, що для будь-якого регулярного виразу R існує кінцевий автомат, який розпізнає мову, визначену R. Це можна зробити за допомогою індукції за структурою регулярного виразу.

Справа наліво: Доводиться, що для будь-якого кінцевого автомата A існує регулярний вираз, який описує мову, розпізнавану A. Це можна зробити за допомогою побудови регулярного виразу з перехідної функції автомата.

Важливі наслідки:

Теорема Кліні дає чітке та лаконічне визначення класу регулярних мов. Робить аналіз та перетворення мов більш зручними, оскільки можна використовувати еквівалентні автоматні та регулярні представлення. Має широке застосування в компіляторах, текстових редакторах, біоінформатиці та інших галузях.

Приклади: Множина рядків, що складаються з будь-якої кількості символів "a" або "b", може бути описана регулярним виразом "(a|b)*".

Множина рядків, що починаються з "a" і закінчуються "b", може бути описана регулярним виразом "a.*b".

Висновок: Теорема Кліні – це фундаментальний результат в теорії автоматів та формальних мов, який має значний вплив на наше розуміння мов, які можна обробляти за допомогою комп’ютерів.

РОЗДІЛ 2. ОСНОВНІ ТЕОРЕТИЧНІ ПОЛОЖЕННЯ ТА ПОСТАНОВКА ЗАДАЧІ

2.1 Побудова скінченних автоматів у Python

В основі представлення автоматів у Python лежить вибір відповідних структур даних. Ось декілька поширених підходів:

Словники (для DFA):

- Стани можуть бути ключами словника.
- Значення, пов’язані з кожним ключем, стають іншим словником: Внутрішній словник зіставляє вхідні символи з результуючим наступним станом.
- Наприклад: { ‘q0’: {‘0’: ‘q1’, ‘1’: ‘q0’}, ‘q1’: {‘0’: ‘q2’, ‘1’: ‘q0’} ...}
- Списки суміжності (NFA):
- Словник, де ключами є стани.
- Значеннями є списки, що представляють можливі переходи на різних вхідних символах (включаючи епсилон-переходи).

Об’єктно-орієнтований підхід:

- Визначення класів станів і переходів може покращити читабельність і полегшити розширення. Це стає все більш корисним з розширеннями або додаванням метаданих про стани та переходи.

Моделювання DFA:

Вибір структури даних у Python добре підходить для моделювання детермінованих скінченних автоматів:

1. Ініціалізація: Створення вибрану вами структуру даних (наприклад, словник) для представлення DFA.
2. Відстеження стану: Підтримування зміни current_state, ініціалізовану початковим станом.

3. Обробка вхідних даних: Ітерація над вхідним рядком символ за символом:

Використання поточний стан і поточний символ для індексації структури даних, визначаючи наступний стан.

- Оновлення поточного стану до наступного стану.

Прийняття: Після обробки повного введення перевірте, чи поточний стан є прийнятним станом.

Моделювання NFA:

Недетермінізм вимагає модифікації підходу:

1. Кілька активних станів: Замість одного поточного стану реалізовано відстежування набору можливих активних станів.
2. Епсилон-переходи: Перед обробкою вхідного символу розширити множину активних станів, відстежуючи всі досяжні епсилон-переходи.
3. Паралельні обчислення: Для вхідного символу оновити множину активних станів усіма результуючими переходами, знайденими з поточних активних станів.
4. Прийняття: Вхідний рядок буде прийнято, якщо хоча б один з кінцевих активних станів є допустимим.

Візуалізація:

Такі бібліотеки, як Graphviz (та її зв’язки з Python) пропонують ефективні інструменти для візуалізації діаграм переходів станів.

1. Створення вхідних даних Graphviz: Створіть сумісний з Graphviz опис (мова DOT), що представляє вузли (стани) і ребра (переходи) вашого автомата.
2. Рендеринг: Завантажте DOT-опис у Graphviz, щоб згенерувати візуальне зображення (PNG, SVG тощо).

Проект коду ілюструє використання бібліотеки (django-viewflow), яка пропонує можливості кінцевих автоматів у моделей Django. Це абстрагує від деяких низькорівневих конструкцій автоматів, які ми

обговорювали, надаючи більш високорівневу основу для управління станами та переходами.

- Продуктивність: Базові реалізації Python можуть стати вузьким місцем для більш великих автоматів або довгих вхідних рядків. Сценарії, критичні до продуктивності, можуть вимагати вивчення бібліотек, оптимізованих для таких завдань.
- Вибух станів (NFA): Розширення множини активних станів під час симуляції NFA може мати наслідки для продуктивності дуже недетермінованих автоматів.
- Примітка: Повна реалізація з прикладами може бути довгою. Я буду радий надати стислий фрагмент коду, що демонструє симуляцію DFA або NFA, якщо ви бажаєте!

Бібліотеки кінцевих автоматів: Python пропонує бібліотеки, спеціально орієнтовані на автомати (наприклад, переходи, Automat). Ознайомтеся з їхніми можливостями та продуктивністю, якщо це може бути застосовано у вашому проекті.

2.2 Рушій регулярних виразів Python (модуль re)

Модуль re дозволяє розробникам використовувати можливості регулярних виразів у своїх програмах. Деякі основні функції, які пропонує цей модуль, включають

- re.search(pattern, string): Сканує рядок у пошуках першого місця, де є збіг з шаблоном регулярного виразу. Повертає об’єкт Match у разі успіху, None у протилежному випадку.
- re.match(шаблон, рядок): Намагається знайти збіг з шаблоном на початку рядка. Подібно до re.search, але з неявним якорем на початку рядка.
- re.findall(шаблон, рядок): Знаходить усі збіги шаблону в рядку, що не перетинаються, і повертає список підрядків, що збігаються.

- re.sub(pattern, repl, string): Виконує операцію пошуку і заміни, замінюючи входження шаблону в рядку рядком (repl) або результатом обчислення функції заміни.

Модуль re підтримує стандартні елементи синтаксису регулярних виразів, описані у попередніх розділах, а також ряд розширень:

- Зворотні посилання: Можливість посилатися на раніше знайдені групи всередині самого шаблону (наприклад, (\w+)\s\1 для пошуку слів, що повторюються)
- Прямі та зворотні твердження: Перевірка умов, що оточують потенційний збіг, без використання символів (наприклад, $(?<=abc)def$ для пошуку "def", тільки якщо йому передує "abc").
- Іменовані групи захоплення: Присвоєння міток захопленим групам за допомогою (?P<name>...), що покращує читабельність для пошуку (наприклад, re.match $("(?P<date>\d\{4\}-\d\{2\}-\d\{2\})",\ldots))$

В основі модуля re у Python лежить гібридний підхід до порівняння регулярних виразів:

1. Компіляція: Шаблони регулярних виразів компілюються в байт-код з використанням детермінованої моделі, подібної до скінченного автомата (DFA). Ця попередня обробка переводить шаблони у форму, яка краще підходить для виконання.
2. Механізм зіставлення: Механізм може використовувати або дослідження на основі зворотного відстеження, або пряме моделювання DFA для зіставлення на основі характеристик шаблону та евристик оптимізації.

Розуміння цього процесу дає уявлення про потенційні застереження щодо продуктивності. Складні шаблони з вкладеними квантифікаторами або зворотним відстеженням можуть стати потенційними вузькими місцями, особливо з довгими вхідними рядками.

- Якорі: Якщо можливо, префіксні (^) або суфіксні ($) якорі можуть обмежити місця, де здійснюються спроби зіставлення у рядку, уникаючи зайвої роботи.
- Класи символів: Явні класи символів (наприклад, [0-9]) часто є ефективнішими за альтернативи, такі як більш загальний метасимвол.
- Групування і неперехоплення: Якщо потрібен лише загальний збіг, використовуйте групи без захоплення (?:...), щоб зменшити накладні витрати, пов’язані зі зберіганням проміжних знайдених частин.
- Профілювання: Щоб точно визначити проблеми у критичній для продуктивності ситуації, профілюйте код, щоб отримати уявлення про те, де оптимізація матиме найбільший вплив.

Наведені фрагменти коду ілюструють використання більш спеціалізованого фреймворку кінцевих автоматів (django-viewflow). Хоча регулярні вирази іноді можуть представляти прості системи переходу станів, їх основна увага приділяється зіставленню шаблонів у тексті, а не загальним робочим процесам, керованим станами, які є центральними для цього проекту.

– У ситуаціях, коли модуля re у Python може виявитися недостатньо, або якщо продуктивність стає першочерговою, варто дослідити такі бібліотеки, як regex (надає ширшу підтримку Unicode та додаткові можливості) або hyperscan - високопродуктивну бібліотеку для пошуку за кількома regex-виразами.

2.3 Реальні приклади використання

Текстові редактори та середовища розробки: Скромна функція "Знайти" в текстових редакторах та інтегрованих середовищах розробки часто

використовує регулярні вирази в своїй основі. Складні операції пошуку і заміни для рефакторингу коду значною мірою покладаються на зіставлення шаблонів.

- Аналіз журналів: Журнали сервера, програми або системи безпеки - величезні обсяги напівструктурованих текстових даних. Для вилучення важливих подій, шаблонів помилок або аномалій безпеки потрібні механізми зіставлення шаблонів, для яких ідеально підходять регулярні вирази.
- Перевірка даних: Перевірка за допомогою регулярних виразів може бути корисною для багатьох типів полів введення. Перевірка дійсних адрес електронної пошти, номерів телефонів, поштових індексів є поширеними випадками використання, коли концепції кінцевих автоматів, явно побудовані або неявно за допомогою реґекс-механізмів, забезпечують якість даних користувача.

Початковим етапом роботи компілятора або інтерпретатора є лексичний аналізатор (або токенізатор). Цей компонент розбиває вихідний код на токени - атомарні одиниці мови програмування (ключові слова, ідентифікатори, оператори тощо). Скінченні автомати та регулярні вирази забезпечують природний спосіб для цього:

Ефективно класифікувати частини вхідного коду за відповідними типами токенів.

1. Системи виявлення та запобігання вторгнень: Аналіз мережевого трафіку для виявлення відомих сигнатур атак або шаблонів шкідливого корисного навантаження досягається за допомогою систем правил, часто виражених через регулярні вирази або представлення кінцевих автоматів. Моніторинг пакетних даних або подій на рівні брандмауера вимагає ефективного зіставлення шаблонів над цими потоками.
2. Аналіз протоколів: Аналіз складних комунікаційних протоколів може бути значно спрощений за допомогою спеціалізованих

моделей кінцевих автоматів, які допомагають у налагодженні та перевірці протоколів.

3. Підсвічування синтаксису: Редактори коду, які ви, ймовірно, використовуєте для написання коду на Python, пропонують підсвічування синтаксису. Часто використовується набір регулярних виразів або простих автоматів для розпізнавання різних мовних конструкцій і візуального розрізнення їх на основі їх типів.
4. Обробка природної мови (NLP): Хоча повноцінне NLP часто потребує більш складних моделей, скінченні автомати та регулярні вирази іноді можуть знайти застосування на ранніх стадіях обробки, таких як базова фільтрація тексту, морфологічний аналіз або розпізнавання об’єктів.
5. Обчислювальна біологія: Такі галузі, як біоінформатика, що включають аналіз генетичних послідовностей, іноді використовують спеціалізовані конструкції автоматів або підходи, подібні до регулярних виразів (послідовності ДНК або білків можуть розглядатися як рядки в певному алфавіті).

Код демонструє використання фреймворку кінцевих автоматів (django-viewflow) у моделях Django. Основний принцип полягає у моделюванні чітко визначених станів (відкритий, призначений, закритий і т.д.) і програмному управлінні переходами між станами. Це більше відповідає концепції скінченних автоматів, аніж простому використанню сирих регулярних виразів. FSM, подібні до того, що використано у прикладі, часто зустрічаються у..:

- Управління робочим процесом: Відстеження різних кроків у процесі, які потребують затвердження, зміни статусу або іншої бізнес-логіки.

- Інтерфейс, керований станом: Поведінка і допустимі дії, доступні в користувацькому інтерфейсі, що керуються логікою станів моделі.

Повсюдність тексту і різних форм структурованих даних робить концепції скінченних автоматів і регулярних виразів напрочуд стійкими. Оскільки обчислювальні області постійно розвиваються, часто з’являються нові та інноваційні застосування для цих, здавалося б, теоретичних конструкцій.

ВИСНОВКИ

У цій дипломній роботі було виконано дослідження скінченних автоматів, регулярних мов та їх застосувань у контексті Python. Підсумуємо деякі основні висновки:

- Скінченні автомати та регулярні мови є потужними знаряддями для моделювання та розв’язання різноманітних обчислювальних проблем.
- Однією з ключових властивостей є їхній здатність до ефективного розпізнавання та обробки послідовностей символів.
- Регулярні мови, у свою чергу, дозволяють описувати та аналізувати ці послідовності шляхом використання різноманітних регулярних виразів.

Використання скінченних автоматів та регулярних мов у програмуванні відкриває широкі можливості для створення компактних, ефективних та легко зрозумілих алгоритмів обробки даних, розпізнавання шаблонів, перевірки належності послідовностей до певних мов та багато іншого.

Тема є важливою як для теоретичної, так і для практичної інформатики. Вивчення скінченних автоматів та регулярних мов допомагає розширити розуміння основних принципів обчислювальної теорії та забезпечує необхідні знання для розробки ефективних програмних рішень. Таким чином, ця робота відкриває шлях до подальших досліджень та застосувань у сфері програмування та інформатики.

В ході виконання роботи було досягнуто глибокого розуміння формальних моделей, що лежать в основі скінченних автоматів (DFA та NFA), їх теоретичної еквівалентності та внутрішнього зв’язку з виражальними можливостями регулярних виразів. Крім того, це дослідження виявило певні обмеження регулярних мов у представленні певного класу обчислювальних задач.

У роботі ефективно втілено теоретичне розуміння у практичні реалізації механізмів моделювання DFA та NFA з використанням мови програмування Python. Було з’ясовано, як модуль re функціонує як практичний рушій регулярних виразів, та проаналізовано стратегії оптимізації його використання.

Дослідження продемонструвало широке застосування скінченних автоматів і регулярних виразів. Реальні сценарії в обробці тексту, перевірці даних, лексичному аналізі та мережевій безпеці підкріпили теоретичні принципи, досліджені в ході роботи. У роботі пропонується кілька плідних напрямків для подальших досліджень та експериментів:

1. Дослідження, орієнтовані на продуктивність: Поглиблене профілювання продуктивності та бенчмаркінг великих автоматів з використанням таких методів, як пряме моделювання DFA, може стати основою для розробки оптимізованих бібліотек автоматів, пристосованих до робочих процесів на Python.
2. Альтернативні рушії регулярних виразів: Оцінка альтернативних рушіїв (наприклад, "regex" або "hyperscan") та компромісів при їх використанні у вузькоспеціалізованих обчисленнях дасть цінну інформацію.
3. Доменно-специфічні додатки: Розробка спеціальних автоматичних інструментів для конкретних проблемних областей, наприклад, спеціалізованих систем зіставлення шаблонів, подібних до регулярних виразів, у сфері аналізу послідовностей ДНК, може виявити унікальні практичні переваги.

Це дослідження підтверджує, що скінченні автомати та регулярні мови, незважаючи на їхнє фундаментальне значення в теоретичній інформатиці, продовжують бути важливими для сучасних різноманітних обчислювальних проблем. Важливо визнати, що володіння цими, на перший погляд, простими конструкціями дає розробникам важливий інструмент у їхньому арсеналі для вирішення проблем.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Hopcroft, John E., et al. "Introduction to Automata Theory, Languages, and Computation." Addison-Wesley, 2006.
2. Sipser, Michael. "Introduction to the Theory of Computation." Cengage Learning, 2012.
3. Linz, Peter. "An Introduction to Formal Languages and Automata." Jones & Bartlett Learning, 2012.
4. Kozen, Dexter. "Automata and Computability." Springer Science & Business Media, 2013.
5. Martin, John C. "Introduction to Languages and the Theory of Computation." McGraw-Hill Education, 2010.
6. Aho, Alfred V., et al. "Compilers: Principles, Techniques, and Tools." Pearson Education, 2006.
7. Jurafsky, Daniel, and James H. Martin. "Speech and Language Processing." Prentice Hall, 2009.
8. Grune, Dick, et al. "Modern Compiler Design." Springer Science & Business Media, 2012.
9. Russell, Stuart J., and Peter Norvig. "Artificial Intelligence: A Modern Approach." Pearson Education, 2021.
10. Medeiros, Paulo B., and Eduardo S. Laber. "PyGraphviz: Python interface to Graphviz." 2003. Available online: https://pypi.org/project/pygraphviz/
11. Salomon, David. "Data Compression: The Complete Reference." Springer Science & Business Media, 2006.
12. Kernighan, Brian W., and Rob Pike. "The Practice of Programming." Addison-Wesley Professional, 1999.
13. Pacheco, Luis. "Parallel Programming with MPI." Morgan Kaufmann, 1997.

14. Cormen, Thomas H., et al. "Introduction to Algorithms." MIT Press, 2009.
15. The Python Software Foundation. "Python Standard Library: re - Regular expression operations." Available online: https://docs.python.org/3/library/re.html

ДОДАТКИ

```
pip install django-viewflow --pre

parser = ParserFSM()
for char in '2*(1+3)':
    parser.next_step(char)
parser.calc() == 8

from django.db import models
from django_fsm import FSMField
class STATUS(models.TextChoices):
    OPEN = 'open', _('Open')
    ASSIGNED = 'assigned', _('Assigned')
    CLOSED = 'closed', _('Closed')
    DEFERRED = 'deferred', _('Deferred')
class Ticket(models.Model):
    text = models.TextField()
    status = FSMField(
        choices=STATUS.choices,
        default=STATUS.OPEN,
        protected=True
    )
    @transition(source=STATUS.OPEN, target=STATUS.ASSIGNED)
    def assign(self,user):
        self.assignee = user

>>> ticket = Ticket(text='sample post')
>>> ticket.assign(request.user)
>>> ticket.state == 'assigned'
True
>>> ticket.assign(request.user)
TransitionNotAllowed: No transition from `assigned` state

class MyModel(models.Model);
    parent_state = FSMField()
    sub_state = FSMField()

    @transition(field=parent_state, source='in_process')
    @transition(field=sub_state, source='start', target='end')
    def do_it(self):
         pass

class Base(models.Model):
    state = FSMField()

    class Meta:
        abstract = True
class Child(Base):
    @transition(field='state', source='start', target='end')
    def new_transition(self):
        pass

class Base(models.Model):
    @transition(field='state', source='new', target='approve'):
    def approve(self, user):
        self.approver = user
class Child(models.Model):
    @transition(field='state', source='new', target='approve'):
    def approve(self, user):
```

```
        self.approver = user
        self.approved_at = timezone.now()

class Email(models.Model)
    state = FSMField()
    @transition(field=state, source='new', target='sending')
    def send(self):
         try:
              self.send_mail()
         except:
             self.failed()
             raise
         else:
             self.complete()
    @transition(field=state, source='sending', target='complete')
    def complete(self):
        pass

    @transition(field=state, source='sending', target='error'):
    def failed(self):
        pass

class Ticket(models.Model):
    state = FSMField(protected=True)
>>> ticket.state = 'DONE'
AttributeError: Direct state modification is not allowed

class TicketFlow(object):
   status = State(Stage, default=STATUS.OPEN)
   def __init__(self, ticket):
       self.ticket = ticket
   @status.transition(source=STATUS.NEW, target=STATUS.DONE)
   def assign():
       pass

class TicketFlow(object):
    status = fsm.State(STATUS, default=STATUS.NEW)

    def __init__(self, ticket):
        self.ticket = ticket

    @status.setter()
    def _set_status(self, value):
        self.ticket.stage = value

    @status.getter()
    def _get_status(self):
        return self.ticket.stage

class TicketFlow(object):
    status = fsm.State(STATUS, default=STATUS.NEW)

    @status.transition(source=STATUS.DONE, target=STATUS.HIDDEN)
    def hide():
        print('base hide')

class SupportTicket(TicketFlow):
    @TicketFlow.stage.super()
    def hide(self):
        # any additional code here
        super().hide.original()
```

```
@stage.on_success()
def _get_report_stage(self, descriptor, source, target, **kwargs):
    with transaction.atomic():
        self.review.save()
        ReviewChangeLog.objects.create(
            review=self.review,
            source=source.value,
            target=target.value
        )
```