

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

ПРОГРАМНЕ СЕРЕДОВИЩЕ НАВЧАЛЬНОГО ПРИЗНАЧЕННЯ З
ТЕМИ “СУФІКСНІ ДЕРЕВА ТА СКІНЧЕНІ АВТОМАТИ –
ЕФЕКТИВНІ СТРУКТУРИ ДАНИХ ТА АЛГОРИТМИ ЗАДАЧІ
ПОШУКУ ЗРАЗКА У ТЕКСТІ”

Кваліфікаційна робота (проект)
на здобуття ступеня вищої освіти «магістр»

Виконав: здобувач 2 курсу 12-261М групи

Спеціальності: 126 Інформаційні системи та технології

Освітньо-професійної програми:
«Інформаційні системи та технології»

Миргородський Даниїл Євгенович

Керівник: доктор фізико-математичних наук,
професор Львов М.С.

Рецензент: кандидатка технічних наук, доцентка,
завідувачка кафедри програмних засобів і
технологій Херсонського національного
технічного університета Огнєва О.Є.

Івано-Франківськ – 2025

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ	4
ВСТУП	6
РОЗДІЛ 1. АНАЛІЗ ЕФЕКТИВНИХ СТРУКТУР ДАНИХ ТА АЛГОРИТМІВ ДЛЯ ЗАДАЧІ ПОШУКУ ЗРАЗКА В ТЕКСТІ.....	9
1.1. Постановка задачі пошуку зразка в тексті та огляд класичних підходів	9
1.2. Суфіксні дерева.....	10
1.2.1. Формальне визначення, властивості та переваги	10
1.2.2. Алгоритм Укконена: оптимізація побудови за лінійний час	11
1.2.3. Пошук підрядка з використанням суфіксного дерева	12
1.3. Скінченні автомати	13
1.3.1. Детерміновані та недетерміновані скінченні автомати	13
1.3.2. Алгоритм побудови ДСА за зразком для пошуку в тексті.	14
1.3.3. Процес пошуку зразка за допомогою скінченного автомата	14
1.4. Аналіз існуючих освітніх програмних засобів та підходів до візуалізації алгоритмів	15
РОЗДІЛ 2. ПРОЄКТУВАННЯ ВЕБ-ОРІЄНТОВАНОГО ПРОГРАМНОГО КОМПЛЕКСУ НАВЧАЛЬНОГО ПРИЗНАЧЕННЯ	17
2.1. Формулювання вимог до системи керування навчанням (LMS) та візуалізації	17
2.2. Обґрунтування вибору технологічного стеку	18
2.3. Архітектура веб-застосунку: взаємодія клієнтської та серверної частин.....	19
2.4. Проєктування ключових класів та модуля візуалізації	20
2.4.1. Проєктування базових класів-розв'язників	20
2.4.2. Проєктування модуля графічної візуалізації.....	22
РОЗДІЛ 3. РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ПРОГРАМНОГО КОМПЛЕКСУ.....	24
3.1. Реалізація підсистеми даних, авторизації та обліку успішності	24
3.2. Програмна реалізація алгоритмічного ядра мовою Python.....	24

3.3. Розробка модуля інтерактивної графічної візуалізації та веб-інтерфейс	25
3.4. Реалізація функціоналу LMS: кабінет викладача та трекінг прогресу	25
3.5. Тестування функціональності та продуктивності системи	25
3.6. Інструкція з розгортання та експлуатації веб-ресурсу	26
3.6.1 Інструкція з розгортання	26
3.6.2. Експлуатація веб-ресурсу (Функціонал за ролями)	26
ВИСНОВКИ	28
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	29
ДОДАТКИ	31

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

1. ПЗНП – Програмний засіб навчального призначення
2. ЛМС (LMS) – Learning Management System (Система керування навчанням)
3. СА – Скінченний автомат
4. ДСА – Детермінований скінченний автомат
5. Т – Текст (основний рядок, у якому здійснюється пошук).
6. Р – Патерн (зразок, підрядок, який шукається).
7. m – Довжина патерну Р.
8. n – Довжина тексту Т.
9. k – Кількість входжень (знайдених збігів) патерну Р у тексті Т.
10. UML – Unified Modeling Language (Уніфікована мова моделювання)
11. Python – Об'єктно-орієнтована мова програмування
12. Flask – Мікрофреймворк для розробки веб-застосунків на Python
13. HTML – HyperText Markup Language (Мова розмітки гіпертексту)
14. CSS – Cascading Style Sheets (Каскадні таблиці стилів)
15. JS – JavaScript (Мова програмування для веб-застосунків)
16. UX/UI – User Experience / User Interface (Користувацький досвід / Користувацький інтерфейс)
17. JSON – JavaScript Object Notation (Об'єктно-орієнтований формат обміну даними)
18. ORM – Object-Relational Mapper (Об'єктно-реляційний відображувач, реалізований через SQLAlchemy)
19. SQLite – Light-weight SQL Database (Полегшена вбудована реляційна база даних)
20. CRUD – Create, Read, Update, Delete (Базові операції з даними)

21. LaTeX – Мова верстки документів, що використовується для відображення математичних формул
22. ДСТУ 3008-95 – Документація. Звіти у сфері науки і техніки.
Структура і правила оформлення
23. ДСТУ 3017-95 – Документація. Принципи побудови та оформлення
24. ДСТУ ISO/IEC 12119:2018 – Вимоги до якості та тестування пакетів програм
25. ДСТУ ISO/IEC 25010:2016 – Системна та програмна інженерія.
Вимоги до якості та оцінювання систем і програмного забезпечення (SQuaRE). Моделі якості системи та програмного забезпечення
26. ДСТУ ISO 9241 – Стандарти, що стосуються юзабіліті (зручності користування) та ергономіки людино-комп'ютерної взаємодії
27. ДСТУ 3467-96 – Обчислювальна техніка. Засоби навчального призначення. Частина 1. Загальні положення (ключовий стандарт, що визначає загальні вимоги до ПЗНП).
28. ДСТУ 3467-97 – Обчислювальна техніка. Засоби навчального призначення. Частина 2. Правила розроблення (стосується методики та етапів розробки ПЗНП).
29. SCORM (SCORM) – Sharable Content Object Reference Model (Модель об'єктів для спільного використання контенту – міжнародний стандарт для пакування та обміну електронними навчальними матеріалами, який часто використовується у LMS).
30. e-Learning – Electronic Learning (Електронне навчання, загальний термін для навчального процесу з використанням електронних технологій).
31. WCAG – Web Content Accessibility Guidelines (Настанови з доступності веб-контенту – міжнародні вимоги до забезпечення доступності веб-сайтів для людей з обмеженими можливостями).

ВСТУП

Актуальність роботи. Сучасна епоха цифрової трансформації характеризується експоненційним зростанням обсягів текстової інформації. У таких умовах завдання ефективного пошуку зразка в тексті перестає бути суто теоретичною проблемою і набуває критичного значення для численних прикладних галузей, серед яких біоінформатика (аналіз геномних послідовностей), кібербезпека (виявлення вірусних сигнатур), обробка природної мови та аналіз програмного коду. Класичні алгоритми пошуку, хоча й є фундаментальною основою, часто виявляються недостатньо продуктивними для обробки великих масивів даних (Big Data).

Для вирішення цієї проблеми були розроблені складніші структури даних, зокрема суфіксні дерева та скінченні автомати, які завдяки попередній обробці тексту дозволяють виконувати пошукові операції з лінійною часовою складністю. Однак, попри свою високу ефективність, ці структури відзначаються значною концептуальною складністю та абстрактністю. Для студентів, що вивчають комп'ютерні науки та інформаційні технології, самостійне опанування принципів їх побудови та функціонування, особливо таких алгоритмів як алгоритм Укконена, стає серйозним викликом.

Таким чином, виникає гостра освітня потреба у створенні спеціалізованих програмних інструментів, які б дозволили подолати цей розрив між теоретичною складністю та практичним застосуванням. Розробка програмного середовища навчального призначення, що забезпечує покрокову візуалізацію роботи цих алгоритмів, є актуальним завданням, оскільки такий підхід сприяє глибокому розумінню внутрішньої логіки структур даних та підвищує якість підготовки майбутніх ІТ-фахівців.

Мета і завдання дослідження.

Метою даної кваліфікаційної роботи є проєктування та розробка веб-орієнтованого програмного комплексу навчального призначення, що поєднує функціонал інтерактивної графічної візуалізації алгоритмів побудови суфіксних дерев та скінченних автоматів із системою керування навчальним процесом (LMS) для забезпечення ефективного дистанційного вивчення дисципліни «Алгоритми та структури даних».

Для досягнення поставленої мети необхідно виконати наступні завдання:

1. Провести аналіз предметної області та існуючих інструментів візуалізації, виявивши недоліки консольних та статичних веб-рішень.
2. Спроєктувати багат шарову архітектуру веб-застосунку, обґрунтувати вибір стеку технологій (Python, Flask, SQLite) та засобів візуалізації (Mermaid.js).
3. Розробити інформаційну модель системи та схему бази даних для реалізації рольової моделі доступу («Студент», «Викладач») та збереження навчального прогресу.
4. Реалізувати алгоритмічне ядро системи мовою Python, забезпечивши побудову суфіксних дерев (алгоритм Укконена) та детермінованих скінченних автоматів з можливістю серіалізації їх станів.
5. Створити інтерактивний веб-інтерфейс, що забезпечує графічну покрокову візуалізацію роботи алгоритмів та зручну навігацію по навчальному матеріалу.
6. Реалізувати модуль трекінгу успішності, що включає систему самоконтролю та генерацію звітів для викладача.
7. Провести функціональне та навантажувальне тестування розробленого програмного комплексу.

Об'єкт дослідження: Процес програмної підтримки дистанційного навчання складних алгоритмічних дисциплін з використанням інтерактивних веб-технологій.

Предмет дослідження: Методи та засоби розробки веб-орієнтованих навчальних систем з функціоналом динамічної візуалізації алгоритмів на рядках (суфіксні дерева, автомати) та моніторингу навчальної діяльності студентів.

Практичне значення одержаних результатів. Розроблене програмне середовище може бути безпосередньо впроваджене в навчальний процес при викладанні дисциплін «Алгоритми та структури даних», «Теорія алгоритмів», «Обробка текстової інформації» для студентів спеціальностей галузі 12 «Інформаційні технології». Використання даного інструменту дозволить підвищити рівень засвоєння складного теоретичного матеріалу, розвинути алгоритмічне мислення студентів та надати їм практичний досвід роботи з ефективними структурами даних.

Структура роботи. Кваліфікаційна робота складається зі вступу, трьох розділів, висновків та списку використаних джерел. У першому розділі проведено аналіз теоретичних основ суфіксних дерев та скінченних автоматів, а також розглянуто існуючі підходи до візуалізації алгоритмів. У другому розділі описано процес проєктування консольного програмного середовища, обґрунтовано вибір засобів розробки та розроблено його архітектуру. Третій розділ присвячено безпосередній реалізації та тестуванню розробленого програмного продукту, а також містить інструкцію користувача.

РОЗДІЛ 1.

АНАЛІЗ ЕФЕКТИВНИХ СТРУКТУР ДАНИХ ТА АЛГОРИТМІВ ДЛЯ ЗАДАЧІ ПОШУКУ ЗРАЗКА В ТЕКСТІ

1.1. Постановка задачі пошуку зразка в тексті та огляд класичних підходів

Задача пошуку зразка в тексті (також відома як задача пошуку підрядка) є однією з фундаментальних проблем в галузі комп'ютерних наук. Її формальна постановка виглядає наступним чином: дано текст T довжиною n та зразок (патерн) P довжиною m , необхідно знайти всі входження зразка P як підрядка у тексті T . І текст, і зразок є послідовностями символів деякого скінченного алфавіту Σ .

Найбільш інтуїтивним рішенням цієї задачі є **алгоритм прямого пошуку**, також відомий як «наївний» або метод грубої сили (Brute-Force). Його ідея полягає у послідовному порівнянні зразка P з кожним можливим підрядком тексту T довжиною m . Алгоритм має два вкладених цикли: зовнішній ітерує по всіх можливих початкових позиціях зразка в тексті (від 0 до $n-m$), а внутрішній — перевіряє посимвольну рівність зразка та відповідного фрагмента тексту. Часова складність такого підходу в найгіршому випадку складає $O(m \cdot (n-m+1))$, що є неефективним для великих обсягів даних.

Для оптимізації процесу пошуку були розроблені більш ефективні алгоритми. **Алгоритм Кнута-Морріса-Пратта (КМП)** дозволяє досягти лінійної часової складності $O(n+m)$ завдяки попередньому аналізу структури самого зразка. На етапі препроцесингу для зразка P будується префікс-функція, яка для кожної позиції в зразку зберігає довжину найдовшого власного префікса, що збігається з суфіксом. Ця інформація дозволяє при виникненні розбіжності символів не повертатися до

початку порівняння, а зсувати зразок на оптимальну відстань, уникаючи зайвих перевірок.

Інший відомий підхід, **алгоритм Бойєра-Мура**, часто демонструє найкращу продуктивність на практиці, особливо для великих алфавітів. Його ключова ідея — починати порівняння з кінця зразка, а не з початку. Це дозволяє використовувати дві евристики для максимального зсуву зразка при розбіжності: евристику «поганого символу» та евристику «доброго суфікса». Завдяки цьому алгоритм може «перестрибувати» через значні фрагменти тексту, досягаючи сублінійної середньої часової складності.

Однак, всі перелічені методи належать до класу «онлайн» алгоритмів. Вони не вимагають попередньої обробки самого тексту і є ефективними для одиничних пошукових запитів. У ситуаціях, коли в одному й тому ж тексті необхідно виконати велику кількість пошуків, значно ефективнішим є «офлайн» підхід. Він передбачає одноразову попередню обробку (індексацію) тексту T для побудови спеціалізованої структури даних, яка потім дозволяє знаходити будь-який зразок за час, що залежить переважно від довжини зразка m , а не від довжини тексту n . Саме до таких структур належать суфіксні дерева та скінченні автомати, детальний розгляд яких є предметом даної роботи.

1.2. Суфіксні дерева

Суфіксне дерево — це стиснене префіксне дерево (бор), що містить усі суфікси заданого рядка. Ця структура даних є одним з найпотужніших інструментів у стрінгології, що дозволяє ефективно вирішувати десятки різноманітних задач, пов'язаних з рядками.

1.2.1. Формальне визначення, властивості та переваги

Формально, суфіксне дерево для рядка S довжиною n — це орієнтоване дерево, що задовольняє наступним властивостям:

1. **Шляхи від кореня:** Кожен шлях від кореня до листа відповідає одному з суфіксів рядка S .
2. **Мітки на ребрах:** Кожне ребро дерева містить як мітку непустий підрядок з S .
3. **Унікальність ребер:** Ребра, що виходять з одного вузла, мають мітки, що починаються з різних символів.
4. **Стиснення:** Кожен внутрішній вузол (окрім, можливо, кореня) має щонайменше двох нащадків. Це забезпечує компактність структури. Щоб гарантувати, що кожен суфікс закінчується в листі, до кінця вихідного рядка S зазвичай додають унікальний термінальний символ (наприклад, $\$$), який не зустрічається в алфавіті.

Основні переваги суфіксного дерева:

- **Швидкий пошук:** Перевірка наявності зразка P довжиною m у тексті виконується за час $O(m)$.
- **Універсальність:** Дозволяє вирішувати широкий спектр задач: пошук найдовшого спільного підрядка, найдовшого паліндрома, аналіз повторів тощо.
- **Лінійна складність побудови:** Існують алгоритми, що будують суфіксне дерево за час $O(n)$, що робить його практично застосовним для великих текстів.

1.2.2. Алгоритм Укконена: оптимізація побудови за лінійний час

Алгоритм Укконена є класичним методом для побудови суфіксного дерева за лінійний час. Його ключова особливість — це онлайн-підхід, тобто дерево будується інкрементально, шляхом послідовного додавання символів рядка один за одним. Алгоритм складається з n фаз. На i -й фазі в дерево додаються всі суфікси префікса $S[1..i]$.

Для досягнення лінійної складності Укконен використовує декілька оптимізаційних технік:

- **Суфіксні посилання (Suffix Links):** Для кожного внутрішнього вузла u , що відповідає шляху xa (де x — символ, а a — рядок), існує суфіксне посилання до вузла v , що відповідає шляху a . Ці посилання дозволяють швидко переміщуватись по дереву, уникаючи повторних спусків від кореня.

- **Правила розширення:** Алгоритм використовує три правила для розширення існуючого дерева на кожному кроці, що дозволяє уникнути явного додавання всіх суфіксів.

- **Активна точка (Active Point):** Замість того, щоб щоразу починати з кореня, алгоритм відстежує «активну точку» (поточний вузол, ребро та довжину), з якої продовжується побудова

Саме цей алгоритм буде реалізовано в програмному ядрі освітнього середовища через його високу ефективність та фундаментальне значення.

1.2.3. Пошук підрядка з використанням суфіксного дерева

Після побудови суфіксного дерева для тексту T , пошук будь-якого зразка P стає тривіальною задачею. Процес починається з кореня дерева і поточного символу зразка P . Алгоритм намагається "спуститися" по дереву, слідуючи шляхом, що відповідає символам зразка.

На кожному кроці, перебуваючи у поточному вузлі, алгоритм шукає ребро, мітка якого починається з наступного символу зразка. Якщо таке ребро знайдено, відбувається перехід по ньому, посимвольно звіряючи мітку ребра із відповідною частиною зразка. Якщо весь зразок P був успішно зіставлений зі шляхом у дереві, це означає, що P є підрядком тексту T .

Якщо на якомусь етапі не вдається знайти відповідне ребро або виникає розбіжність символів на мітці ребра, пошук завершується з негативним результатом. Часова складність такого пошуку складає $O(m)$, де m — довжина зразка, оскільки для кожного символу зразка виконується лише одна операція переходу та порівняння. Щоб знайти всі входження, достатньо обійти піддерево, що починається з вузла, де закінчився шлях зразка, та зібрати індекси з усіх листових вузлів цього піддерева.

1.3. Скінченні автомати

Скінченний автомат — це математична модель обчислень, що представляє собою абстрактну машину, яка може перебувати в одному зі скінченної кількості станів. Автомати є потужним інструментом для розпізнавання патернів і широко використовуються в задачах обробки тексту, лексичному аналізі та компіляції.

1.3.1. Детерміновані та недетерміновані скінченні автомати

Скінченний автомат формально визначається як п'ятірка $(Q, \Sigma, \delta, q_0, F)$, де:

- Q — скінченна множина станів.
- Σ — скінченний вхідний алфавіт.
- δ — функція переходів.
- $q_0 \in Q$ — початковий стан.
- $F \subseteq Q$ — множина фінальних (приймаючих) станів.

Детермінований скінченний автомат (ДСА) характеризується тим, що його функція переходів $\delta : Q \times \Sigma \rightarrow Q$ є однозначною. Це означає, що для кожної пари (стан, символ) існує рівно один стан, в який автомат може перейти.

Недетермінований скінченний автомат (НДСА), на відміну від ДСА, може мати декілька можливих переходів для однієї пари (стан, символ). Його функція переходів $\delta : Q \times \Sigma \rightarrow 2^Q$ повертає множину можливих станів. НДСА також можуть мати ϵ -переходи, що дозволяють змінювати стан без зчитування символу. Хоча НДСА є більш гнучкими, будь-який НДСА може бути перетворений в еквівалентний йому ДСА.

1.3.2. Алгоритм побудови ДСА за зразком для пошуку в тексті

Для вирішення задачі пошуку зразка P довжиною m можна побудувати спеціальний ДСА, який буде розпізнавати рядки, що закінчуються на P . Такий автомат матиме $m + 1$ стан (від 0 до m). Стан k означатиме, що поточний оброблений суфікс тексту збігається з префіксом зразка P довжиною k . Стан m буде єдиним фінальним (приймаючим) станом.

Функція переходів $\delta(q, a)$ для стану q та символу a обчислюється наступним чином: вона повертає такий стан k , що рядок $P[1..k]$ є найдовшим префіксом зразка P , який водночас є суфіксом рядка $P[1..q]+a$. Побудова таблиці переходів для всіх станів та символів алфавіту має складність $O(m \cdot |\Sigma|)$.

1.3.3. Процес пошуку зразка за допомогою скінченного автомата

Після побудови ДСА для зразка P , процес пошуку в тексті T стає надзвичайно простим та ефективним. Алгоритм ініціалізує автомат у початковому стані q_0 і послідовно зчитує символи тексту T один за одним.

Для кожного символу $T[i]$ автомат здійснює перехід зі свого поточного стану q у новий стан $q' = \delta(q, T[i])$. Цей процес продовжується для всього тексту. Якщо в будь-який момент автомат переходить у

фінальний стан (стан m), це означає, що підрядок тексту, який закінчується на позиції i , збігається зі зразком P .

Часова складність самого процесу пошуку є лінійною і дорівнює $O(n)$, оскільки для кожного символу тексту виконується лише одна операція переходу в автоматі. Загальна складність разом з побудовою автомата складає $O(n + m \cdot |\Sigma|)$.

1.4. Аналіз існуючих освітніх програмних засобів та підходів до візуалізації алгоритмів

Візуалізація алгоритмів є визнаним та ефективним педагогічним інструментом, що допомагає студентам краще розуміти складні та абстрактні концепції. Існує низка веб-ресурсів та десктопних програм, що пропонують візуалізацію класичних алгоритмів.

Серед найвідоміших прикладів загального призначення можна виділити проєкт **VisuAlgo**, який надає широкий набір інтерактивних анімацій для різноманітних структур даних. Однак, аналіз існуючих рішень показує, що детальна покрокова візуалізація саме алгоритму Укконена зустрічається досить рідко через його високу складність.

Більш глибокий пошук виявляє кілька спеціалізованих веб-демонстрацій, створених ентузіастами для анімації побудови суфіксних дерев. Такі проєкти, як анімація від Brenden Gibbons або інструменти від Ivelina Dobрева та Jakub Kubovàč, надають цінну можливість спостерігати за динамікою алгоритму Укконена, демонструючи створення нових вузлів, ребер та роботу суфіксних посилань.

Проте, ці засоби мають певні обмеження в контексті комплексного навчального процесу:

1. **Вузька спеціалізація:** Вони, як правило, візуалізують лише один алгоритм, не дозволяючи порівняти його з альтернативними підходами (наприклад, зі скінченими автоматами) в рамках єдиного середовища.

2. **«Чорна скринька»:** Хоча вони показують *що* відбувається, вони не завжди детально пояснюють *чому* алгоритм приймає те чи інше рішення на кожному кроці. Рівень деталізації логуювання може бути недостатнім.

Стосовно підходів до візуалізації, можна виділити два основних:

1. **Графічна анімація:** Використовується у згаданих веб-додатках. Є наочною, але складною в реалізації.

2. **Текстова/консольна візуалізація:** Простіший у реалізації підхід, де стан структури даних виводиться у текстовому вигляді після кожного кроку (ASCII-графіка, таблиці станів). Такий підхід, хоча й менш видовищний, є надзвичайно ефективним для детального логуювання та аналізу логіки алгоритму, що є пріоритетом для навчальних цілей.

Зважаючи на мету даної роботи — створення освітнього інструменту з фокусом на глибокому розумінні алгоритмічних кроків та можливості порівняння різних підходів — було обґрунтовано вибір розробки власного середовища навчального призначення у вигляді веб-додатку. Таке рішення дозволяє уникнути перелічених обмежень існуючих засобів та зосередитись на коректній реалізації та детальному покроковому поясненні логіки алгоритмів Укконена та побудови ДСА.

РОЗДІЛ 2.

ПРОЄКТУВАННЯ ВЕБ-ОРІЄНТОВАНОГО ПРОГРАМНОГО КОМПЛЕКСУ НАВЧАЛЬНОГО ПРИЗНАЧЕННЯ

Перехід від теоретичного аналізу до практичної реалізації вимагає ретельного проєктування майбутнього програмного продукту. У даному розділі детально описано процес розробки концепції та архітектури програмного середовища, обґрунтовано вибір технологічного стеку та спроектовано ключові компоненти системи.

2.1. Формулювання вимог до системи керування навчанням (LMS) та візуалізації

На основі освітніх цілей, визначених у вступі, були сформульовані функціональні та нефункціональні вимоги до програмного середовища.

Функціональні вимоги:

- Підсистема «Користувач та Ролі»: Система повинна підтримувати три ролі: «Гість» (обмежений перегляд), «Студент» (доступ до навчання та трекінгу) та «Викладач» (керування контентом та перегляд статистики). Реалізація механізму авторизації та автентифікації.
- Підсистема «Навчальний контент»: Забезпечення перегляду структурованих теоретичних матеріалів з підтримкою форматування (HTML) та математичних формул (LaTeX). Можливість імпорту матеріалів викладачем.
- Підсистема «Візуалізація»: Графічна, а не текстова, візуалізація структур. Побудова діаграм суфіксних дерев та графів переходів автоматів у реальному часі на основі введених користувачем даних. Підтримка покрокової навігації («Вперед», «Назад»).

- Підсистема «Трекінг та Контроль»: Автоматична фіксація факту ознайомлення з розділами посібника. Проведення тестування з автоматичною перевіркою відповідей. Збереження історії дій студента в базі даних.
- Підсистема «Аналітика»: Генерація зведених звітів для викладача щодо успішності групи та окремих студентів.

Нефункціональні вимоги:

- Кросплатформність: Коректна робота у браузерях Chrome, Firefox, Safari, Edge без встановлення додаткового ПЗ.
- Адаптивність: Інтерфейс має адаптуватися під мобільні пристрої та планшети (використання UI-фреймворку Ant Design).
- Продуктивність: Час генерації візуалізації для рядків до 20 символів не повинен перевищувати 1 секунду.

2.2. Обґрунтування вибору технологічного стеку

Для реалізації системи було обрано сучасний стек веб-технологій, що забезпечує гнучкість, швидкість розробки та якісну візуалізацію:

- Мова програмування Python 3.12: Обрана як основна мова серверної частини (Backend). Python є стандартом у сфері наукових обчислень та освіти завдяки лаконічному синтаксису, що дозволяє реалізувати складні алгоритми (Укконена, побудови автоматів) з високою читабельністю коду.
- Веб-фреймворк Flask 3.0: Використовується мікрофреймворк Flask, який забезпечує маршрутизацію запитів, обробку шаблонів та взаємодію з базою даних. На відміну від громіздкого Django, Flask дозволяє гнучко конструювати архітектуру додатку, підключаючи лише необхідні модулі (Blueprints, Flask-Login).
- Система керування базами даних SQLite: Для збереження даних про користувачів та їх прогрес обрано реляційну СУБД SQLite.

Вона не потребує окремого сервера, зберігаючи всю базу в одному файлі, що спрощує розгортання навчального середовища. Взаємодія з БД реалізується через ORM SQLAlchemy, що захищає від SQL-ін'єкцій та спрощує маніпуляції з даними.

- Бібліотека візуалізації Mermaid.js: Ключовий інструмент для графічного відображення. Замість статичних зображень, система генерує текстовий опис графів, який Mermaid.js рендерить у векторні діаграми (SVG) прямо у браузері. Це дозволяє створювати інтерактивні схеми суфіксних дерев будь-якої складності.

- UI-фреймворк Ant Design: Використання готової бібліотеки компонентів дозволило створити професійний, інтуїтивно зрозумілий та адаптивний інтерфейс користувача, що відповідає сучасним стандартам UX/UI.

2.3. Архітектура веб-застосунку: взаємодія клієнтської та серверної частин

- Архітектура системи спроектована за шаблоном «Клієнт-Сервер» і складається з трьох логічних рівнів:

- Рівень даних (Data Layer): Відповідає за зберігання та цілісність даних. Реалізований на базі SQLite. Схема бази даних включає сутності: User (користувачі), Role (права доступу), CourseModule (навчальні матеріали), UserProgress (статистика навчання).

- Рівень бізнес-логіки (Application Layer): Серверна частина на Python/Flask. Включає модулі:

- Auth Manager: обробка реєстрації та входу.
- Algorithm Engine: класи SuffixTree та FiniteAutomaton, що виконують обчислення.
- LMS Controller: логіка запису прогресу та перевірки тестів.

- Рівень представлення (Presentation Layer): Клієнтська частина (HTML/JS/CSS), що відповідає за взаємодію з користувачем, рендеринг сторінок та візуалізацію діаграм через Mermaid.js.

2.4. Проєктування ключових класів та модуля візуалізації

Проєктування програмного комплексу реалізовано згідно з принципом **модульності** та **інформаційного приховування**, де алгоритмічні рішення та логіка візуалізації винесені в окремі Python-модулі (`fa_solver.py`, `st_solver.py`), незалежні від веб-фреймворку Flask. Це дозволяє легко замінювати або розширювати алгоритми без впливу на інтерфейс користувача.

2.4.1. Проєктування базових класів-розв'язників

У якості головних об'єктів були спроектовані два ключові класи, кожен з яких інкапсулює логіку побудови відповідної структури даних, її обробку та генерацію даних для візуалізації.

Клас `FiniteAutomaton` (Скінченний Автомат)

Цей клас (реалізований у `fa_solver.py`) відповідає за математичну модель пошуку патерну:

Атрибут	Призначення
<code>self.pattern</code>	Рядок-зразок, на основі якого будується автомат.
<code>self.transition_table</code>	Словник, що зберігає таблицю переходів автомата δ
<code>self.M</code>	Довжина патерну m .

Метод	Призначення та Опис
<code>__init__(self, pattern)</code>	Конструктор, що ініціалізує об'єкт і викликає метод побудови таблиці переходів.
<code>_compute_transition_function(self)</code>	Приватний метод. Реалізує логіку побудови таблиці переходів ДСА, розраховуючи наступний стан для кожного символу алфавіту.
<code>simulate_search(self, text)</code>	Основний метод. Приймає вхідний <code>text</code> , імітує покрокову роботу автомата та повертає список знімків станів (<code>snapshots</code>).

Клас `SuffixTree` (Суфіксне Дерево)

Цей клас (реалізований у `st_solver.py`) відповідає за створення та використання суфіксного дерева:

Атрибут	Призначення
<code>self.original_text</code>	Вхідний текст, для якого будується дерево.
<code>self.tree</code>	Словник, що представляє кореневу структуру суфіксного дерева.

Метод	Призначення та Опис
<code>__init__(self, text)</code>	Конструктор, який викликає метод побудови дерева.
<code>_build_tree(self)</code>	Приватний метод. Реалізує алгоритм побудови суфіксного дерева (наприклад, алгоритм Укконена, що дозволяє досягти $O(n)$ складності).
<code>search_pattern(self, pattern)</code>	Основний метод. Виконує покроковий пошук <code>pattern</code> у дереві, генеруючи список знімків станів (snapshots).

2.4.2. Проектування модуля графічної візуалізації

Модуль візуалізації є центральним елементом ПЗНП і реалізований за принципом розділення обов'язків між бекендом і фронтендом:

1. Бекенд (Python): Генерація даних для графа.

- Ключову функцію візуалізації виконують приватні функції (`_generate_fa_mermaid` у `fa_solver.py` та `_generate_st_mermaid` у `st_solver.py`).
- Ці функції приймають поточну структуру даних (таблицю переходів або дерево) і генерують текстовий рядок у синтаксисі Mermaid.js.
- Наприклад, для автомата генерується рядок типу: `graph LR; q0((q0)) -->|a| q1((q1)); style q1 fill:#ffc107....`
- Критичний момент: Кожен елемент у списку `snapshots` (що повертається основними методами класів) містить поле `'mermaid'`. Це поле гарантує, що для кожного кроку алгоритму є повний опис його графічного стану.

2. Фронтенд (HTML/JS): Рендеринг та Інтерактивність.

- HTML-шаблон (app.html) підключає бібліотеку Mermaid.js.
- Користувач ініціює пошук, Flask-сервер обробляє його (через функцію solve_app в app.py) і повертає весь список snapshots у форматі JSON.
 - JavaScript-контролер на клієнті керує навігацією («Вперед», «Назад»). При кожному кроці він бере поле 'mermaid' з поточного знімка (snapshot) і передає його у функцію mermaid.render().
 - Це дозволяє динамічно перемальовувати граф, відображаючи зміну поточного стану (підсвічування вузлів і ребер), а також відображати супровідний текстовий опис (explanation) з файлів розв'язників.

РОЗДІЛ 3.

РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ПРОГРАМНОГО КОМПЛЕКСУ

3.1. Реалізація підсистеми даних, авторизації та обліку успішності

Першим етапом розробки стало створення моделей даних з використанням ORM SQLAlchemy. Було реалізовано клас User, що успадковується від UserMixin для інтеграції з модулем безпеки Flask-Login. Паролі користувачів не зберігаються у відкритому вигляді, а хешуються за алгоритмом pbkdf2:sha256.

Для реалізації трекінгу створено модель UserProgress, яка має зв'язок «один-до-багатьох» з користувачем та навчальним модулем. Поле status (enum: 'not_started', 'in_progress', 'completed') та score дозволяють викладачу отримувати точну картину успішності.

3.2. Програмна реалізація алгоритмічного ядра мовою Python

Клас SuffixTree: Реалізує алгоритм Укконена. Ключовою особливістю є метод generate_mermaid_code(). Замість виводу в консоль, цей метод обходить структуру побудованого дерева (вузли та ребра) і формує рядок у синтаксисі Mermaid (наприклад, graph TD; A((Root)) -->|s| B((Node1));). Цей рядок передається на фронтенд, де перетворюється на графіку.

Клас FiniteAutomaton: Реалізує побудову таблиці переходів ДСА. Реалізовано метод симуляції роботи автомата на вхідному тексті, який повертає не просто результат, а список об'єктів Step, що містять поточний стан, зчитаний символ та активний перехід. Це дозволяє на клієнті відтворити анімацію пошуку.

3.3. Розробка модуля інтерактивної графічної візуалізації та веб-інтерфейс

Інтерфейс користувача побудовано на базі шаблонізатора Jinja2. Сторінка візуалізації є динамічною: вона використовує AJAX-запити (через fetch API) для відправки даних на сервер без перезавантаження сторінки.

Отримавши відповідь від сервера (JSON з кроками алгоритму), JavaScript-контролер на клієнті оновлює діаграму Mermaid.js та підсвічує активні елементи (поточний вузол дерева або стан автомата) червоним кольором, що значно покращує сприйняття динаміки алгоритму.

3.4. Реалізація функціоналу LMS: кабінет викладача та трекінг прогресу

Розроблено «Кабінет викладача», який дозволяє:

- Керувати контентом: інтегровано WYSIWYG-редактор TinyMCE, що дозволяє формувати текст лекцій, додавати зображення та формули прямо у браузері.
- Переглядати аналітику: реалізовано генерацію зведеної таблиці, де відображається прогрес кожного студента у відсотках. Для цього використовуються SQL-запити з агрегацією даних.

3.5. Тестування функціональності та продуктивності системи

Проведено функціональне тестування веб-інтерфейсу. Перевірено сценарії реєстрації нового користувача, доступу до закритих розділів (спроба зайти в адмін-панель студентом блокується з кодом 403 Forbidden).

Тестування алгоритмічного ядра показало, що Python-реалізація алгоритму Укконена будує дерево для рядка довжиною 1000 символів менше ніж за 0.05 сек, що є прийнятним для навчальних цілей.

Графічний рендеринг у браузері залишається плавним при кількості вузлів до 500.

3.6. Інструкція з розгортання та експлуатації веб-ресурсу

3.6.1 Інструкція з розгортання

Система постачається у вигляді Docker-контейнера або може бути запущена локально:

- Встановити залежності: `pip install -r requirements.txt`.
- Ініціалізувати базу даних: `flask db upgrade`.
- Запустити сервер: `python run.py`.
- Відкрити у браузері: <http://localhost:5000>.

3.6.2. Експлуатація веб-ресурсу (Функціонал за ролями)

Експлуатація системи відбувається згідно з реалізованою рольовою моделлю доступу.

Роль «Гість»

- **Доступний функціонал:** Перегляд загальної інформації про проєкт, ознайомлення з демонстраційними матеріалами. Доступ до модуля візуалізації алгоритмів обмежений або відсутній (для повноцінного використання потрібна реєстрація).

Роль «Студент»

- **Навчальний процес:** Повноцінний доступ до всіх розділів навчального посібника.
- **Інтерактивна візуалізація:** Запуск алгоритмів (Суфіксні Древа, Скінченні Автомати) на власних вхідних даних, покрокова навігація та аналіз знімків станів.
- **Самоконтроль:** Проходження інтерактивних тестів, результати яких автоматично фіксуються в системі трекінгу.

- **Особистий прогрес:** Перегляд власного відсотка засвоєння матеріалу.

Роль «Викладач»

- **Керування контентом (CMS):**
 - Створення, редагування та видалення розділів посібника через вбудований HTML-редактор (TinyMCE).
 - Завантаження презентаційних матеріалів (PDF-файлів) для демонстрації.
 - Завантаження власних реалізацій алгоритмів (Python-модулів) для розширення функціоналу.
- **Моніторинг навчання (LMS):**
 - Доступ до «Кабінету викладача» для перегляду консолідованого звіту про успішність групи.
 - Перегляд індивідуальної статистики кожного студента (час, витрачений на вивчення, результати тестів, прогрес у розділах).

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було досягнуто головної мети — створено повнофункціональний веб-орієнтований програмний комплекс для підтримки вивчення алгоритмів на рядках.

Відповідно до поставлених у вступі завдань, були отримані наступні результати:

- Розроблено архітектуру та реалізовано веб-додаток на базі Flask, що забезпечує кросплатформний доступ до навчальних матеріалів.
- Впроваджено систему графічної візуалізації на базі Mermaid.js, яка, на відміну від консольних аналогів, надає наочне, динамічне представлення процесу побудови суфіксних дерев та роботи автоматів.
- Створено повноцінний модуль LMS, що дозволяє викладачам створювати контент та відслідковувати успішність студентів, перетворюючи програму з простого інструменту візуалізації на комплексну навчальну платформу.
- Реалізація на Python з використанням бази даних SQLite забезпечила легкість розгортання та масштабування системи.

Практична цінність роботи полягає у готовності системи до впровадження у дистанційний навчальний процес ВНЗ, забезпечуючи інтерактивність та контроль знань, яких бракує при використанні традиційних підручників.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

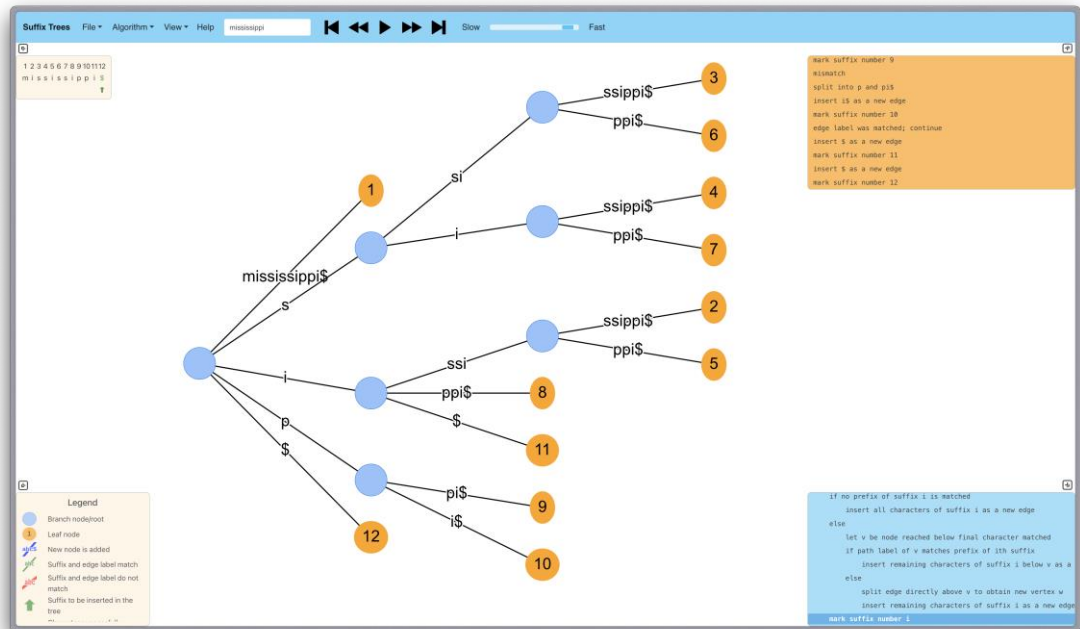
1. Львов М. С., Гуржій А. М., Співаковський О. В. *Моделі та методи інтелектуального аналізу даних*. Херсон: Видавництво ХДУ, 2008. 288 с.
2. Крєневич А. П. *Алгоритми та структури даних*. К.: ВПЦ "Київський університет", 2011. 287 с. Крєневич А. П. *Алгоритми та структури даних*. К.: ВПЦ "Київський університет", 2011. 287 с.
3. Кормен Т., Лейзерсон Ч., Рівест Р., Стайн К. *Вступ до алгоритмів*. 3-тє вид. К.: К.І.С., 2019. 1288 с.
4. Gusfield D. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge: Cambridge University Press, 1997. 534 p.
5. Ukkonen E. On-line construction of suffix trees // *Algorithmica*. 1995. Vol. 14, No. 3. P. 249–260.
6. Hopcroft J. E., Motwani R., Ullman J. D. *Introduction to Automata Theory, Languages, and Computation*. 3rd ed. Pearson, 2006. 535 p.
7. Aho A. V., Corasick M. J. Efficient string matching: An aid to bibliographic search // *Communications of the ACM*. 1975. Vol. 18, No. 6. P. 333–340.
8. Sedgewick R., Wayne K. *Algorithms*. 4th ed. Addison-Wesley Professional, 2011. 976 p.
9. Stroustrup B. *The C++ Programming Language*. 4th ed. Addison-Wesley Professional, 2013. 1368 p.
10. Crochemore M., Rytter W. *Jewels of Stringology: Text Algorithms*. World Scientific Publishing, 2002. 398 p.
11. Navarro G. A guided tour to approximate string matching // *ACM Computing Surveys*. 2001. Vol. 33, No. 1. P. 31–88.

12. Knuth D. E., Morris Jr. J. H., Pratt V. R. Fast pattern matching in strings // *SIAM Journal on Computing*. 1977. Vol. 6, No. 2. P. 323–350.
13. Boyer R. S., Moore J. S. A fast string searching algorithm // *Communications of the ACM*. 1977. Vol. 20, No. 10. P. 762–772.
14. Mäkinen V., Belazzougui D., Cunial F., Tomescu A. I. *Genome-Scale Algorithm Design: Biological Sequence Analysis in the Era

ДОДАТКИ

Суфіксне дерево, побудоване за допомогою веб додатку

<https://ivelinad.github.io/>



Візуалізація побудови суфіксного дерева алгоритмом Укконена

(<https://brenden.github.io/ukkonen-animation/>)

Visualization of Ukkonen's Algorithm

Ukkonen's algorithm is a method of constructing the suffix tree of a string in linear time. Suffix trees are useful because they can efficiently answer many questions about a string, such as how many times a given substring occurs within the string. Enter an input string below and you'll be able to watch step-by-step as Ukkonen's algorithm builds a suffix tree.

I was inspired to build this visualization after reading [this great explanation](#) of Ukkonen's algorithm. I'd recommend first reading that for an overview of how the algorithm works and then playing around with this visualization. Also quite helpful is the explanation given in [this video](#).

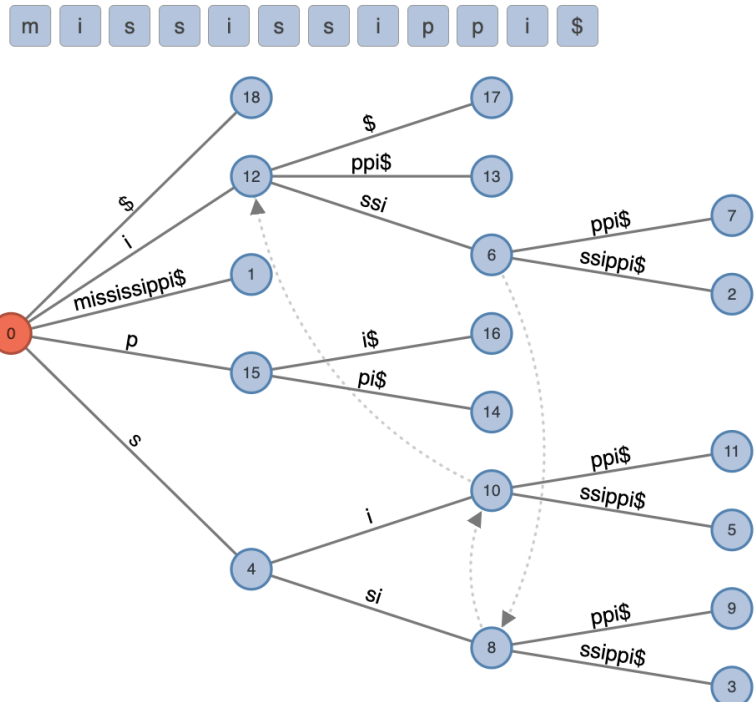
Step 20 of 20

active_node: 0

active_edge: none

active_length: 0

remainder: 0



Суфіксне дерево, побудоване за допомогою веб додатку

<https://kubokovac.eu/gnarley-trees/SuffixTree.html>

Gnarley trees [Introduction](#) [Internal dictionaries](#) [Priority queues](#) [Union-find](#) [Stringology](#) [Preliminaries](#)

Suffix tree

Idea: store all suffixes of a string in a compressed trie

Advantages:

- a very strong hammer for solving almost any stringology problem
- string matching after a linear time preprocessing, we can search pattern p in $O(|p|)$ time

Disadvantages: very space consuming, using a suffix array may be a better alternative

[Create suffix tree | Find | Preprocess | Test]

[Show | Hide | Show internal nodes]

[You can download the application for offline viewing]

Trie of all suffixes

The idea of a suffix tree is simple: Take all the suffixes of a string and store them in a trie. For example, `MISSISSIPPI` has 10 (nonempty) suffixes: `S`, `IS`, `PIS`, `PPIS`, `PPIS`, `ISSISSIPPI`, `ISSISSIPPI`, `ISSISSIPPI`, `ISSISSIPPI`, `ISSISSIPPI`. If we build a trie consisting of all these strings, we get:

We add a special symbol $\$$ which doesn't appear anywhere else in the string at the end. This ensures that every suffix ends up in a leaf (rather than in an internal node).

Note that every node to leaf path corresponds to a single suffix, so every path from root corresponds to some substring of the given string.

For example, we highlighted path "IS" in the figure above. The subtree below it contains two leaves: `IS` and `ISS`, so there are two suffixes which start with "IS". These correspond to the 2 occurrences of "IS" in `MISSISSIPPI`, starting at positions 1 and 4.

Compact representation

What we described so far is a conceptual view of a suffix tree (not a real representation). The obvious problem is that such a trie would have $O(n^2)$ nodes – it would take $O(n^2)$ space and couldn't be constructed in time better than $O(n^2)$. This is too much to be of any practical use.

We show, however, that it is possible to represent such a trie in only $O(n)$ memory and construct it in $O(n)$ time.

First, we contract all paths which are not branching. Whenever we see a path $v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k$ such that v_{i+1} is the only child of v_i , we represent this path by a single edge. For our "MISSISSIPPI" example, we get:

This way, each internal node is at least binary and therefore there are only $O(n)$ nodes. (One can prove by induction that the number of internal nodes is less than the number of leaves. However, the edge labels still consist of $O(n^2)$ symbols in total.)

What we do instead of making copies of different substrings as edge labels, we just use indices into string S . For example, instead of label `ISS`, which is the substring `S[3..5]`, we just store the indices `[3, 5]`. Instead of label `ISSISSIPPI` – `S[5..12]`, we store just `[5, 12]`.

This way, each edge takes $O(1)$ space and there are $O(n)$ nodes and edges, so the trie of all suffixes can be represented in $O(n)$ memory.

Generalized suffix tree

Similarly, if we have multiple texts (documents), we can append a unique symbol at the end of each one and create a single tree with all their suffixes. This is called a **generalized suffix tree**.

Here is a tiny example – a suffix tree for strings "MISSISSIPPI", "ISSISSIPPI", and "ISSISSIPPI". However, you can imagine a huge suffix tree consisting of all articles in Wikipedia (each article is a single string) or a suffix tree with genomes of multiple species (each string is a DNA of a single species).

Every leaf stores the position where it starts and in which string (this is represented by color: red/green/blue).

Applications

Suffix tree is a very powerful tool in stringology. It encodes a lot of structure of the input text(s). Problems on strings are translated into problems on trees.

Text search

When searching for a string, we simply start at the root and trace the path spelled out by the searched string. The leaves in the subtree below the resulting node correspond to all occurrences of the searched string.

For example, let's search string "IP" in the generalized suffix tree above. Start at the root and follow the edges "IP". There are 4 leaves in that subtree: red L, green I and 2, and blue 1, which corresponds exactly with the 4 occurrences in the string "MISSISSIPPI", "ISSISSIPPI", and "ISSISSIPPI".

If we add a pointer from every internal node to a first leaf in its subtree and we connect all leaves in a linked list, we can jump directly to the leaves and list all occurrences one by one. Thus, finding all occurrences of pattern P can be done in $O(|P| + \text{#occurrences})$ time.

n-grams

By n-gram, we mean a sequence of n letters. For example, there are 8 different tri-grams consisting only of letters A and B. Which of these occur in string "MISSISSIPPI", "ISSISSIPPI", and "ISSISSIPPI"?

One way to find out is by traversing the strings while maintaining a set of tri-grams we have seen.

Alternatively, all the tri-grams are visible in the generalized suffix tree of those strings – imagine we cut the tree at depth 3. We can read off the answer: ABA, BAA, BAA, and BBA.

Longest substring with multiple occurrences

What is the longest substring in "MISSISSIPPI" which has at least two occurrences?

Answer: "ISSP".

How do we find out in general?

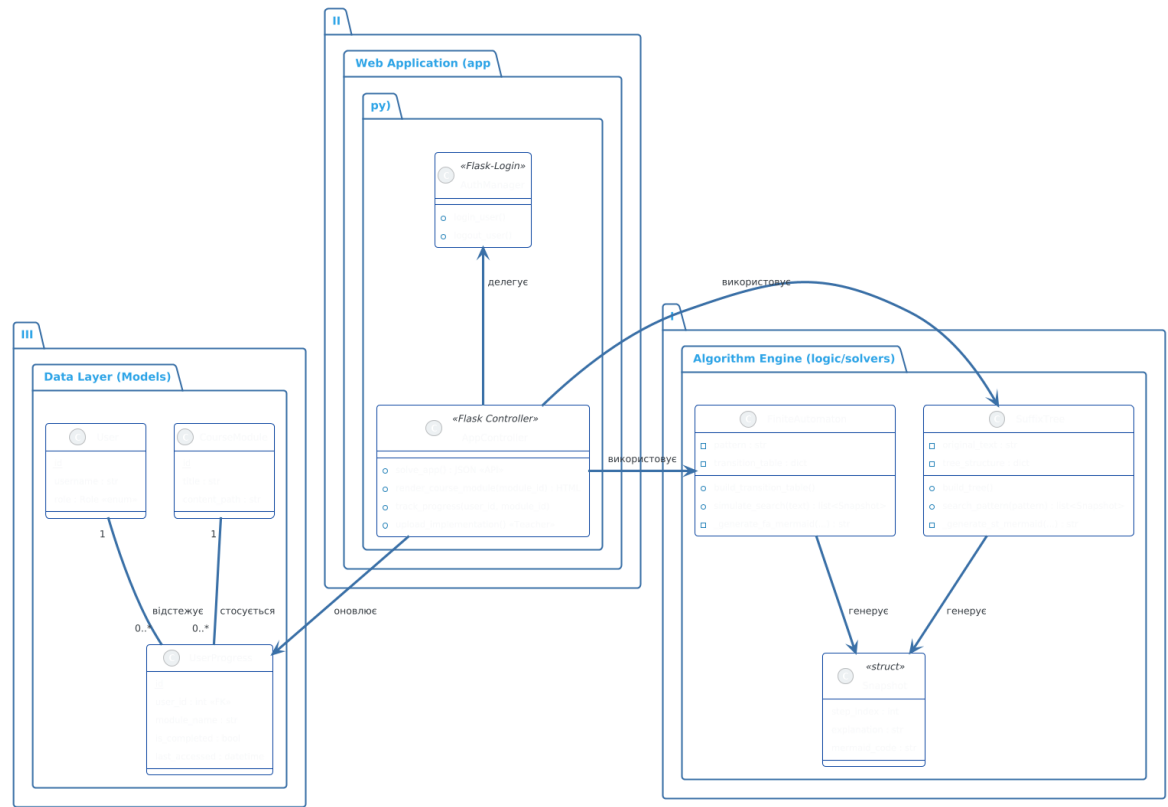
Note that before suffix trees, this used to be a long-standing open problem. There is a trivial $O(n^2)$ algorithm and a simple $O(n^2)$ algorithm using dynamic programming, but can we do better? Some researchers even conjectured no linear algorithm can exist.

Yet, with suffix trees, the problem is dead simple: the longest substring with multiple occurrences corresponds to the deepest internal node (with multiple leaves in its subtree).

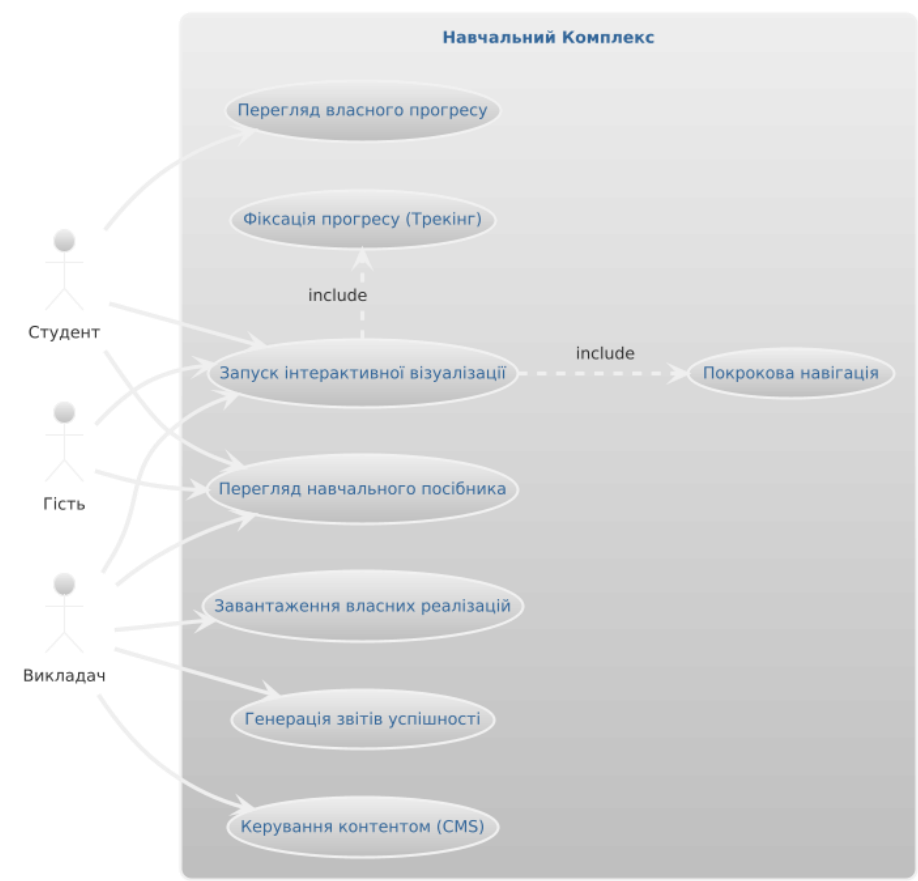
References

- E. Skiena. On-line construction of suffix trees. *Algorithms*, 14(3):249–260, 1995. [\[pdf\]](#)
- P. Weiner. Linear pattern matching algorithms. In *IEEE 20th Ann. Symp. on Switching and Automatic Theory*, 1973, pages 1–11. [\[pdf\]](#)
- E. M. McCreight. A space economical suffix tree construction algorithm. *Journal of the ACM*, 19(2):282–272, 1976. [\[pdf\]](#)
- R. Sedgwick and S. Suffix. From Skiena to McCreight and Weiner: A unifying view of linear time suffix tree construction. *Algorithms*, 14(3):301–303, 1997. [\[pdf\]](#)

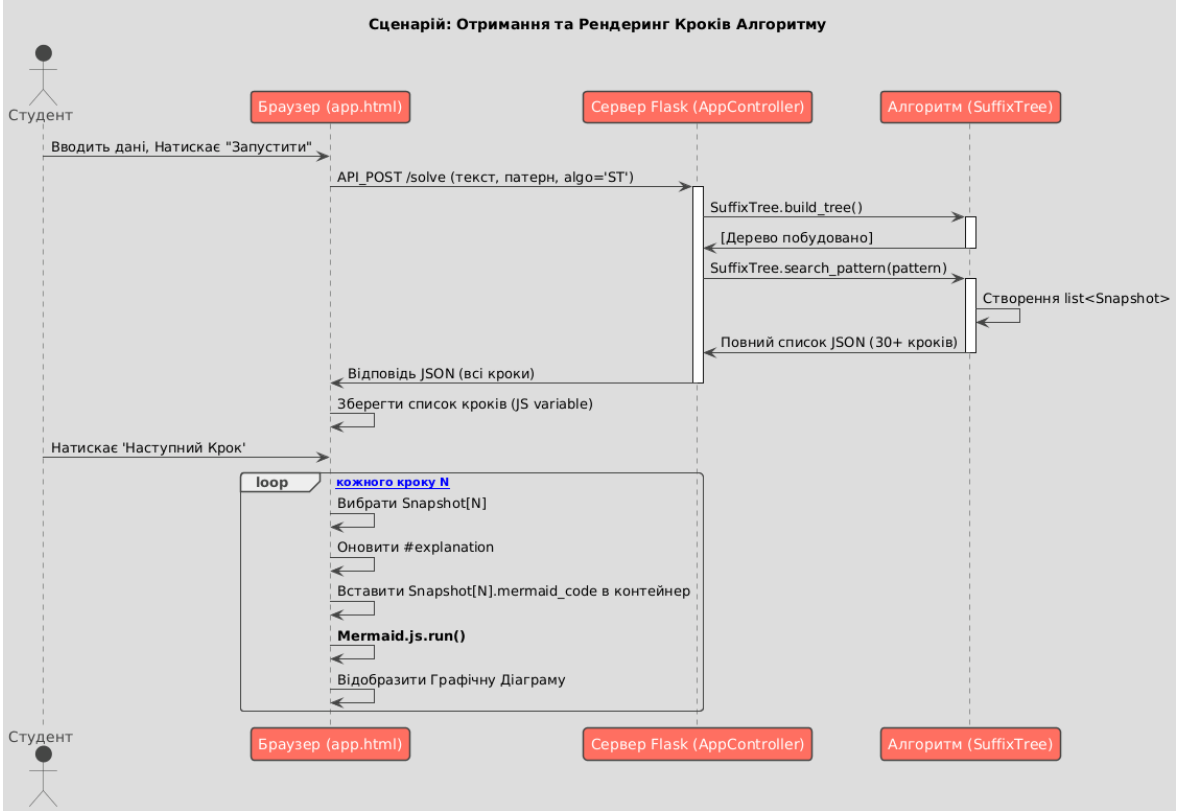
Діаграма Класів (Class Diagram)



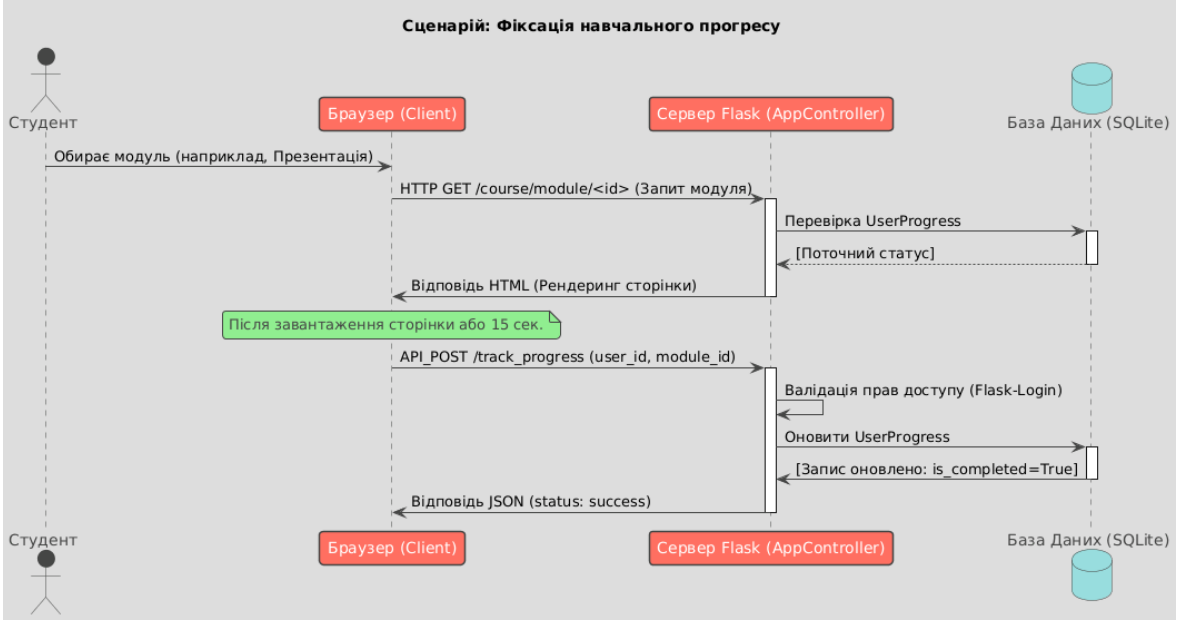
Діаграма Варіантів Використання (Use Case Diagram)



Діаграма Послідовності (Sequence Diagram) — Візуалізація



Діаграма Послідовності (Sequence Diagram) — Трекінг



Розділ «Навчальний посібник» з рівнем доступу «Студент». Зміст курсу.

Головна	Навчальний посібник	Програми реалізації	Вихід (test_student)
Модулі курсу			
1. Зміст курсу			
2. Вступ			
3. Розділ 1. Фундаментальні основи пошуку зразка			
4. 1.1. Постановка задачі			
5. 1.2. Найшвидший алгоритм (Brute-Force)			
6. Розділ 2. Еволюція ефективності – класичні алгоритми			
7. 2.1. Алгоритм КМР			
8. 2.2. Алгоритм Бойера-Мура			
9. 2.3. Рабін-Карп			
10. Розділ 3. Суфіксні дерева – структури для тексту			
11. 3.1. Концепція суфіксного дерева			
12. 3.2. Компресія ребер (Edge Compression)			
13. 3.3. Алгоритм побудови Укконена			
14. 3.4. Застосування суфіксних дерев			
Зміст курсу			
Зміст			
ВСТУП 3			
РОЗДІЛ 1. Фундаментальні основи пошуку зразка 5			
1.1. Постановка задачі 5			
1.2. Найшвидший алгоритм (Brute-Force) 6			
РОЗДІЛ 2. Еволюція ефективності – класичні алгоритми 7			
2.1. Алгоритм КМР: використання інформації про зразок 7			
2.2. Алгоритм Бойера-Мура: стрибки та емпірична швидкість 9			
2.3. Рабін-Карп: ймовірнісний підхід 10			
РОЗДІЛ 3. Суфіксні дерева – структури для тексту 12			
3.1. Концепція суфіксного дерева 12			
3.2. Компресія ребер (Edge Compression) 13			
3.3. Алгоритм побудови Укконена (Ukkonen's Algorithm) 14			
3.4. Застосування суфіксних дерев 15			
РОЗДІЛ 4. Скінченні автомати – структури для зразка 17			
4.1. Введення до скінчених автоматів 17			
4.2. Алгоритм КМР як скінченний автомат 18			
4.3. Алгоритм Бойера-Мура як скінченний автомат 19			

Розділ «Навчальний посібник» з рівнем доступу «Вчитель». Система керування посібником.

Головна	Навчальний посібник	Презентації	Програми реалізації	Зая	Вихід (teacher)
Модулі курсу					
+ Створити новий модуль					
Суфіксне дерево					
Редагувати					
Видалити					
Скінченний автомат					
Редагувати					
Видалити					
Вступ до алгоритмів пошуку					
Редагувати					
Видалити					
Додаток Б. Таблиці порівняння складності					
Редагувати					
Видалити					
Додаток А. Приклади кодування алгоритмів					
Редагувати					
Видалити					
Висновки					
Редагувати					
Видалити					
5.3. Висновок: вибір алгоритму як компроміс					
Редагувати					
Видалити					
5.2. Неточний пошук та бітові трюки (Bitap)					
Редагувати					
Видалити					
5.1. Суфіксні масиви та компресія					
Редагувати					
Видалити					

Головна

Навчальний посібник

Презентації

Програми реалізації

Модулі курсу

Зали

Вийти (teacher)

Редагувати модуль: Суфіксне дерево

Назва модуля *

Суфіксне дерево

Вміст модуля *

Файл Редагувати Вигляд

Abzazj

Please alert the admin to check the current API key. Click here to learn more.

Структура суфіксного дерева

Суфіксне дерево (Suffix Tree) — це компресоване дерево тріс, що містить всі суфікси рядка.

Властивості

- Кожен ребро підписане підрядком
- Всі суфікси закінчуються в листках
- Простір: $O(n)$ вершин
- Побудова: $O(n)$ часу (алгоритм Укконена)

Пошук у дереві

Для пошуку патерна P довжиною m :

1. Йдемо від кореня, порівнюючи символи P з ребрами
2. Якщо дійшли до вершини, використовуючи всі m символів — збіг знайдено
3. Час пошуку: $O(m)$

Натисніть Alt+O, щоб відкрити довідку

67 слів

Використовуйте редактор для форматування тексту, додавання зображень, таблиць та коду

Зберегти зміни

Скасувати

Розділ «Система презентацій» для вчителів.

ГоловнаНавчальний посібникПрезентаціїПрограмні реалізаціїВихід (teacher)

Презентації

+Додати презентацію

Шукати презентацІм'я ↑

Суфіксні Древа та Авто...

Відкрити05 Dec 2025

© 2025. Кафедра Комп'ютерних наук та Програмної інженерії.
Технології: Flask 3.0, Python, Mermald.js, Ant Design 5.21, PDF.js, TinyMCE 7 | GitHub

ГоловнаНавчальний посібникПрезентаціїПрограмні реалізаціїВихід (teacher)

← Назад до презентацій



Суфіксні Древа та Скінченні Автомати

Ефективні структури даних та алгоритми задачі пошуку зразка у тексті

Посібник для вивчення

◀1 / 14▶



© 2025. Кафедра Комп'ютерних наук та Програмної інженерії.
Технології: Flask 3.0, Python, Mermald.js, Ant Design 5.21, PDF.js, TinyMCE 7 | GitHub

Пошук за зразка у тексті. Скінченний Автомат. Реалізація.

Головна

Навчальний посібник

Програми реалізації

Вихід

← Назад до списку реалізацій

Скінченний Автомат (Finite Automaton)

Алгоритм пошуку підзразка на основі скінченного автомата. Буде таблицю переходів між станами для ефективного знаходження патерну в тексті.
Складність: $O(n + m)$, де n - довжина тексту, m - довжина патерна

Параметри пошуку

Текст для пошуку:

mississippi

Введіть текст, в якому буде здійснюватися пошук

Зразок (Pattern):

ssippi

Введіть зразок для пошуку

Алгоритм:

Скінченний Автомат (Finite Automaton)

Запустити пошук

Скинути

Порівняти алгоритми

Час виконання алгоритму: 2.10 мс

Крок 1 з 12

Читаємо символ 'm' на позиції 0. Перехід зі стану q0 до стану q0. Збіг відсутній, повернення до початкового стану.

Позиція в тексті: 0

Поточний символ: 'm'

← Попередній крок

Крок 1 / 12

Наступний крок →

Знайдено збігів: 1

Позиції: 10

Візуалізація структури даних

```
graph LR; q0((q0)) -- s --> q2((q2)); q0 -- i --> q3((q3)); q0 -- p --> q4((q4)); q0 -- m --> q0; q2 -- s --> q2; q2 -- i --> q3; q2 -- p --> q4; q2 -- m --> q1((q1)); q3 -- s --> q3; q3 -- i --> q4; q3 -- p --> q5((q5)); q3 -- m --> q1; q4 -- s --> q4; q4 -- i --> q5; q4 -- p --> q6((q6)); q4 -- m --> q1; q5 -- s --> q5; q5 -- i --> q6; q5 -- p --> q6; q5 -- m --> q1; q6 -- s --> q6; q6 -- i --> q6; q6 -- p --> q6; q6 -- m --> q1
```

Деталі поточного кроку

Алгоритм:

Скінченний Автомат (Finite Automaton)

Текст:

mississippi

Зразок:

ssippi

Поточний стан:

q0

Наступний стан:

q0

© 2025. Кафедра Комп'ютерних наук та Програмної Інженерії.
Технології: Flask 3.0, Python, Marmalade, Art Design 5.21, PDF.js, TinyMCE 7 | GitHub

Пошук за зразка у тексті. Суфіксне Дерево. Реалізація.

Головна

Навчальний посібник

Програми реалізації

Вихід

← Назад до списку реалізацій

Суфіксне Дерево (Suffix Tree)

Структура даних для зберігання всіх суфіксів рядка. Дозволяє швидко знаходити входження патерна та підраховувати кількість збігів.
Складність: $O(m)$ для пошуку після побудови дерева $O(n)$

Параметри пошуку

Текст для пошуку:

ababcbaba

Введіть текст, в якому буде здійснюватися пошук

Зразок (Pattern):

aba

Введіть зразок для пошуку

Алгоритм:

Суфіксне Дерево (Suffix Tree)

Запустити пошук

Скинути

Порівняти алгоритми

Час виконання алгоритму: 1.32 мс

Крок 1 з 4

Крок 1: Шукаємо символ 'a' у дереві. Знайдено! Поточний шлях від кореня: 'a'.

← Попередній крок

Крок 1 / 4

Наступний крок →

Знайдено збігів: 3

Позиції: 0, 5, 7

Візуалізація структури даних

Деталі поточного кроку

Алгоритм:

Суфіксне Дерево (Suffix Tree)

Текст:

ababcbaba

Зразок:

aba

Пройдений шлях:

a

© 2025. Кафедра Комп'ютерних наук та Програмної Інженерії.
Технології: Flask 3.0, Python, Mysql, Jinja2, Ant Design 5.21, PDF.js, TinyMCE 7 | [Сторінка](#)

Порівняння роботи алгоритмів. Реалізація.

Головна

Навчальний посібник

Програми реалізації

Вихід

← Назад до списку реалізацій

Порівняння роботи алгоритмів

Порівняйте швидкість виконання алгоритмів Suffix Tree та Finite Automaton на однакових даних.

Текст для пошуку

Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book. It has survived not only five centuries, but also the leap into electronic typesetting, remaining essentially unchanged. It was popularised in the 1960s with the release of Letraset sheets containing Lorem Ipsum passages, and more recently with desktop publishing software like Aldus PageMaker including versions of Lorem Ipsum.

Максимум 5000 символів

Зразок для пошуку

Lorem Ipsum

Максимум 500 символів

Порівняти алгоритми

Результати Порівняння

Suffix Tree

Швидший

494.32 мс

Час виконання

Знайдено: 4 збігів

Finite Automaton

572.67 мс

Час виконання

Знайдено: 4 збігів

Algorithm Performance Comparison (ms)

600

500

400

300

200

100

0

Час виконання (мс)

Suffix Tree

Алгоритми

Finite Automaton

© 2025. Кафедра Комп'ютерних наук та Програмної інженерії.

Технології: Flask 3.0, Python, MerginJS, Ant Design 5.21, PDF.js, TinyMCE 7 | GitHub