DOI: https://doi.org/10.36910/6775-2524-0560-2024-55-38
УДК 004.77
Садовий Ян Станіславович, здобувач ступеня доктора філософії
https://orcid.org/0000-0003-0387-6772
Державний університет «Житомирська політехніка», м. Житомир, Україна

ПОРІВНЯЛЬНИЙ АНАЛІЗ РОЗПОДІЛЕНИХ СУФІКСНИХ ДЕРЕВ І ТРАДИЦІЙНИХ МЕТОДІВ УПРАВЛІННЯ ДАНИМИ

Садовий Я.С. Порівняльний аналіз розподілених суфіксних дерев і традиційних методів управління даними. Дана стаття містить огляд різних методів реалізації суфіксних дерев, порівнюючи їх із традиційними методами подання даних, такими як префіксні дерева та перетворення Берроуза-Вілера (BWT). Переглянуті методи включають наївний підхід, алгоритм Укконена та алгоритм розділення та запису лише зверху вниз (Partition and Write Only Top Down – PWOTD). Серед розглянутих методів алгоритм PWOTD виявляється найефективнішим для реалізації суфіксних дерев. Використовуючи розбиття суфіксів, цей підхід значно зменшує вимоги до основної пам'яті для побудови дерева суфіксів, дозволяючи створювати незалежні піддерева повністю в основній пам'яті. Незважаючи на збільшення обчислювальних витрат, пов'язаних із розділенням, алгоритм PWOTD пропонує суттєве скорочення вимог до простору для суфіксів і тимчасових масивів, а також масиву дерева. Порівняльний аналіз показує, що суфіксні дерева перевершують інші методи завдяки ефективному зберіганню та представленню всіх суфіксів певного рядка в компактному та доступному форматі. Крім того, суфіксні дерева підтримують динамічні оновлення з логарифмічною часовою складністю, полегшуючи додавання або видалення рядків з вихідного набору даних без необхідності повної реконструкції, на відміну від префіксних дерев. У підсумку, ця стаття підкреслює ефективність алгоритму PWOTD у реалізації суфіксальних дерев і підкреслює переваги суфіксальних дерев над традиційними методами представлення даних. Універсальність, ефективність і динамічні можливості дерев суфіксів роблять їх незамінними для широкого спектру завдань обробки рядків у різних областях, включаючи біоінформатику, індексування тексту та обробку природної мови.
Ключові слова: суфіксні дерева, алгоритм PWOTD, репрезентація даних, обробка рядків, перетворення, обчислювальна ефективність.

Sadovyi I. The Comparative Analysis Of Distributed Suffix Trees And Traditional Data Management Methods. This article provides a comprehensive review of various methods for implementing suffix trees, comparing them with traditional data representation techniques such as prefix trees and the Burrows-Wheeler transformation (BWT). The reviewed methods include the naive approach, the Ukkonen algorithm, and the Partition and Write Only Top Down (PWOTD) algorithm. Among the methods discussed, the PWOTD algorithm emerges as the most effective for implementing suffix trees. By employing suffix splitting, this approach significantly reduces the main memory requirements for constructing a suffix tree, enabling the creation of independent subtrees entirely in main memory. Despite the increased computational costs associated with partitioning, the PWOTD algorithm offers substantial reductions in space requirements for suffixes and temporary arrays, as well as the tree array. Comparative analysis reveals that suffix trees outperform other methods by efficiently storing and presenting all suffixes of a given string in a compact and accessible format. Moreover, suffix trees support dynamic updates with logarithmic time complexity, facilitating the addition or removal of rows from the original dataset without necessitating complete reconstruction, unlike prefix trees. In summary, this article highlights the effectiveness of the PWOTD algorithm in implementing suffix trees and underscores the advantages of suffix trees over traditional data representation methods. The versatility, efficiency, and dynamic capabilities of suffix trees make them indispensable for a wide range of string processing tasks across various domains, including bioinformatics, text indexing, and natural language processing.
Key words: suffix trees, PWOTD algorithm, data representation, string processing, transformation, computational efficiency.

Вступ та постановка проблеми. Суфіксні дерева являють собою стиснуті таблиці, які зберігають всі суфікси заданого текстового рядка. Ця структура даних інтенсивно використовується для зіставлення шаблонів на рядках і деревах із широким діапазоном застосувань, таких як молекулярна біологія, обробка даних, редагування тексту, переписування термінів, проектування інтерпретатора, пошук інформації, абстрактні типи тощо. Структура даних суфіксного дерева має центральне значення в обробці рядків. Воно забезпечує компактне представлення всіх суфіксів рядка та дає можливість ефективного вирішення проблеми найдовшого загального підрядка, яка полягає у визначенні найдовшого рядка, який є безперервною послідовністю символів, присутніх у двох або більше вхідних рядках. Зокрема, для двох рядків мета полягає в тому, щоб знайти підрядок максимальної довжини, який з'являється в обох рядках. На відміну від підпослідовностей, підрядки повинні займати послідовні позиції в оригінальних рядках. Проблема є важливою для різних застосувань, включаючи біоінформатику для аналізу послідовності ДНК, обробку тексту для виявлення плагіату та стиснення даних для виявлення повторюваних шаблонів. Крім того суфіксні дерева здатні вирішувати проблеми точної відповідності рядків.

З іншого боку, суфіксні дерева являють собою ніщо інше, як метод представлення рядків. Великою проблемою цього типу представлення є те, що він вимагає багато місця для зберігання.

Загалом, суфіксні дерева вимагають у декілька разів більше пам'яті, ніж якби рядки зберігалися без використання подібного типу структурування. Інша проблема полягає в тому, що конструювання суфіксного дерева вимагає великих часових ресурсів. Іншими словами будування такого дерева ще не означає наявність необхідних даних. На фоні цього, існує необхідність порівняння методів представлення даних у вигляді суфіксних дерев з одного боку та традиційних методів представлення даних, які включають: префіксні дерева та перетворення Барроуза-Вілера, з іншого, що і є метою поточного дослідження.

Аналіз останніх досліджень і публікацій. У науково-дослідницькому просторі сьогодення з'являються роботи, присвячені винаходу та аналізу методології по розробці поліпшених методів управління даними.

У роботі [1] досліджувалися алгоритми побудови суфіксних дерев у широкому спектрі джерел даних і розмірів. Було показано, що на сучасних процесорах кеш-ефективний алгоритм зі складністю O(n2) перевершує такі популярні алгоритми, як Укконена та Мак-Крайта, навіть для побудови в пам'яті. Для більших наборів даних вимога до дискового вводу-виводу швидко стає слабким місцем у продуктивності кожного алгоритму. Щоб вирішити цю проблему, було описано два підходи. Спочатку було представлено стратегію керування буфером для алгоритму O(n2). Отриманий новий алгоритм, який має назву «Top Down Disk-based» (TDD), масштабується до розмірів, набагато більших, ніж описано раніше в літературі. Цей підхід значно перевершує найкращі відомі методи конструювання на основі дисків. По-друге, було представлено новий дисковий алгоритм побудови суфіксного дерева, який базується на парадигмі сортування-злиття.

Робота [2] була зосереджена на побудові суфіксного масиву для набору рядків. є Було представлено gsufsort, що являє собою програмне забезпечення з відкритим вихідним кодом для побудови суфіксних масивів і відповідних структур індексування даних для колекції рядків із N символів за O(N) час. Інструмент написаний мовою ANSI/C і базується на алгоритмі gSACA-K, найшвидшому алгоритмі для створення суфіксних масивів для колекцій рядків. Інструмент підтримує великі файли fasta, fastq і текстові файли з кількома рядками як вхідні дані. Експерименти показали хорошу продуктивність на різних типах рядків.

Крім того, варто зазначити праці наступних науковців: Фарруджіа Андреа, Гегі Тревіс, Наварро Гонсало, Пуглісі Сімон, Сірен Джуні [3], Фунакосі Міцуру, Мієно Такуя, Накашима Юто, Іненага Сюнсуке, Баннаї Хідео, Такеда Масаюкі [4], Касерес Мануель, Еварлена Цюса [5] , Кішор Чандра, Джаганатан Субаш Чандра Бос, Азат М., Решмі Т., Марина Нінослав [6], Білазугі Джамаль, Косолобов Дмитро, Пуглісі Симон, Раман Раджив [7], Буше Христина, Цвачо Ондржей, Гегі Тревіс, Голуб Ян, Манзіні Джованні, Наварро Гонсало, Россі Массіміліано [8], Барскі Марина, Стеге Ульріке, Томо Алекс [9], Укконен Е [10], Аяд Лоррейн, Лукідес Григоріос, Пісіс Солон, Вербек Хільде [11], Аджеро Дональд, Тімоті Белл , Мукерджі Амар [12], Рабеа Зеаніб, Ель-Метваллі Сара, Ельмугі Самір, Закарія Магді [13], Ву І, Нонг Ге, Чан Вей, Хань Лін [14], Флік Патрік, Алуру Срінівас [15] та інших.

Проте, беручи до уваги вище зазначену наукову документацію, питання, пов'язане з методологією по розробці поліпшених методів управління даними все ще залишається недостатньо дослідженим та потребує подальшого опрацювання.

Постановка завдання. Метою роботи є проведення порівняльного аналізу розподілених суфіксних дерев і традиційних методів управління даними.

Викладення основного матеріалу дослідження. Побудова суфіксного дерева може бути досягнута за допомогою різних алгоритмів. Ці алгоритми відрізняються за своїми підходами та ефективністю, задовольняючи різні обчислювальні потреби. Наївний підхід, хоч і є простим методом, є неефективним для великих рядків. Більш складні методи включають алгоритм Мак-Крейта, який поступово будує дерево, і алгоритм Укконена, відомий своєю складністю в лінійному часі.

Наївна побудова суфіксного дерева, яка зображена схематично на рисунку 1, передбачає ітеративне вставлення кожного суфікса рядка в дерево по одному. В якості прикладу береться рядок xabxac$. Початок являє собою побудову дерева суфіксів для всього рядка, оскільки суфікс xabxac$ починається з позиції рядка 1. Таким чином рядок проходиться символ за символом.

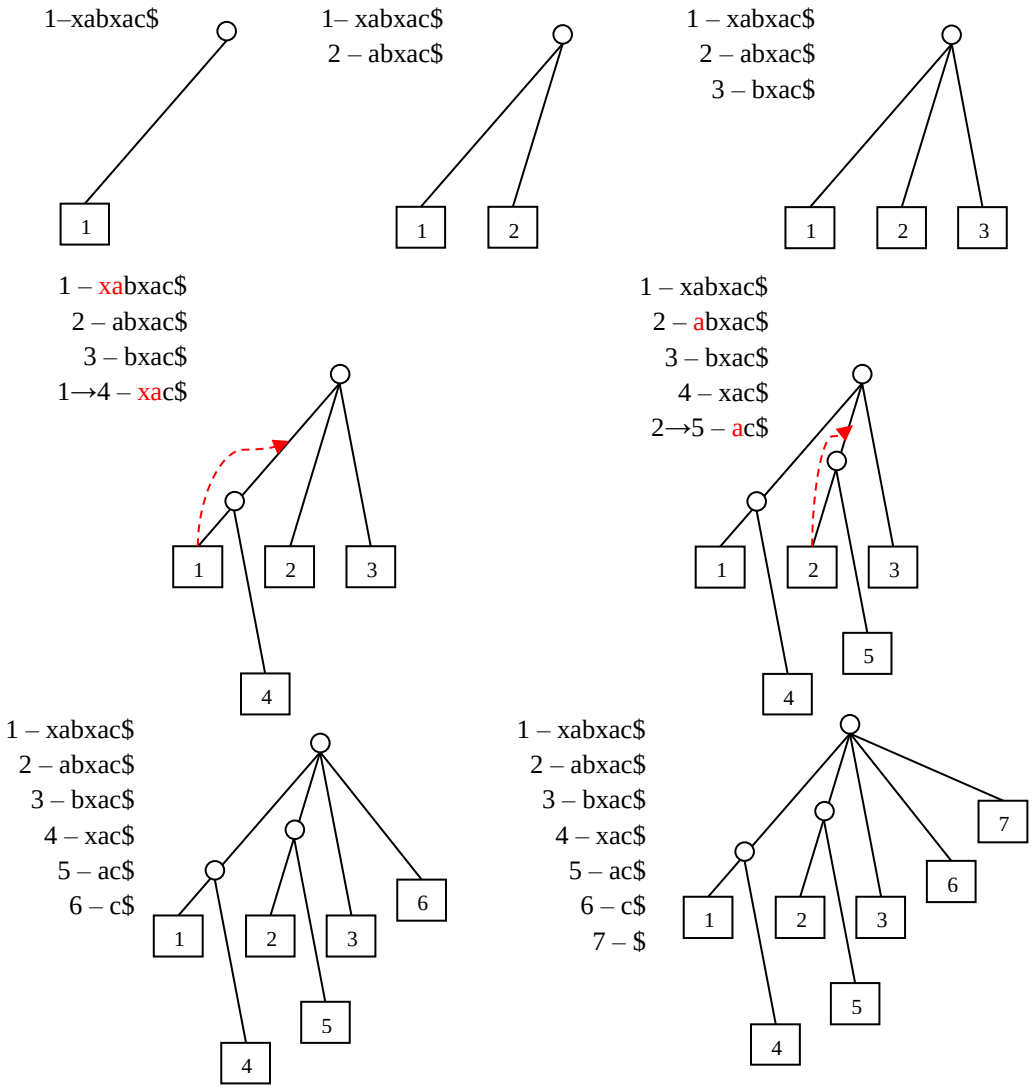

Рис.1. Схема побудови системи суфіксних дерев за допомогою наївного методу

Однак, якщо виникає суфікс, початкові символи якого вже з'являються в дереві суфіксів, існуюче ребро розбивається на два ребра (слова) саме в тому місці, де закінчується рівність між уже існуючим суфіксом і суфіксом, який потрібно вставити. Головною перевагою використання даного методу є його простота, яка полягає в ітеративному вставленні кожного суфіксу рядка. Однак цей метод має значні недоліки, які обмежують його практичність для більш масштабних застосувань, як наприлад складання коротких послідовностей ДНК у повний геном. Найбільш помітним недоліком є його неефективність, наївний підхід має квадратичну часову складність, тобто час побудови різко збільшується з довжиною рядка. Ця неефективність виникає через те, що кожна вставка суфікса часто вимагає переходу від кореня, що призводить до повторних і надлишкових операцій.

Наступним розглядається алгоритм Укконена, який полягає у поступовому будуванні дерева, додаючи по одному символу до вхідного рядка під час створення дерева на льоту. Це інкрементний алгоритм, який починає з дерева для пустого слова і поступово додає нові символи до слова, одночасно оновлюючи дерево суфіксів. Кожен символ додається за амортизований постійний час. З цієї причини даний алгоритм будує суфікс не дерево для слова σ за час $O(|\sigma|)$. Припускається, що ребра до нащадків одного вузла можуть бути проіндексовані їхніми початковими символами при умові, що алфавіт є фіксованим та малим. Коли слово σ розширюється до σx, дерево змінюється наступним чином:

– всі існуючі вузли дерева (включаючи приховані) відповідають підсловам σ. Це також підслова σa, тому вони також зустрічаються як вузли нового дерева;

– якщо β було розгалуженим підсловом, воно залишається розгалуженим – отже, внутрішні вузли залишаються такими;

– кожен новий суфікс βa отримується шляхом розширення вихідного суфікса β.

Якщо β був невкладеним, тобто листом, βa також буде невкладеним. Мітки їхніх країв мають бути подовжені на a. Щоб зробити це ефективним, вводяться відкриті ребра, мітки яких індексують σ від заданої позиції до кінця. Якщо β був вкладеним суфіксом (тобто внутрішнім або прихованим вузлом), тоді або βa присутній у σ. і є вкладеним суфіксом нового слова, і дерево не потребує змін; або βa не зустрічається в σ. Потім необхідно створити новий лист з відкритим краєм і, можливо, новим внутрішнім вузлом, під яким буде з'єднаний новий лист.

Для того щоб визначити чи є суфікс вкладеним, чи ні, припускається, що якщо α є вкладеним суфіксом слова σ, а β є суфіксом слова α, то β також є вкладеним суфіксом слова σ. Слово σ містить як αx, так і αy для деяких різних символів x і y. Оскільки α закінчується на β, десь у σ повинні бути і βx, і βy, тому β є вкладеним. З цієї причини достатньо зберегти найдовший вкладений суфікс слова σ. Він називається активним суфіксом і позначається α(σ). Кожен суфікс $\beta \subseteq \sigma$ є вкладеним коли $|\beta| \leq |\alpha(\sigma)|$. Активний суфікс розмежовує невкладені та вкладені суфікси.

Потім, необхідно підтримувати умову $\alpha = \alpha(\sigma)$ і, додаючи новий символ a, перевірити, чи αa залишається вкладеним. Якщо так, нічого не зміниться. Якщо цього не відбувається, додається новий аркуш і, можливо, також новий внутрішній вузол; потім видаляється перший символ α і продовжується перевірка. Додавання символу до σ викликає амортизовану постійну кількість модифікацій дерева. Щоб виконувати кожну модифікацію в (амортизованому) постійному часі, нам потрібне відповідне представлення слова α, яке підтримує ефективні додавання, вирізання першого символу та перевірки існування відповідного вузла в дереві.

Наступний етап полягає у введенні поняття референтної пари. Референтна пара для слова $\alpha \subseteq \sigma$ являє собою пару (π, τ), де π є вузлом дерева, τ – довільним словом, а $\pi\tau = \alpha$. Крім того, $\tau \subseteq \sigma$, тому τ може бути представлено парою індексів у σ. Референтна пара є канонічною, якщо немає ребра з вузла π з міткою, яка є префіксом τ . Кожне слово $\alpha \subseteq \sigma$ має одну канонічну референтну пару, що представляє його. Заднє ребро $back(\pi)$ веде від внутрішнього вузла π до внутрішнього вузла, який представляє $\pi[1:]$.

Активний суфікс α представляється референтною парою (π, τ). При додаванні a до τ, отримується референтна пара для αa, але вона не обов'язково є канонічною. Перед додаванням необхідно перевірити, чи присутній α в дереві. Якщо π не є коренем дерева, π замінюється на $back(\pi)$ і зберігається τ. В іншому випадку π є порожнім рядком, внаслідок чого перший символ видаляється із τ (збільшується відповідний початковий індекс). Наступним кроком проводиться канонізація, оскількі попередні дві операції можуть створити неканонічну пару. Отже, перевіряється, чи є пара канонічною: для τ 6= ε перевіряється, чи є ребро з π, індексоване значенням τ, достатньо коротким, щоб бути префіксом τ.

Таким чином остаточний вид алгоритму формулюється так. Є вхідна послідовність $\alpha = \alpha(\sigma)$, представлена канонічною референтною парою (π, τ), суфіксним деревом T для σ, задніми ребрами $back$ і новим символом a. Перевіряється, чи є αa в дереві, і за потреби створюється у ньому. Якщо $\tau = \varepsilon$: ($\alpha = \pi$ – внутрішній вузол) Якщо є ребро з вузла π, мітка якого починається з a , то наявне αa. Якщо такого ребра немає, його немає, створюється нове відкрите ребро, що йде від π до нового листа. У противному випадку τ 6 = ε: (α – прихований вузол) Далі шукається ребро з π, мітка якого починається з τ. Якщо після τ у мітці стоїть a, то присутнє αa. В іншому випадку його немає, тому ребро ділиться, та створюється новий вузол так, що ребро від π до нового вузла позначається τ. Новий вузол матиме відкритий край для нового дочірнього вузла. Якщо αa не було присутнім, видаляється перший символ α і операцій починається знову з перевірки наявності αa в дереві. При присутності αa, оновлюється референтна пара, щоб представляти αa та перераховуються задні ребра Наприкінець відомо, що $\alpha = \alpha(\sigma a)$ представляється як канонічна референтна пара (π, τ), суфіксне дерево T наявне для σa та задніх ребер $back$.

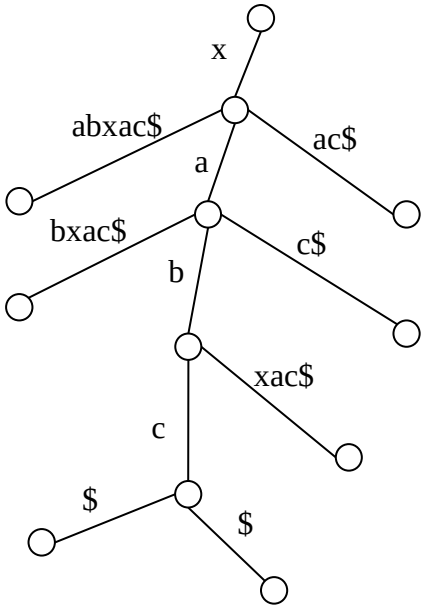

Рис.2. Схематичне зображення побудованого суфіксного дерева завдяки алгоритму Укконена

Узагальнюючи, необхідно відмітити, що однією з переваг алгоритму Укконена є його лінійна часова складність, O(n), що робить його надзвичайно ефективним для обробки довгих рядків. Ця ефективність досягається завдяки розумному використанню суфіксальних посилань і активних точок, що дозволяє алгоритму уникати надлишкової роботи та гарантувати швидку обробку кожної фази. Крім того, алгоритм Укконена поступово створює дерево суфіксів, дозволяючи йому обробляти потокові дані та оновлювати дерево, коли нові символи додаються до рядка. Ця поетапна конструкція особливо корисна в програмах, які вимагають оновлення в реальному часі, таких як текстові редактори та системи пошуку в реальному часі. Крім того, здатність алгоритму обробляти великі алфавіти без значного зниження продуктивності ще більше підвищує його універсальність.

Незважаючи на ці сильні сторони, алгоритм Укконена також має деякі недоліки, які полягають у складності його реалізації та потреба у великій кількості пам'яті, особливо для дуже великих рядків, через необхідність зберігати численні суфіксні посилання та вузли дерева.

Останнім у даній статті розглядатиметься алгоритм Partition and Write Only Top Down (PWOTD). Суфікси вхідного рядка розбиваються на розділи $|X|^{prefixlen}$, де $|A|$ являє собою розмір рядка в алфавіті, а $prefixlen$ – це глибина розділення. Етап розділення виконується наступним чином. Вхідний рядок сканується зліва направо. У кожній позиції індексу i наступні символи $prefixlen$ використовуються для визначення одного з розділень $|X|^{prefixlen}$. Потім цей індекс i записується в буфер обчисленого розділу. Наприкінці сканування кожен розділ міститиме покажчики суфіксів для суфіксів, які мають однаковий префікс розміру $prefixlen$. Необхідно зауважити, що кількість розділів ($|X|^{prefixlen}$) є набагато меншою за довжину рядка. Розділення рядка XABXAC$ використовуючи $prefixlen$ зі значенням 1 створило б чотири розділи суфіксів, по одному для кожного символу в алфавіті, оскільки ігнорується остаточний розділ, що складається лише з символу закінчення рядка $. Суфіксне розділення для символу X буде {0,3}, що представляє суфікси {XABXAC$, XAC$}. Розділення суфікса для символу A буде {1,4}, що представлятиме суфікси {ABXAC$, AC$}.

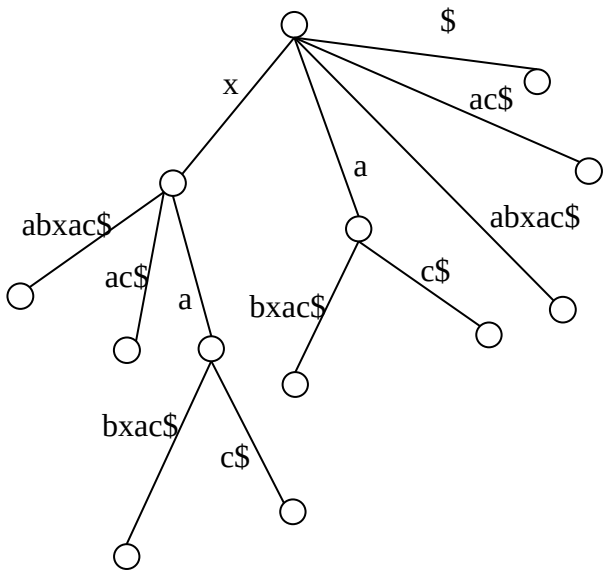

Рис. 3. Схематична побудова суфіксного дерева за допомогою алгоритму PWOTD

У підсумку, Алгоритм PWOTD пропонує ефективний підхід до побудови дерев суфіксів для великих рядків шляхом поділу вхідного рядка та роботи з підмножинами суфіксів. Це розділення зменшує вимоги до основної пам'яті для побудови дерева суфіксів, дозволяючи будувати незалежні піддерева повністю в основній пам'яті. З іншої сторони, однак розділення зменшує вимоги до простору для суфіксів і тимчасових масивів, а також масиву дерева, це відбувається за рахунок збільшення обчислювальних витрат на етапі розділення.

Одним з основних традиційних методів представлення даних є префіксні дерева, які являють собою спеціальні дерева пошуку для зберігання кількох рядків символів одночасно. Дані типи дерев включають тип стиснення даних, оскільки загальні префікси рядків символів зберігаються лише один раз. Вони будуються з використанням набору довільних рядків символів. Кожне вихідне ребро вузла в префіксному дереві анотується одним символом, так що шлях, починаючи від кореня до листа в дереві, представляє один із рядків символів, з яких будується дерево. Порівняно з ними, суфіксні дерева дозволяють виконувати запити швидше та використовують менше пам'яті, особливо для текстів із великою кількістю суфіксів.

Далі, перетворення Барроуза-Вілера представляються як оборотне перетворення вхідного рядка s додаванням до нього унікального символ кінця файлу. Потім формується концептуальна матриця, що містить усі циклічні зсуви s. Далі сортуються рядки цієї матриці в лексикографічному порядку справа наліво і встановлюється значення bw(s) як перший стовпець відсортованої матриці з кінцем – символ файлу видаляється. Необхідно зауважити, що цей процес еквівалентний сортуванню s з використанням, як ключа сортування для кожного символу, його контексту, тобто набору символів, що йому передують. Результатом перетворення Берроуза-Вілера є рядок $bw(s)$ та індекс I у відсортованій матриці рядка, що починається з символу кінця файлу. Приклад даного метода зображений у таблиці 1. Уваги потрубує той факт що, на відміну від суфіксних дерев, найбільшим недоліком алгоритмів, заснованих на перетвореннях Барроуза-Вілера, є те, що вони не працюють в режимі on-line, тобто вони повинні обробити велику частину вхідних даних, перш ніж можна буде створити один вихідний біт.

Тим не менш, основна перевага перетвореннь Берроуза-Вілера порівняно з суфіксними деревами полягає в його простоті реалізації. Зазвичай BWT потребує менше пам'яті та обчислювальних ресурсів порівняно з суфіксними деревами, що робить його більш придатним для програм з обмеженою пам'яттю чи обчислювальною потужністю.

Таблиця 1 – Приклад перетворення Берроуза-Вілера. bw(mississippi) = msspipissii. Матриця справа отримана шляхом сортування рядків у лексикографічному порядку справа наліво.

mississippi•		m ississippi•
ississippi•m		s sissippi•mi
ssissippi•mi		• mississippi
sissippi•mis		s sippi•missi
issippi•miss		s sippi•missi
ssippi•missi	→	i ssissippi•m
sippi•missis		i ssissippi•m
ippi•mississ		i ssissippi•m
ppi•mississi		i ssissippi•m
pi•mississip		s ippi•missis
i•mississipp		i ssippi•miss
•mississippi		i ppi•mississ

Крім того, BWT є особливо ефективним для завдань стиснення, що особливо помітно для повторюваних або структурованих даних, де перетворена послідовність має тенденцію мати довгі цикли повторюваних символів, що дозволяє використовувати ефективні алгоритми стиснення, такі як Move-to-Front (MTF) і Run-Length Encoding (RLE).

Висновки. У даній роботі був проведений огляд основних методів реалізації суфіксних дерев, які включали наївний метод, алгоритм Укконена та алогоритм Partition and Write Only Top Down (PWOTD) та їх порівняння з урахуванням традиційних методів репрезентації даних, таких як префіксні дерева та перетворення Берроуза-Вілера.

При порівнянні різних методів реалізації суфіксних дерев, найбільш ефективним виявився метод на основі алгоритму PWOTD, оскільки він пропонує поділ вхідних рядка та роботу з підмножинами суфіксів. Розділення суфіксів зменшує вимоги до основної пам'яті для побудови дерева суфіксів, дозволяючи будувати незалежні піддерева повністю в основній пам'яті. Тим не менш, дане розділення зменшує вимоги до простору для суфіксів і тимчасових масивів, а також масиву дерева за рахунок збільшення обчислювальних витрат.

При порівнянні суфіксних дерев до інших методів, виявляється що вони, на відміну від інших методів здатні ефективно зберігати та представляти всі суфікси заданого рядка в компактній та легкодоступній формі. Крім того, суфіксні дерева можна модифікувати в логарифмічній часовій складності, дозволяючи ефективно додавати або видаляти рядки з вихідного набору даних без необхідності повної реконструкції, на відміну від префіксних дерев.

Список бібліографічного опису

1. Hankins R., Tata S., Patel J., Tian Y. Practical methods for constructing suffix trees. *The VLDB Journal.* 2005. №14. DOI:10.1007/s00778-005-0154-8.
2. Louza F., Telles G., Gog S., Prezza N., Rosone G. gsufsort: constructing suffix arrays, LCP arrays and BWTs for string collections. *Algorithms for molecular biology : AMB.* 2020. №15. №18. DOI:10.1186/s13015-020-00177-y.
3. Farruggia A., Gagie T., Navarro G., Puglisi S., Sirén J. Relative Suffix Trees. The Computer Journal. 2018. №61. P.773-788. DOI:10.1093/comjnl/bxx108.
4. Funakoshi M., Mieno T., Nakashima Y., Inenaga S., Bannai H., Takeda M. Computing palindromes and suffix trees of dynamic trees. 2024. DOI:10.21203/rs.3.rs-4167645/v1.
5. Caceres M., Navarro G. Faster Repetition-Aware Compressed Suffix Trees based on Block Trees. *Information and Computation.* 2021. №285. DOI:10.1016/j.ic.2021.104749.
6. Cirkoska E., Kishore C., Jaganathan S., Azath M., Reshmi T., Marina N. A Study on Suffix Trees and Their Applications in Genome Sequences Using MUMmer. 2021. DOI:10.1007/978-981-15-9647-6_2.
7. Belazzougui D., Kosolobov D., Puglisi S., Raman R. Weighted Ancestors in Suffix Trees Revisited. 2021.
8. Boucher C., Cvacho O., Gagie T., Holub J., Manzini G., Navarro G., Rossi M. PFP Compressed Suffix Trees. 2021. DOI:10.1137/1.9781611976472.5.
9. Barsky M., Stege U., Thomo A. Suffix trees for inputs larger than main memory. *Inf. Syst.* 2011. №36. P.644-654. DOI:10.1016/j.is.2010.11.001.
10. Ukkonen E. On-line construction of suffix trees. *Algorithmica*. 1995. №14(3) P.249–260.
11. Ayad L., Loukides G., Pissis S., Verbeek H. Sparse Suffix and LCP Array: Simple, Direct, Small, and Fast. 2024. DOI:10.1007/978-3-031-55598-5_11.
12. Adjeroh D., Bell T., Mukherjee A. Suffix trees and suffix arrays. 2008. DOI:10.1007/978-0-387-78909-5_4.
13. Rabea Z., El-Metwally S., Elmougy S., Zakaria M. A fast algorithm for constructing suffix arrays for DNA alphabets.

Journal of King Saud University - Computer and Information Sciences. 2022. №34. DOI:10.1016/j.jksuci.2022.04.015.
14. Wu Y., Nong G., Chan W., Han L. Checking Big Suffix and LCP Arrays by Probabilistic Methods. *IEEE Transactions on Computers.* 2017. P. 1. DOI:10.1109/TC.2017.2702642.
15. Flick P., Aluru S. Distributed enhanced suffix arrays: efficient algorithms for construction and querying. *SC '19: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* 2019. P.1-17. DOI:10.1145/3295500.3356211.

References:

1. Hankins R., Tata S., Patel J., Tian Y. Practical methods for constructing suffix trees. *The VLDB Journal.* 2005. №14. DOI:10.1007/s00778-005-0154-8.
2. Louza F., Telles G., Gog S., Prezza N., Rosone G. gsufsort: constructing suffix arrays, LCP arrays and BWTs for string collections. *Algorithms for molecular biology : AMB.* 2020. №15. №18. DOI:10.1186/s13015-020-00177-y.
3. Farruggia A., Gagie T., Navarro G., Puglisi S., Sirén J. Relative Suffix Trees. The Computer Journal. 2018. №61. P.773-788. DOI:10.1093/comjnl/bxx108.
4. Funakoshi M., Mieno T., Nakashima Y., Inenaga S., Bannai H., Takeda M. Computing palindromes and suffix trees of dynamic trees. 2024. DOI:10.21203/rs.3.rs-4167645/v1.
5. Caceres M., Navarro G. Faster Repetition-Aware Compressed Suffix Trees based on Block Trees. *Information and Computation.* 2021. №285. DOI:10.1016/j.ic.2021.104749.
6. Cirkoska E., Kishore C., Jaganathan S., Azath M., Reshmi T., Marina N. A Study on Suffix Trees and Their Applications in Genome Sequences Using MUMmer. 2021. DOI:10.1007/978-981-15-9647-6_2.
7. Belazzougui D., Kosolobov D., Puglisi S., Raman R. Weighted Ancestors in Suffix Trees Revisited. 2021.
8. Boucher C., Cvacho O., Gagie T., Holub J., Manzini G., Navarro G., Rossi M. PFP Compressed Suffix Trees. 2021. DOI:10.1137/1.9781611976472.5.
9. Barsky M., Stege U., Thomo A. Suffix trees for inputs larger than main memory. *Inf. Syst.* 2011. №36. P.644-654. DOI:10.1016/j.is.2010.11.001.
10. Ukkonen E. On-line construction of suffix trees. *Algorithmica*. 1995. №14(3) P.249–260.
11. Ayad L., Loukides G., Pissis S., Verbeek H. Sparse Suffix and LCP Array: Simple, Direct, Small, and Fast. 2024. DOI:10.1007/978-3-031-55598-5_11.
12. Adjeroh D., Bell T., Mukherjee A. Suffix trees and suffix arrays. 2008. DOI:10.1007/978-0-387-78909-5_4.
13. Rabea Z., El-Metwally S., Elmougy S., Zakaria M. A fast algorithm for constructing suffix arrays for DNA alphabets. *Journal of King Saud University - Computer and Information Sciences.* 2022. №34. DOI:10.1016/j.jksuci.2022.04.015.
14. Wu Y., Nong G., Chan W., Han L. Checking Big Suffix and LCP Arrays by Probabilistic Methods. *IEEE Transactions on Computers.* 2017. P. 1. DOI:10.1109/TC.2017.2702642.
15. Flick P., Aluru S. Distributed enhanced suffix arrays: efficient algorithms for construction and querying. *SC '19: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* 2019. P.1-17. DOI:10.1145/3295500.3356211.