КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет комп'ютерних наук та кібернетики
Кафедра математичної інформатики

«До захисту допущено»
Завідувач кафедри
В. М. Терещенко _____
(підпис)

«___» _____20___ р.

Дипломна робота
на здобуття ступеня магістра

за спеціальністю 122 Комп'ютерні науки

на тему:

ЗАСТОСУВАННЯ МУЛЬТИРОЗДІЛЬНИКОВИХ КОДІВ ДО СТЕМІНГОВАНОГО ТЕКСТУ

Виконав студент 2 курсу
Герман Олег Олександрович _____
(підпис)

Науковий керівник:
доцент, доктор фіз.-мат. наук
Завадський Ігор Олександрович _____
(підпис)

Засвідчую, що в цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент _____
(підпис)

Київ – 2021

РЕФЕРАТ

Обсяг роботи 50 сторінок, 3 ілюстрації, 5 таблиць, 18 джерел посилань.

СТИСНЕННЯ ТЕКСТІВ, СТЕМІНГОВАНИЙ ТЕКСТ, СТИСНЕННЯ МУЛЬТРОЗДІЛЬНИКОВАНИМИ КОДАМИ, ЗБЕРЕЖЕННЯ СТИСНЕНОГО ТЕКСТУ, ЗАСТОСУВАННЯ МУЛЬТИРОЗДІЛЬНИКОВОГО КОДУ, ОБЕРНЕНИЙ СТЕМІНГ.

Об’єктом роботи є процес стиснення початкового тексту за допомогу мультироздільникових кодів та стемінгу без втрат. Предметом роботи є програмна реалізація стемінгу текстів англійської мови та мультироздільникових кодів для стиснення текстів.

Метою роботи є дослідження ефективності стискання стемінгованих текстів методом мультироздільникових кодів та знаходження методів для оптимізації даного процесу.

Методи розроблення: програмування, алгоритм стемінгу англійської мови, створення реалізації мультироздільникових кодів, розробка програми для стиснення текстів без втрат. Інструменти розроблення: безкоштовне для персонального користування, вільно поширюване інтегроване середовище розробки Visual Studio 2019, мова програмування C++.

Результати роботи: досліджено та реалізовано метод мультроздільникових кодів, здійснена реалізація алгоритму стемінгу для англійської мови, були застосовані мультроздільникові коди до стемінгованого тексту без втрат, було розроблена програма, що відтворює стемінг та мультироздільникові коди разом із зберіганням стисненого тексту без втрат.

Програмне забезпечення, що було створено в результаті реалізації мультироздільникових кодів та стемінгу може бути використаним для створення окремого розширення файлу для стиснення та збереження текствих даних.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ

MD – мультироздільниковий, багатороздільниковий;

Fib3 – Фібоначчі 3-го порядку;

RPBC – Restricted Prefix Properties Codes, коди з обмеженими префіксними можливостями;

SCDC – (s,c)-Dense Codes, (s,c)-стиснені коди;

NLP – Natural Language Processing, обробка природньої мови;

ШІ – штучний інтелект;

ML – Machine Learning, машинне навчання

ВСТУП

Оцінка сучасного стану об’єкта розробки. Серед звичайних користувачів комп’ютерів, смартфонів, портативних обчислювальних пристроїв тощо, найбільш розповсюдженні формати збереження текстових даних є «.txt» та «.doc». Звичайний текстовий файл та документ Microsoft Word, відповідно. Дійсно, серед просунутих користувачів, а саме програмістів, викладачів, аналітиків та інших, можуть бути поширені деякі інші системи збереження тексту, які більш зручні в локальному користуванні.

Можна сказати, що вищезгадані розширення файлів «.txt» та «.doc» створили певну монополію серед користувачів. Однак ці формати мають неоптимальний формат збереження. Перший зберігає текст у чистому вигляді без змін, а другий додатково ще зберігає стилі, розмір, кольори, розділи тощо, і це збільшує розмір кінцевого файлу в кілька разів. Однак варто зазначити, що Microsoft Word все ж таки стискає дані, але несуттєво.

Зберігання текстових даних, а особливо у великих розмірах, вважається актуальною проблемою, бо кожен день мільйони людей використовують, зберігають та надсилають текстові файли, розмір яких постійно збільшується. Однак, потенційних розв’язків проблеми великого обсягу даних немає.

Актуальність роботи та підстави для її виконання. Під час роботи із текстами, наприклад реферати, книги, статті, бази даних, звіти, часто виникає необхідність передати цілий файл через мережу чи локальний носій. Навіть із сьогоднішніми технологіями, процес збереження та передачі великих за розміром файлів займає певний час. Іноді скорочення цього часу на кілька хвилин може надати суттєвих змін на оптимізацію певних процесів у виробництвах та обміну інформацій.

Уже більше 20-ти років люди використовують одні і ті ж самі методи збереження текстових даних. Це може виникнути за двох причин: поточні методи вже максимально оптимізовані або немає необхідності у розроблені

нового, альтернативного та актуального методу збереження та стиснення текстових даних. На жаль, жоден з двох попередніх пунктів не відповідають дійсності.

Саме тому, створення нового алгоритму для стиснення та збереження даних вважається доцільним, тому що вже існуючі формати починають мати недостатні можливості. Це постає питання часу, коли популяризується новий метод стиснення та збереження даних. Тому в цій роботі буде здійснена спроба розробити новий алгоритм стиснення даних без втрат за допомогою мультироздільникових кодів.

Мета й завдання роботи. Метою дипломної роботи є розробка програми, що відтворює алгоритм мультироздльникових кодів на стемінгованому тексті. Для досягнення цієї мети були визначені наступні завдання:

- Дослідити та реалізувати мультироздільникові коди;
- Розібрати та запрограмувати алгоритм стемінгу;
- Адаптувати результати стемінгу для MD-кодів;
- Проаналізувати результати та вивести порівняльну статистику;

Об’єкт, методи та засоби розроблення. Об’єктом роботи є процес стиснення початкового тексту втрат, за допомогою програмної реалізації стемінгу текстів англійської мови та мультироздільникових кодів для стиснення текстів.

Основними методами були MD-коди та стемер Портера, тобто реалізація англомовного стемінгу. В якості інструменту реалізації було обрано безкоштовне для персонального користування, вільно поширюване інтегроване середовище розробки Visual Studio 2019, а уся розробка відбувалась на мові програмування C++.

Серед необхідних функцій у цій роботі, усі наявні у мові C++, а саме: двохзв’язні списки, хеш-таблиці, а також висока швидкість виконання програмного коду.

Можливі сфери застосування. Алгоритм стиснення текстів за допомогою мультроздільникових кодів може бути застосований як альтернативне розширення для текстових файлів. Однак цей метод може потребувати певного допрацювання.

Взаємозв'язок з іншими роботами. Метод стискання даних у вигляді мультироздільникових кодів, був обраний через працю І. Завадського та А. Анісімового «Зворотні мультироздільникові коди для стискання», яка і стала натхненням для написання цієї роботи.

РОЗДІЛ 1 МУЛЬТИРОЗДІЛЬНИКОВІ КОДИ

1.1 Введення та визначення

У великих текстових базах даних класичні коди Хаффмана, при застосуванні до слів, що розглядаються як символи, демонструють хорошу ефективність стиснення, що наближається до теоретично найкращої. Однак, в даний час, не тільки швидкість стиснення, але і такі особливості як швидкий пошук стислих даних, висока швидкість декодування та надійність коду мають велике значення. Як відомо, коди Хаффмана не дуже підходять для таких вимог.

Альтернативним підходом до кодів Хаффмана пов'язано з використанням змінної довжини кодів з суфіксом роздільники відомі як шаблонні коди[2]. Нехай *P* буде словом (шаблоном). Властивості таких кодів, як синхронність, повнота, універсальність, і середня довжина кодового слова були інтенсивно вивчені.

Серед шаблонів кодів найбільш відомим є клас кодів Фібоначчі. Ці коди походять від використання номерів Фібоначчі вище замовлень. У новаторській роботі було запущено математичне дослідження кодів Фібоначчі. Сімейство кодів Фібоначчі вважаються надійними при передачі необмеженими рядками та доведена повнота, універсальність та ефективність даних кодів. Тим не менше, для коду Фібоначчі третього порядку (далі Fib3), є випадки, коли декодер повністю втрачає синхронізацію. В якості альтернативи є шаблон коду зі спеціальним суфіксом 01^r. Цей код є синхронним, повним і універсальним. Крім того, він містить велику кількість коротких кодових слів, ніж відповідний код Фібоначчі порядку *r+1*, а його асимптотичний виступ ще гірше.

Ідея шаблонів кодів була збагачена введенням байтових кодів, де кодові слова складаються з інтегральної кількості байтів. Завдяки структурі

байтового вирівняння, ці коди легко конструювати, і вони можуть бути розшифровані і мають швидкий пошук. Це робить байт коди привабливими для практичного використання.

Серед кодів, вирівняних побайтово, RPBC стискає текстові дані краще, хоча показники ентропії все ще далекі від ідеальних (перевищують їх приблизно на 14-16%). Крім того, весь потік з RPBC-стиснутих даних може бути пошкодженим, коли відбуваються розрядні інверсії, вставки або видалення. Інші коди, вирівняні за байтом, можуть подолати проблему синхронізації у разі помилкового біта інверсій, але не додавання або видалення. Таким чином, коди, вирівняні за байтом, не є добре придатними для надійної передачі стиснутих даних.

Поява байт-кодів стимулює розробку алгоритмів декодування, вирівняних на основі байтів, для кодів з бітовими довжиною. Потенціал Fib3 вважається як кращий код Фібоначчі для стиснення тексту і розробив швидкі алгоритми, побайтового вирівняння для декодування та пошуку у стиснутих текстах. Однак, ці алгоритми залишаються значно повільнішими (більше ніж у два рази), ніж ті, байт-коди. У той же час, середню довжину кодового слова Fib3 перевищує значення ентропії на 6-8%, що набагато краще, ніж SCDC/ RPBC, але все ще залишається місце для поліпшень. У цих випадках ентропія є важливим показником ефективноситі.

З наміром вирівняти розподіл даних більш щільно, це не важко узагальнити визначення шаблонів кодів для випадку декількох суфіксних шаблонів. Тим не менш, проста реалізація цієї ідеї може бути марним без ретельного вивчення синхронізації і щільність коду питання. Роздільники "Cappocelli-Style" форми 01^k забезпечують синхронізацію, але неможливо побудувати код з більш ніж одним роздільником цього типу (останній також вірно для "Fibonacci-Style" роздільників форми 1^k), так як коротші роздільники не повинні бути префіксами більше ніж раз[3]. Роздільники форми 01^k0, як видається, хороший вибір, однак, подальші удосконалення можливі.

Розглянемо код з декількома роздільниками-суфіксів форми *01...10*, який також містить деякі слова форми *1...10* отриманих від обмежувачів шляхом вирізання їх крайньої лівої біти. Ці слова дають роздільники разом із закінченням нулів попередніх кодових слів у стиснутому файлі. Для фіксованого набору натуральних цілих чисел $m_1, m_2,...,m_t$, що дається в порядку зростання багатороздільникового коду позначається $D_{m1,...,mt}$, складається з t-слів форми $1^{mi}0$ і всіх інших бінарних слів з суфіксами форми $01^{mi}0$, які не можуть відбутися в інших місцях слова, $i = 1,...,t$.

Дане визначення означає, що роздільники коду в $D_{m1,...,mt}$ є послідовностями форми $01^{mi}0$. Проте код також містить коротші слова форми $1^{mi}0$, що важливо, оскільки використання коротких кодових слів може покращити швидкість стиснення.

Очевидно, що будь-який мультирозділювач коду є префіксом і, таким чином, однозначно декодований. Деякі інші чудові особливості MD-кодів є наступні. По-перше, ці коди можуть працювати з (m_1+2)-бітовим роздільником $01^{m1}0$, подібно до кодів моделі, і в той же час містити (m_1+1)-бітне слово, яке коротше найкоротшим роздільником. По-друге, вони можуть використовувати багато обмежувачів, і, змінюючи їх довжини, можна щільно Вирівняти код до заданої ймовірності розподілу вхідних текстових символів. По-третє, у зв'язку з формою обмежувачів, MD-коди є синхронними. Нарешті, MD-коди мають просту структуру звичайної мови. Це дозволяє нам створювати ефективні кодування, що вирівнюються байт, декодування і алгоритми пошуку за допомогою розширення байт його побітових скінченних автоматів перехідної таблиці.

Цікаво відзначити, що хоча MD-коди не є шаблоном, вони можуть розглядатися як узагальнення деяких класів шаблонів кодів. Також, код $D_{k-\infty}$, тобто код з усіма можливими роздільниками, що містять k або більше, можна розглядати як набір всіх бінарних рядків, в яких візерунок $1^{\kappa}0$ зустрічається

лише як суфікс. Цей код є ізоморфним для коду Capocelli $S\,(k + 1, 01^k)$, який, у свою чергу, можна розглядати як узагальнення деяких кодів Фібоначчі.

Тим не менш, для MD-кодів, важливою проблемою оптимального індексування словника не було уточнено. Всі алгоритми кодування/декодування ми розглядаємо вписатися в наступну загальну схему. Під час кодування використовується зіставлення (слово тексту, кодові слова), де слова тексту сортуються за спаданням їх частоти, а кодові слова сортуються за зростанням довжини. Процес декодування скасовано: слід створити зіставлення з набору кодових слів до набору текстових слів. Для того, щоб швидко декодуватися, структура даних з низьким часом доступу повинна бути використана для зберігання слів тексту. Для цих цілей найефективніша структура даних є масивом із цілочисельними індексами. Однак вона вимагає побудови ефективного відображення набору цілих індексів до набору кодових слів.

Багатороздільювач кодів має один важливий недолік: кодові слова $\psi(1), \psi(2),\ldots$ не сортуються в порядку збільшення їх довжини. Випливає, що кодові слова $\psi(1),\ldots, \psi(n)$ не є набором n найкоротших кодових слів. Однак для того, щоб отримати максимальну ступінь стиснення, слід використовувати n найкоротших кодових слів $\psi(i_1),\ldots, \psi(i_n)$, де в може бути набагато більше, ніж n. Таким чином, масив, що має непорожні елементи з індексами $i_1,\ldots,i_n$ може бути дуже рідкісним і непрактичним навіть для кодування відносно коротких текстів, що містять 2000-3000 різних слів. Найкращим варіантом є створення монотонного кодування ψ (тобто якщо $i_1 > i_2$, то $| \psi (i_1) | \geq | \psi (i_2) |$).

Однак, для MD-кодів здається проблематично побудувати зворотну картографічну ψ^{-1} на основі обробки біти кодові слова зліва направо, оскільки обробка частини кодового слова не дає багато інформації про позицію цілого кодового слова в словнику. Можливо, таке відображення може бути побудовано для коду без префіксу, якщо коротші кодові слова є префіксами довших. У цьому випадку розшифровка кодового слова uv, де u також кодові

слова, може бути зроблено наступним чином: спочатку, в впорядкований набір всіх кодових слів ми отримуємо індекс вас, а потім збільшити цей індекс за деякими обчислене значення, яке залежить від *v*. Зокрема, такий підхід дозволяє нам обробляти байт-за-байтові кодові слова, які займають кілька байтів. Цей фактор є ключовим для стиснення даних та подальшого розрахунку ентропії.

Важливо, якщо писати біти MD-кодового слова в зворотному порядку, то ми отримуємо не префікс, але однозначно декодований код. Дійсно, кодований файл може бути розшифрований справа наліво як багаторозділювач коду. Крім того, він може бути однозначно розшифрований зліва направо, так як в цьому коді всі роздільники залишаються незмінними, як у вихідному багатороздільникового коду, однак, вони розміщуються в початках кодових слів замість їх кінців. Основним предметом цієї презентації є такі «зворотні» мультироздільникові коди. Для зміни цього способу мультірозділювача кодів, найбільш примітним властивістю якого є наявність монотонного кодування зіставлення з набору природних чисел до набору кодового слова, для яких просте зворотне декодування на основі обробки біти зліва направо може бути легко побудована.

1.2 Набір кодових слів

Нехай *m* дорівнює $\{m_1,...,m_t\}$ – це набір цілих чисел, заданих у порядку зростання.

Визначення. Зворотний багатоканальний розділювач $R_{m1,...,mt}$ складається з усіх слів форма 01^{mi}, *i* належить проміжку *1,...,t* і всі інші слова, які відповідають наступним вимогам:

1. слово починається з префіксу $01^{mi}0$ для деяких *mi*, що належать до *M*;
2. для будь-якого $m_i \in M$ слово не закінчується послідовністю 01^{mi};

3. для будь-якого $m_i \in M$ слово може містити послідовність $01^{mi}0$ тільки як префікс.

Дане визначення означає, що роздільники коду в $R_{m1,...,mt}$ є послідовностями форми $01^{mi}0$. Однак, код також містить коротші слова форми 01^{mi}, які утворюють роздільник разом з першим нулем наступного кодового слова.

Тепер ми готові побудувати мономонічне кодування з набору натуральних чисел до набору кодових слів реверсу мультироздільникового коду $R_{m1,...,mt}$. Нехай k дорівнює $\{k_1,...,k_q\}$ бути зростанням послідовності всіх цілих чисел у діапазоні $[0, m_t+1]$, які не належать до m. Наприклад, K дорівнює $\{0, 1, 3, 6\}$ для коду $R_{2, 4, 5}$. Зверніть увагу, що кожне слово з $R_{m1,..., mt}$ має наступну структуру: вона складається з префіксу 01^{mi}, для деяких $m_i \in m$, які можуть дотримуватися деякі групи бітів, що мають форму 01^s, де $s \in K$ або $s > kq$. Зворотна заява також правильна: будь-яка послідовність дії описаної структури формує кодові слова.

Тому в коді $R_{m1,...,mt}$ будь-якого кодового слова довжини L може бути побудовано в одному (і тільки один) з наступних способів:

1. це слово форми 01^{mi}, i належить проміжку $1,...,t$;
2. воно складається з кодового слова довжини L-s-1, після чого послідовність 01^s, s належить до K;
3. воно складається з кодових слів довжини L-1 з суфіксом 01^r, r менше за MT, який додається одним "1" біт.

Спираючись на ці факти, ми формулюємо принцип побудови впорядкованого набору кодових слів довжини L, коли вже побудовано набори коротких кодових слів.

1. ітеруючи i від 1 до q, розмножувати набори кодових слів довжини L-k_i-1 і доповнити послідовності форми $01ki$, Ki належить K, до них.
2. відтворити набір слів довжини L-1 з суфіксом $01r$, r менше за MT і додати один "1" біт для всіх елементів цього набору.

3. якщо L дорівнює $mi+1$, i належить $\{1,..., t\}$, додайте слово 01^{mi} до набору кодових слів.

Якщо ми покроково застосовуємо цей підхід до формування наборів кодових слів довжини m_1+1, m_1+2,..., ми отримуємо набір всіх $R_{m1,...,mt}$ кодових слів наказав за рахунок зростання їх довжини. Наприклад, набір $R_{2,4,5}$ кодових слів довжини 8 або менше дається на рисунку 1. Крім того, він демонструє, як слова довжина 8 побудовані з слів довжини 7, 6 і 4, додавши розрядні послідовності 0, 01 і 0111 відповідно (правило 1). Три слова, які виділені сірим кольором, будуються шляхом застосування правила 3. Найкоротше слово, яке будується шляхом застосування правила 2, має довжину 11. Його конструкція показана в лівій нижній частині рисунку 1.

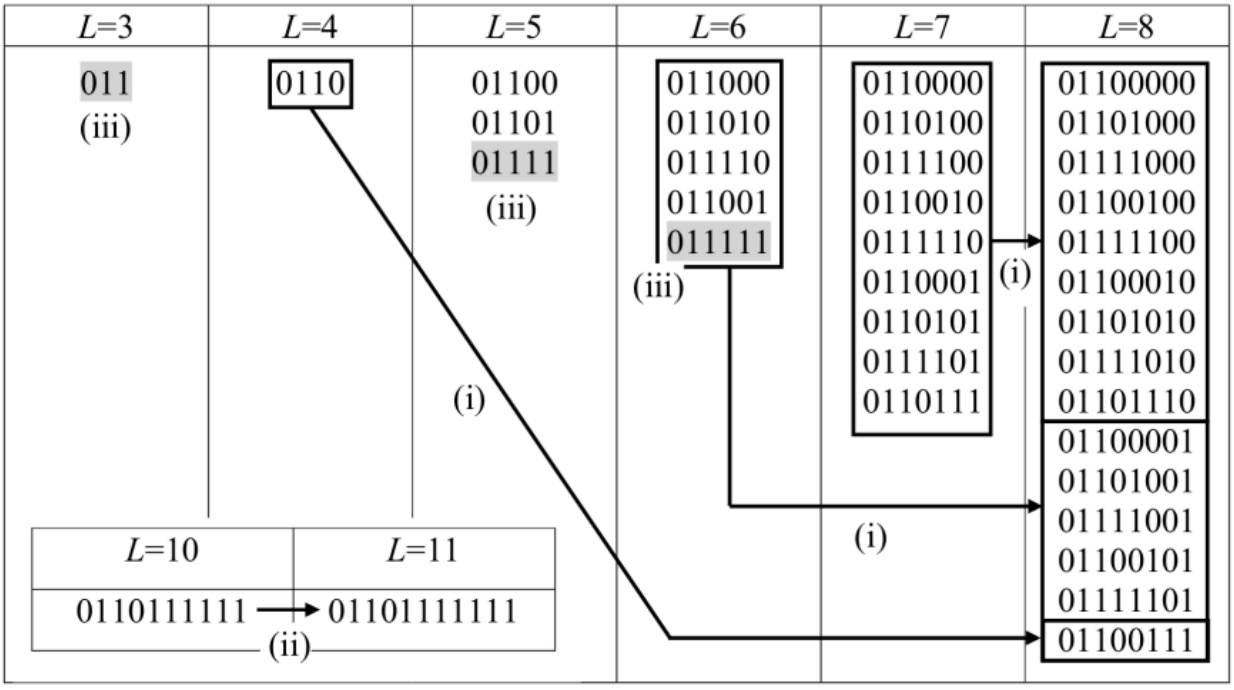

Рисунок 1 – $R_{2,4,5}$ конструкція кодового слова

Будь-який зворотний мультирозділювач коду $R_{m1,...,mt}$ містить таку ж кількість кодових слів заданої довжини, як "прямий" мультироздільниковий код $D_{m1,...,mt}$. Таким чином, зворотний MD-коди мають такі властивості MD-

кодів як повнота і універсальність, а також їх асимптотичні щільності, що являється показником ефективності.

1.3 Кодування та декодування

Після набору коду слів $c_1, c_2,...$ впорядкована за зростанням їх довжини будується, кодування тексту стає тривіальним. Це вимагає лише замовлення слів тексту, які будуть закодовані відповідно до порядку убування їх частоти. Слово w_i в цій впорядкованої набір отримує кодові слова c_i.

Декодування є досить більш витонченими. Насправді, він завершує визнаючи кодові слова c_i в потоці кодового слова і обчислення його індексу я в наборі кодового слова побудовані за принципом, заданого в попередньому розділі. Зверніть увагу, що код $R_{m1,...,mt}$ є звичайною мовою в двійковому алфавіті, який розпізнається скінченним автоматів, що обробляє біти кодового слова зліва направо. Таким чином, ми можемо застосувати цей автомат для розрахунку індексу кодового слова в впорядкований набір кодових слів.

У продовження ми зосередимося на коді $R_{2, 4, 5}$, оскільки він покращує код $D_{2, 4, 5}$, який перевершує всі інші MD-коди з 3 або менше обмежувача в стисненні велика тексти. Схоже, аналогічні конструкції можна застосувати до будь-якого зворотного мультирозділювача коду. Декодувати автомат для коду $R_{2, 4, 5}$ показаний на рисунку 2. Він обробляє кодові слова в потоці, починаючи в державі, позначену заповненим колом і обробкою в державі 1 після обробки розділювача на початку деяких кодових слів. Таким чином, для декодування буде успішно завершена, кодований файл повинен бути додані з деякими роздільником, наприклад, 0110. Автомат розпізнає слова біт-за-біт і обчислює значення двох змінних: L-довжина кодових слів і p-результуючий індекс кодового слова. Під час кожного переходу, вхідний біт вказується перед скісною рискою і операції на L і p позначені після лінії (порядок операцій важливий).

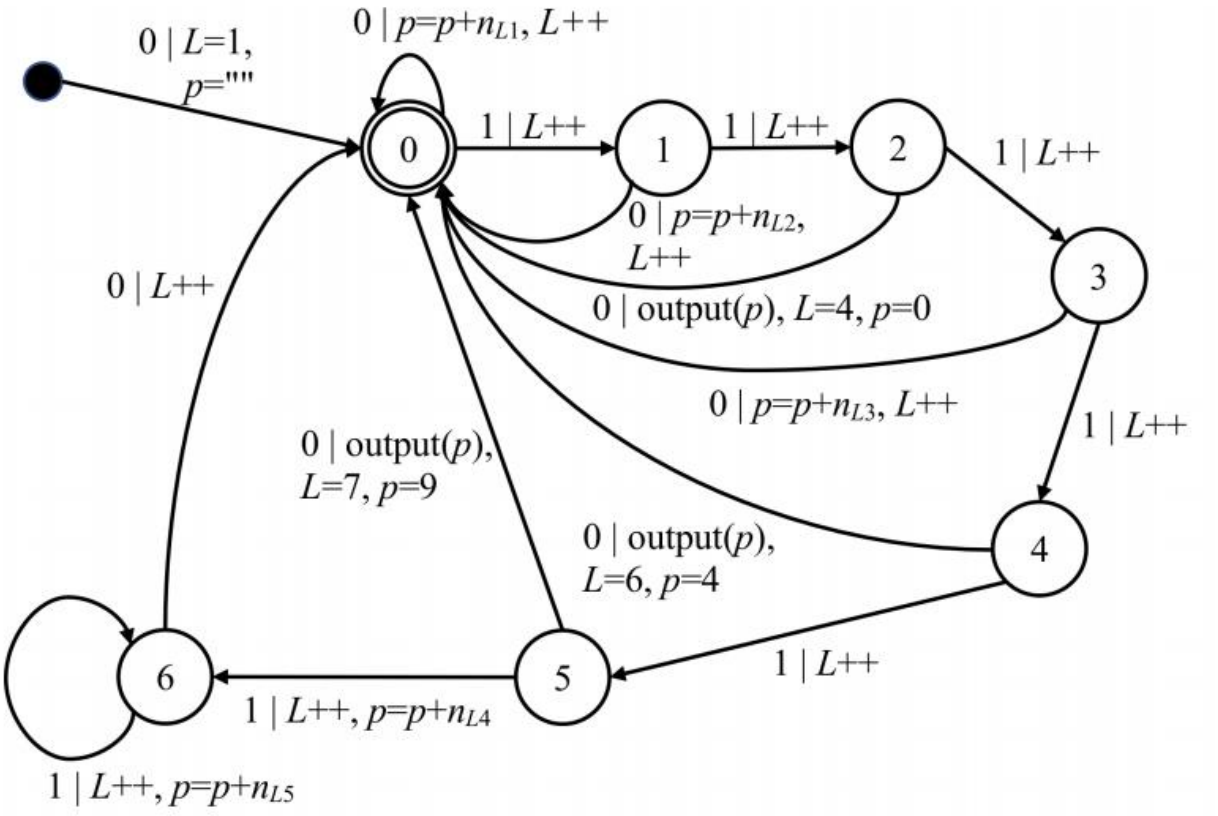

Рисунок 2 – Розшифровка автомата для $R_{2,4,5}$

Пояснимо, як працює автомат. Будь-яке слово $R_{2,4,5}$ можна представити як конкатенації бітових послідовностей форми 01^t, де $t \geq 0$. Після обробки будь-якої такої послідовності і "0" після того, як автомат доходить до стану 0. Розрядні послідовності 011, 01111 і 011111 можуть бути тільки префіксами кодових слів. Після їх обробки, автомат приходить до стану 0 з держав 2, 4 або 5, виводить результуючий індекс p попередніх кодових слів та ініціалізує p з новим значенням 0, 4 або 9, що відповідає індексу кодового слова 011, 01111 або 011111. Крім того, на переходи 2 → 0, 4 → 0 і 5 → 0, L ініціалізується відповідна довжина кодових слів, а на всіх інших переходами він збільшується на 1. Після обробки бітових послідовностей форми *01...10* з 0, 1, 3, 6 або більше, Автомат робить переходи до стану 0 з станів 0, 1, 3 і 6, відповідно i додає до кодового слова положення p значення n_{Li}. Якщо i менше 5, n_{Li}

дорівнює зсуву кодових слів в списку всіх кодових слів через додавання розрядної послідовності форми 01^{ki} ($k_i \in k = \{0, 1, 3, 6\}$), коли довжина результуючих кодових слів є L. Значення n_{L5} дорівнює зсуву кодових слів довжини $L-1$, з 6 або більше з них в кінці, у списку всіх кодових слів через додавання "1" біт і виробництво результуючих кодових слів довжини L. Зауважимо, що вищезгаданий принцип побудови списку кодових слів гарантує, що додавання тієї ж групи $01...1$ для будь-якого кодового слова заданої довжини зміщується це кодові слова на тій же відстані, а також додавання одинарного "1" біт. Значення n_{Li} можуть бути розраховані на етапі попередньої обробки разом з впорядкованим набором кодових слів.

Застосування автомату декодування для обробки закодованого тексту біт-за-біт може бути досить повільним. Тим не менш, ми можемо зробити байт "кількісна оцінка" автомат. Це робиться шляхом ітеративного читання фіксованої інтегральної кількості байтів і виробництва відповідних вихідних номерів.

Кількісно автомат дається як таблиця підстановки $TAB[a_{start}][L_{start}][x]$, що складається з кортежів (p, a_{fin}, L_{fin}), де x є частиною байт закодованого файлу, a_{start} є станом автомату перед обробкою x, a_{fin} приходить після обробки x, L_{start} це довжина вже декодованому частина останніх кодових слів, який знаходиться під обробкою, коли декодування x починається, LF в це довжина декодованої частини останніх кодових слів, створеного з x та p є виходом створеним автоматом при обробці x.

Декодування байтів наведений у додатку А, вирівняних за кодом $R_{2,4,5}$. Цей же алгоритм може також застосовуватися до будь-якого зворотного MD-коду з найкоротшим роздільником 011. Його таблиця підстановки побудована на автоматі, щоб кількісно читати байти кодованого тексту. Для того, щоб швидко декодувати, ми використовуємо серію одномірних таблиць підстановки $TAB_j[x]$ замість однієї трьохмірної. Номер j таблиці підстановки залежить від параметрів a_{start} і L_{start}, наприклад, він може бути отриманий як

$l_{start}*7+a_{start}$ (з 7-численних станів автомата). Аналогічно, кортежів в таблиці підстановки містять тільки один номер *JF* замість пари (a_{fin}, L_{fin}). Вивід *p* представлено числами p_1, p_2, p_3, p_4 – показниками декодувати кодових слів або їх декодовано частинами в масиві всіх кодових слів (легко показати, що не більше 4 кодових слів може бути розшифрований, повністю або частково, при обробці одного байту коду з найкоротшим роздільником *011*). Значення p_1 також може представляти зсув кодових слів *uv* в списку кодових слів до його вже декодувати префікс *u*.

Легко обчислити розмір таблиці підстановки представленого алгоритму вирівнювання байтів, вирівняних. Для більшого тексту, в наших експериментах, що містять 288 тисяч різних слів, максимальна довжина кодових слів *L* є 34.

З урахуванням кількості станів автомата і простору, необхідного для зберігання значень p_1-p_4 та *j*, ми отримуємо приблизно 600 кілобайт пам'яті, необхідної для зберігання всіх масивів масиви TAB_j, що цілком прийнятно для сучасних комп'ютерів.

Для перевірки ефективності, варто застосовувати ентропію, так як це вважається чудовим показником для визначення безладу, випадковості або невизначеності. Ентропія – це фізична властивість, і в той же час наукове поняття, яку використовують у різних сферах науки.

Вперше термін ентропії був використаний у термодинаміці, щоб визначити залежність внутрішньої енергії від зовнішніх параметрів. Тому і був введений цей термін, що у якомусь коефіцієнті, який переводиться у відсотки, визначити цю залежність.

В нашому випадку ентропія необхідна для визначення ефективності стискання. Якщо уявити, що отриманий набір бітів це випадкова подія, то співвідношення цього набору до початкових значень, то можна отримати відсоток ефективності стискання.

Як було згадано у підрозділі 1.1, доброю ентропією вважається, якщо показник не перевищує значення 14%-16%. Ідеальні показники ентропії повинні наближатися до 0, проте в реальності, вони мусять бути менше принаймні 10%.

РОЗДІЛ 2 СТЕМІНГ ТЕКСТУ

2.1 Обробка природньої мови

Під час програмної обробки тексту, написаного природною мовою, нерідко виникає необхідність у морфологічному розборі слів та речень. У такій ситуації застосовуються методи, принципи та алгоритми NLP.

Обробка природньої мови (NLP) – це результат комбінування лінгвістики, інформатики та сфер штучного інтелекту, який опирається на зв'язок та взаємодію людини з комп'ютером. Основна задача NLP полягає в тому, щоб адаптувати природну (людську) мову, щоб її зрозуміла комп'ютерна програма. В результаті, комп'ютер мусить розуміти вміст документів, сенс тексту та відрізняти контекстні висловлювання. Таку систему використовують чат-боти, пошукові системи, голосові детектори тощо.

З 50-х років минулого століття почали з'являтися перші спроби створення NLP систем. Цими програми частіше являлися автоматичні перекладачі та збірниками мовних правил. Наприклад, перекладач з російської мови до англійської (Georgetown–I.B.M. experiment[6]) та китайський розмовник (Chinese room[7]).

Саме у період з 1950 до 1990 була так звана епоха символічного NLP через відсутність повного доступу до джерел різних мов та цифрових даних й текстів, тобто конвертованих або передрукованих у файли, аналогічних .txt, на комп'ютер.

До 2010-х років, почався активний розвиток інтернету до загальнодоступності інформації та даних. Цей період називають епохою статистичного NLP, бо більшість програм та систем були створені, які почали базуватися на великих базах даних. У цей період починають виникати програми машинного перекладу, які вдосконалюються і по сьогодні. Популяризація інтернету та безкоштовних і публічних даних дали

неймовірних поштовх у розвиток NLP та обширність інформації, котру можна обробити.

З 2010-х по сьогодні настала епоха нейронного NLP. Завдяки розвитку нейронних мереж, сьогодні у кожного є можливість знаходити та перекладати текст у режимі «реального часу». Також NLP системи на основах ШІ досягли неймовірних результатів в обробці природньої мови, синтаксичному та морфологічному аналізі, моделювання мови тощо. Також нейронні мережі доповнюють технології та дані для програм із статистичної епохи, що позволяє сучасним програмам поповнювати свої бази даних та знань у реальному часі й відразу застосовувати їх.

Сьогодні NLP має багато, але в той же час, поширених та необхідних завдань:

а) Обробка мовлення та тексту:
   - *Оптичне розпізнавання*. Знайти та визначити друкований або прописний текст на зображенні;
   - *Аудіо-розпізнавання мовлення*. На звуковій доріжці, знайти розмову людини або групи людей та визначити текст. Вагома складність цього завдання – це природне мовлення. Люди можуть недотримуватися певних пауз між словами чи висловлюванням, а також ігнорувати деякі правила та порядок слів;
   - *Сегментація мовлення*. На відміну від аудіо-розпізнавання мовлення, задача сегментації мовлення, це зі звукової доріжки визначити лише слова, не зберігаючи зміст розмови;
   - *Перетворення тексту в мовлення*. Отриманий текст необхідно перетворити на звукову доріжку. Зазвичай голос синтезується через збір великої бази фраз однієї людини. Після обробки усіх фраз, є можливість скомпонувати звукову доріжку людського мовлення;

- *Токенізація*. Іншими словами – сегментація слів. Процес розділення тексту на слова, зазвичай, орієнтуючись на пробіли. В багатьох мовах така практика є тривіальною і несе сенс для слів, які необхідно обробляти;

б) Морфологічний аналіз:

- *Лематизація*. Визначення леми у слові. Лема – це нормальна або словникова форма слова, рідше інфінітив;
- *Морфологічна сегментація*. В деяких мовах практикується словотворення з декількох слів, тому для них варто застосовувати морфологічну сегментацію. Її задачею є виділення в слові морфем та визначення класу морфем;
- *Стемінг*. Це процес скорочення слова до базової форми, часто збігається з морфологічним коренем слова. Стемери прибирають кінцівки та суфікси слів, рідше префікси;

в) Синтаксичний аналіз:

- *Індукція граматики*. Створення цілої та офіційної граматики, що повністю описує синтаксис мови. Можна також створювати граматику для вигаданих та фентезійних мов;
- *Розбір речення*. Процес визначення дерева розбору речення. Це складний процес визначення підметів, присудків, другорядних слів та взаємозв’язку між словами. Ключовим моментом може бути визначення належності прикметника до конкретного слова і тому подібного. Існує два основних типи синтаксичного аналізу: аналіз виборчих околів та синтаксичний аналіз залежностей, який фокусується на пошук предикатів;

г) Семантика:

- *Лексична семантика*. Визначення обчислювальних значень слів у контексті речення або навіть тексту. Також вона має наступні підтипи:

— *Дистрибутивна семантика*. Відповідає на питання: «Як можемо засвоїти семантичні подання зданих?»;
— *Аналіз настрою*. Знаходять суб'єктивну інформацію, зазвичай із набору текстів та документів. Часто застосовується для аналізу статей та публічних думок в інтернеті;
— *Розпізнавання іменованої сутності*. Визначення в тексті, які слова є власними, тобто виявлення імен, назв країн, місцевостей, організацій тощо;
— *Вилучення термінології*. Задачею цього процесу є автоматичне вилучення термінів із даного тексту, абзацу чи уривку;
— Неоднозначність сенсу слова. Деякі слова можуть мати різні значення, особливо, що залежать від контексту. Задачею цього процесу – це визначення вірного значення слова на основі контексту речення чи цілого тексту;

– *Реляційна семантика*. Визначення обчислювальних значень цілого речення. Також має наступні підтипи:
— *Витяг відносин*. Береться частина тексту та визначається взаємозв'язок між обраними сутностями;
— *Семантичний синтаксичний розбір*. Обирається фрагмент тексту (зазвичай речення) та подається офіційне представлення семантики цього фрагменту, рідше у вигляді графіка;
— *Семантичне маркування ролей*. Вибирається речення та визначається роз'єднані семантичні предикати, так звані, фрейми речення;

– *Дискурс*. Це процес здійснення семантики за межами окремих речень. Ділиться на наступні підтипи:
  — *Аналіз дискурсу*. Це сукупність різних задач дискурсу. Головною є визначення дискурсної структури зв’язаного тексту, а саме природи дискурсних зв’язків між даними реченнями;
  — *Роздільна здатність кореферентності*. Аналізуючи речення або фрагмент тексту, визначається, які слова стосуються або відносяться до конкретних об’єктів («сутностей»). Саме цей процес може відділяти займенники та присвоює їх то іменників та саме імен;
  — *Неявне семантичне маркування ролей*. Досліджує речення, визначає та роз’єднує семантичні предикати, частіше словесні фрейми, та явні семантичні ролі цих предикатів в поточному реченні. Після цього визначає семантичні ролі, які точно не реалізовані в даному реченні, класифікує їх і створює на основі цих класів аргументи, які точно реалізовані за межами цього речення в тексті;
  — *Визначення текстового втягування*. Цей процес порівнює два речення або фрагменти тексту та визначає, чи схожі вони за сенсом, відповідають дійсності, заперечують одні одному чи позволяє одній бути хибною чи істинною примусово;
  — *Сегментація та розпізнавання тем*. Фрагмент тексту ділиться на сегменти, кожен з яких відповідає якійсь темі та необхідно визначити тему цього сегменту;
  — *Видобуток аргументів*. За допомогою видобутку аргументів з речення, відбувається автоматичне вилучення та ідентифікація аргументованих структур із поточного або даного тексту. Ці аргументовані структури включають в себе

передумову, висновки та схему аргументації. Також мають взаємозв'язок між основними та допоміжними аргументами або контраргументами;

ґ) Програми NLP вищого рівня (генерація текстів та природної мови, машинний переклад, кібернетика діалогу, виправлення помилок, аналіз тексту та питань тощо).

Під кожну задачу та систему виводяться свої формули, моделі та методи. Проте, Джордж Лакофф привів узагальнену методологію для побудови NLP алгоритмів та токенізації вхідних даних за допомогою двох аспектів:

1. Застосування теорії концептуальної метафори, яка надає уявлення про те, що хоче донести автор. Тобто надання важливості чи вагомості словам, без якої сенс речення залишиться дещо неоднозначним для людини та когнітивного алгоритму NLP;
2. Призначення відносних міри значення слову, фразі, реченню чи фрагменту тексту на основі інформації, представленої до та після цього ж фрагменту тексту;

$$RMM(token_N) = PMM(token_N) \times$$
$$\times \frac{1}{2d}\left(\sum_{i=-d}^{d}((PMM(token_{N-1}) \times PF(token_N, token_{N-1}))_i\right) \quad (2.1)$$

де RMM – відносна міра значення;

token – токен, будь-яка частина тексту, речення, фрази або слова;

N – кількість токенів;

PMM – ймовірна міра значення

d – розташування токену вздовж послідовності токенів N-1

PF – функція ймовірності (для кожної мови специфічна)

Це дало поштовх для когнітивного ШІ[10]. Ця сфера штучного інтелекту передбачає вивчення когнітивних явищ у програм та машинах. Завдяки цьому

з’являється більше інструментів для моделювання людського інтелекту та вивчення когнітивних явищ.

Варто зазначити певні переваги NLP із появою алгоритмів машинного навчання. А саме:

- Процедури навчання, що використовуються під час ML, зосереджуються в автоматичному режимі на найпоширеніших випадках, коли при написанні правил вручну часто зовсім не очевидно, куди слід застосовувати зусилля, бо іноді важко визначити, які випадки важливіші в даній ситуації;
- Процедурам автоматичного навчання доступні алгоритми статистичного висновку для створення моделей, стійких до невідомих або незнайомих термінів (наприклад, містять слова або конструкції, яких раніше не бачили), і до помилок під час вводу (наприклад, словами, що були невірно написані або пропущені випадково). Як правило, витончено обробляти такі винятки за допомогою рукописних правил, що приймають м’які рішення, надзвичайно складно, через це часто виникають помилки і вимагає багато часу;
- Системи, засновані на автоматичному засвоєнні правил, можуть бути більш коректними, через те, що мають більше вхідних даних. Однак, системи, які створені на рукописних правилах, можуть бути більш точними лише через збільшення комплексності правил, що є набагато складнішим завданням. Однак, існує обмеження складності систем, заснованих на правилах, що написані вручну та поза якими системи стають дедалі менш керованими. Проте, створення більшої кількості даних для внесення до системи ML просто вимагає аналогічного збільшення кількості людського часу на роботу, зазвичай, без збільшення складності процесу виписування.

Однак, популярність ML в обробці природньої мови, існують випадки, коли рукописні правила будуть більш ефективні ніж NLP на основі машинного навчання. Іншими слова такий метод також має певні недоліки саме через комплексність цієї системи. Неефективним метод ML в природній обробці мови може бути:

- Коли обсяг даних для навчання є недостатнім для ефективного застосування методів машинного навчання, наприклад, для автоматичного перекладу мов з обмеженим рівнем ресурсів, таких як система Apertium;
- Для початкової обробки за допомогою методів NLP, наприклад, токенізація;
- Для подальшої обробки та зміни вихідних даних конвеєрів оброки природної мови, таких як, вилучення знань із різних синтаксичних розборів.

2.2 Стемінг

Під час створення програм та додатків, які повинні зберігати, або шукати слова, то частіше за все використовують стемінг. Основною із задача стемінг є скорочення слів для зменшення розміру під час зберігання у файлі. Стемер – це програма або реалізація, яка застосовує процес стемінгу до слова і знаходить його основу, не корінь. Тобто скорочене слово, ця основа, може мати зовсім несхожий зміст та сенс, ніж воно мало в початковому вигляді. Стемінговане слово може збігатися з морфологічним коренем слова, але це не завжди так.

Частіше стемінг можна зустріти у пошукових системах як «Google», «Bing» тощо та нормалізації тексту. У цій роботі буде використана інша властивість стемінгу як зменшення розміру слова. Стемінговані слова можна

зберігати в скороченому вигляді, що зменшить кінцевий розмір текстового файлу.

Для різних завдань стемінгу застосовуються різні алгоритми, бо деякі з них опираються більше на суть слова, а інші на максимальне стиснення. Серед них:

а) *Алгоритм пошуку за таблицею флексій*. Флексія – це сукупність морфем, які творять слова, наприклад кінцівки, рідше префікси та суфікси. На основі цих флексій можна створювати нові слова, а точніше інші словоформи, які навіть не згадувалися раніше в тексті, але можуть допомогти у пошуку спільнокореневих чи схожих за сенсом слів;

б) *Алгоритм усічення кінцівок*. Цей алгоритм застосовує базу даних чи просту таблицю кінцівок конкретної обраної мови і зберігає морфологічну основу слова. Однак ця основа не мусить збігатися із етимологічним коренем слова. За необхідністю, можна зберігати усічені кінцівки;

в) *Алгоритм обробки афіксів*. Цей алгоритм вважається додатком до алгоритму усічених кінцівок, бо після прибирання кінцівок, видаляються префікси та суфікси. Їх також можна, за бажанням, зберігати;

г) *Стохастичні алгоритми*. Немає конкретно виявлених типів цих алгоритмів. Вони опираються на побудові ймовірнісних моделей та самі будують власні таблиці флексій. Частіше ці алгоритми основані на нейронних мережах;

г) *Алгоритми лематизації*. Зазвичай це процеси приведення слова до його леми чи нормальної форми, іноді інфінітиву;

д) *Аналіз методом N-грам*. Це не зовсім алгоритм стемінгу, але все рівно знаходить основу слова або створює якийсь еквівалент йому, але вже опираючись на контекст речення, або навіть тексту.

Серед перших та найвідоміших стемерів є реалізація Портера, яка була розроблена ще у 1980 році і активно доповнюється до сьогодні. Однак стемер Портера лише для англійської мови. На жаль неможливо розробити універсальний алгоритм стемінгу через унікальні правила та винятки різних мов. Тому для кожної мови створюється окремі програми та алгоритми, однак сама методологія створення алгоритмів може не сильно відрізнятися, наприклад використовувати алгоритм усічення кінцівок.

2.3 Стемер англійської мови

У розділі 3 цієї роботи будуть використовуватися для прикладів англомовні тексти, тому розглянимо як будується стемер англійської мови, дотримуючись загальних лінгвістичних правил.

Варто зазначити, був обраний алгоритм усічення кінцівок, який був описаний у підрозділі 2.2, а саме алгоритм Портера[14].

Перед першими кроками необхідно здійснити деякі визначення для кожного слова перед стемінгом:

- Визначення *голосної* як однієї з: *«a, e, i, o, u, y»*;
- Визначення *дублю* як одного з: *«bb, dd, ff, gg, mm, nn, pp, rr, tt»*;
- Визначення *дійсного li-закінчення* як одного з: *«c, d, e, g, h, k, m, n, r, t»*;
- Визначення *R1* як область після першої приголосної, що стоїть за голосною, або кінця слова, якщо такої приголосної немає;
- Визначення *R2* область після першого приголосного, що стоїть за голосною в R1, або кінця слова, якщо такої приголосної немає;
- Визначення короткого складу у слові як *(a)* голосну, за якою стоїть приголосна, крім *w, x* або *y*, перед якою приголосна, або як *(b)* голосну на початку слова, після якої йде приголосна;

- Слово називається коротким, якщо воно закінчується коротким складом та якщо *R1* дорівнює нулю;
- Якщо слово має 2 чи менше літер, то залишити це слово незмінним.
- Якщо на початку є *«'»*, видалити;
- Якщо на початку слова або після голосної є *«y»*, замінити на *«Y»*, а потім заново встановити *R1* та *R2*.

Якщо після попередніх визначень слово проходить далі, тоді вже застосовуються наступні кроки алгоритму Портера:

1) Знайти найдовших суфікс серед *«', 's, 's'»* та видалити його, якщо наявний;
2) Пошук суфіксів:
   a. Знайти найдовший з наступних суфіксів та, якщо належать в R1, то виконати:
      - *«sses»*, замінити на *«ss»*;
      - *«ied»* або *«ies»* та перед ним більше 1 літери, замінити на *«i»*;
      - *«ied»* або *«ies»* та перед ним 1 літера, замінити на *«ie»*;
      - *«s»* та є голосна не перед ним, видалити;
      - *«us»* або *«ss»*, нічого;
   b. Знайти найдовший суфікс з наступних і якщо наявний, то виконати:
      - *«eed»* або *«eedly»* та в *R1*, замінити на «ее»;
      - *«ed»*, *«edly»*, *«ing»* або *«ingly»* та містить голосну в попередній частині слова, видалити, а потім, якщо:
        — закінчується на *«at»*, *«bl»* або *«iz»*, додати *«e»*;
        — закінчується на *дубль*, видалити останню літеру;
        — слово коротке, додати *«e»*;
   c. Замінити суфікс *«y»* або *«Y»* на *«i»*, якщо перед ним приголосна, яка не є першою літерою слова;

3) Знайти найдовший з наступних суфіксів та, якщо належать в *R1*, то виконати:
   - *«tional»*, замінити на *«tion»*;
   - *«enci»*, замінити на *«ence»*;
   - *«anci»*, замінити на *«abli»*;
   - *«entli»*, замінити на *«ent»*;
   - *«izer»* або *«ization»*, замінити на *«ize»*;
   - *«ational»*, *«ation»* або *«ator»* замінити на *«ate»*;
   - *«alism»*, *«aliti»* або *«alli»*, замінити на *«al»*;
   - *«fulness»*, замінити на *«ful»*;
   - *«ousli»* або *«ousness»*, замінити на *«ous»*;
   - *«iveness»* або *«iviti»*, замінити на *«ive»*;
   - *«biliti»* або *«bli»*, замінити на *«ble»*;
   - *«ogi»* та перед ним *«l»*, замінити на *«og»*;
   - *«fulli»*, замінити на *«ful»*;
   - *«lessli»*, замінити на *«less»*;
   - *«li»* та перед ним *дійсне li-закінчення*, видалити;
4) Знайти найдовший з наступних суфіксів та, якщо належать в *R1*, то виконати:
   - *«tional»*, замінити на *«tion»*;
   - *«ational»*, замінити на *«ate»*;
   - *«alize»*, замінити на *«al»*;
   - *«icate»*, *«iciti»* або *«ical»*, замінити на *«ic»*;
   - *«ful»* або *«ness»*, видалити;
   - *«ative»*, видалити, якщо в *R2*;
5) Знайти найдовший з наступних суфіксів та, якщо належать в *R2*, то виконати:

- *«al»*, *«ance»*, *«ence»*, *«er»*, *«ic»*, *«able»*, *«ible»*, *«ant»*, *«ement»*, *«ment»*, *«ent»*, *«ism»*, *«ate»*, *«iti»*, *«ous»*, *«ive»* або *«ize»*, видалити;
- *«ion»* та перед ним *«s»* або *«t»*, видалити;

6) Знайти найдовший суфікс з наступних і якщо наявний, то виконати:
   - *«e»* та в *R2*, або в *R1* і перед яким не короткий склад, видалити;
   - *«l»* та в *R2* та перед ним *«l»*, видалити;
7) Замінити усі наявні *«Y»* на *«y»*.

Для прикладу та перевірки алгоритму, візьмемо два слова «artificial» та «intelligence». Далі, покроково застосуємо стемінг до кожного з цих слів.

Дано слово «artificial»:

1) Кроки 1-4 пропускаємо, бо жодна з умов не співпадає;
2) На кроці 5 «artificial» перетворюється на «artifici»;
3) Кроки 6-7 пропускаємо, бо жодна з умов не співпадає.

В результаті маємо «artifici» – стемінговане слово.

Дано слово «intelligence»:

1) Кроки 1-4 пропускаємо, бо жодна з умов не співпадає;
2) На кроці 5 «intelligence» перетворюється на «intellig»;
3) Кроки 6-7 пропускаємо, бо жодна з умов не співпадає.

В результаті маємо «intellig» – стемінговане слово.

У додатку Б наведені приклади результатів роботи стемеру Портера на деяких схожих слів за написанням та/або сенсом.

З результатів стемінгу можна побачити, що короткі слова майже завжди залишаються незмінними, а довгі та комплексні слова, скорочуються до коротших, а саме до основи слова, іноді до морфологічного кореня, однак в англійській мові це буває рідше.

Під час роботи зі стемінгованими словами може виникнути необхідність у зворотному стемінгу. Зазвичай усі дії відбуваються з кінця на початок, але в англійській мові є кілька відмінностей.

Щоб цей процес був дійсним, необхідно створювати окрему базу зберігання кінцівок, щоб, користуючись ними, можна було відновити початкове слово.

Опираючись на стемер Портера можна розробити наступний зворотний алгоритм стемінгу:

1) Якщо на початку слова або після голосної є *«y»*, замінити на *«Y»*, а потім заново встановити *R1* та *R2*;
2) Додати кінцівку, якщо:
   - *«e»* та в кінці стемінгованого слова *R2*, або *R1* і перед яким не короткий склад;
   - *«l»* та в *R2* та в кінці стемінгованого слова ним *«l»*;
3) Додати кінцівку, якщо:
   - *«ion»* та в кінці стемінгованого слова *«s»* або *«t»;*
   - *«al»*, *«ance»*, *«ence»*, *«er»*, *«ic»*, *«able»*, *«ible»*, *«ant»*, *«ement»*, *«ment»*, *«ent»*, *«ism»*, *«ate»*, *«iti»*, *«ous»*, *«ive»* або *«ize»*;
4) Додати кінцівку, якшо належить до *R1* та:
   - *«tional»* та в кінці стемінгованого слова *«tion»*;
   - *«ational»* та в кінці стемінгованого слова *«ate»*;
   - *«alize»* та в кінці стемінгованого слова *«al»*;
   - *«icate»*, *«iciti»* або *«ical»* та в кінці стемінгованого слова *«ic»*;
   - *«ful»* або *«ness»*;
   - *«ative»* та якщо в *R2*;
   - *«tional»* та в кінці стемінгованого слова *«tion»*;
   - *«enci»* та в кінці стемінгованого слова *«ence»*;
   - *«anci»* та в кінці стемінгованого слова *«abli»*;
   - *«entli»* та в кінці стемінгованого слова *«ent»*;
   - *«izer»* або *«ization»* та в кінці стемінгованого слова *«ize»*;
   - *«ational»*, *«ation»* або *«ator»* та в кінці стемінгованого слова *«ate»*;

- *«alism»*, *«aliti»* або *«alli»* та в кінці стемінгованого слова *«al»*;
- *«fulness»* та в кінці стемінгованого слова *«ful»*;
- *«ousli»* або *«ousness»* та в кінці стемінгованого слова *«ous»*;
- *«iveness»* або *«iviti»* та в кінці стемінгованого слова *«ive»*;
- *«biliti»* або *«bli»* та в кінці стемінгованого слова *«ble»*;
- *«ogi»* та перед ним *«l»* та в кінці стемінгованого слова *«og»*;
- *«fulli»* та в кінці стемінгованого слова *«ful»*;
- *«lessli»* та в кінці стемінгованого слова *«less»*;
- *«li»* та в кінці стемінгованого слова *дійсне li-закінчення*;

5) Замінити суфікс *«i»* на *«y»*, якщо перед ним приголосна, яка не є першою літерою слова;
6) Якщо на початку слова або після голосної є *«y»*, замінити на *«Y»*, а потім заново встановити *R1* та *R2*;
7) Додати кінцівку, якщо:
   - *«eed»* або *«eedly»* та в *R1* та в кінці стемінгованого слова *«ee»*;
   - *«ed»*, *«edly»*, *«ing»* або *«ingly»* та в кінці стемінгованого слова є голосна в попередній частині слова;
8) Додати кінцівку, якщо в кінці стемінгованого слова *R1* та:
   - *«sses»*, замінити на *«ss»*;
   - *«ied»* або *«ies»* та перед ним більше 1 літери, замінити на *«i»*;
   - *«ied»* або *«ies»* та перед ним 1 літера, замінити на *«ie»*;
   - *«s»* та є голосна не перед ним, видалити;
9) Додати *«’, ’s, ’s’»*;
10) Замінити усі наявні *«Y»* на *«y»*;

Хоча з першого погляду здається, що звичайний стемер за алгоритм зворотного стемінгу ідентичні, просто виконується з кінця, це не так. У зворотному стемері пункти 1 і 6 збігаються та повторюються, бо під час виконання пунктів між ними, ці символи можуть змінюватися.

При кожній ітерації зворотного алгоритму стемінгу, додана частина кінцівки видаляється та якщо залишилась якась частина збереженої кінцівки, то алгоритм продовжується. Якщо кінцівка залишається порожньою, то можна закінчувати алгоритм – слово вже в початковому вигляді.

Логічно, коли кінцівки зовсім немає, то зворотний стемінг пропускається, що часто може бути причиною прискореного виконання даного алгоритму.

РОЗДІЛ 3 СТИСНЕННЯ СТЕМІНГОВАНОГО ТЕКСТУ

3.1 Реалізація програми

Перед початком розробки програмного коду, необхідно поставити загальну задачу та мету програми. При стисненні стемінгованого тексту, очевидно вихідний файл буде менший та дійсно стиснений. Однак, при декодуванні, в результаті отримаємо текст, а точніше набір стемінгованих слів у порядку знаходження в початковому тексті. Саме тому необхідно розробити алгоритм, щоб була можливість отримати даний текст при декодуванні без втрат.

Як було указано в розділі 2 цієї роботи, під час стемінгу, за необхідністю, можна додатково зберігати відлучені кінцівки слів разом із стемінгованим словом.

Перед тим, як застосовувати стемінг, необхідно здійснити токенізацію усього тексту. Орієнтуючись на пробіли між словами, робиться сегментація тексту по токенам, що будуть нашими початковими словами, з якими буде відбуватися подальша обробка.

Якщо зберігати і стемінговане слово, і кінцівку разом, то очевидно, що розмір кінцевого файлу буде значно більшим, тому необхідно оптимізувати алгоритм збереження стемінгованих слів разом із кінцівками. Щоб обрати спосіб оптимізації, варто переглянути, які методи може надати середовище розробки.

Для створення програмного коду для реалізації мультироздільникових кодів, стемінгу та зберігання даних, була обрана мова програмування C++ у середовищі «Visual Studio».

Дана мова була обрана через швидкість виконання коду, ефективність при обробці даних, гнучкість та апаратного доступу. В загалом, C++ – це мова

для загального призначення із об’єктно орієнтованим програмуванням та походить із сімейства «C».

Ця мова складається з двох основних компонентів:

– Пряме відображення функцій;
– Абстракція із нульовими витратами, основуючись на даних зіставленнях;

У програмуванні існує поняття хеш-таблиць. Вони являють собою структуру даних, яка реалізує асоціативний масив, створений на типі абстрактних даних. Основною унікальністю хеш-таблиць є наявність індексації та використання хеш-функцій для стиснення даних.

Хешування – це процес перетворення даних будь-якого розміру в дані обраного фіксованого розміру. Зазвичай кінцеве значення хешованих даних – це набір символів та знаків, які не несуть сенсу без хеш-функції. У цій функцій вкладений алгоритм кодування та декодування, зазвичай завдяки спеціальному ключу хешування.

У хеш-таблицях стиснення за допомогою хешування дає змогу зберігати в статичному розмірі усі дані, в нашому випадку слова, незалежно від їх розміру. Також у комірці хеш-таблиці може бути не звичайна текстова змінна, а й цілий масив. Мова C++ позволяє працювати з хеш-таблицями.

В нашому випадку необхідно обрати спосіб зменшити обсяг даних, що будемо зберігати. Тому будемо зберігати унікальні слова в окремому текстовому масиві.

У таблиці 1 показано, як буде зберігатися кожне стемінговане слово. В першій позиції масиву буде саме стемінговане слово, на другій позиції буде відлучена кінцівка початкового слова, а на третій – масив чисел, який вказує, на яких позиціях в тексті були початкові слова. Тобто, якщо абсолютно однакове слово зустрічалося в тексті кілька разів, він буде збережений у один масив. Таким чином, хеш-таблиця матиме менше елементів в собі, ніж

початковий текст. Кожен такий масив буде оброблений хеш-функцією та матиме фіксований розмір.

Стемінговане слово	Кінцівка слова	Позиція в тексті
`artifici`	`al`	{5, 150, 363}
`consist`	``	{19, 266}
`consist`	`ently`	{102, 355}

Таблиця 1 – Масив стемінгованого слова

У мові програмування C++ масиви являються статичними змінними, а щоб можна було під час обробки тексту поповнювати масив позицій в тексті необхідний динамічний масив. Тому я буду використовувати двохзв’язний список типу «list». Це абстрактних тип даних, який являється списком впорядкованих даних. Цими даними можуть бути як звичайна змінна, так і масив або клас.

Тому третім елементом масиву стемінгованого слова буде об’єкт типу «list», який зберігає усі числа в порядку додавання у список. Також у цей список дані будуть вноситись послідовно, тому в додатковаму сортуванні немає необхідності.

Коли дані декодуються та перетворюють з хешу до масиву стемінгованого слова, то необхідно скомпонувати перший та другий елементі масиву до початкового слова, щоб був без втрат. Для цього необхідно застосувати алгоритм, який застосовували для стемінгу цих слів, а саме стемер Портера, але у зворотному порядку. Тобто проходити кроки алгоритму починаючи з кінця. Якщо на якомусь кроці є згадка про кінцівку або її частину даного стемінгованого слова, то виконується зворотна дія. Якщо вказано видалити, тоді треба додати цю кінцівку; якщо замінити, то замінити елемент зі стемінгованого слова на кінцівку.

Наприклад стемінговане слово «artificі» разом з його кінцівкою «al»: у кроках 7 та 6 немає ніяких випадків, де збігаються видалені кінцівки або склади *R1* та *R2* у стемінгованому слові. На кроці 5 є випадок для кінцівки «al». На цьому кроці вказано, що видаляти цю кінцівку за простою її наявністю. Тому зворотна дія буде – просто додати цю кінцівку. В наступних кроках як і немає випадків для змін слова, так і немає доступних збережених кінцівок. Отже таким чином здійснюється зворотний стемінг слову, а точніше відтворення початкового вигляду слова.

Так як занесення стемінгованих слів до хеш-таблиці відбуватиметься послідовно з кожним словом тексту, тому вивід слів з хеш-таблиці буде відбуватися послідовно. Однак треба зауважити, що необхідно слідкувати за третім елементом масиву стемінгованого слова, тому що там може бути кілька значень, тому варто видаляти використані позиції при виведенні стисненого тексту. А коли усі позиції в тексті використані, то просто видалити увесь масив з метою оптимізації.

Отже, алгоритм стиснення виглядає наступним чином:

1) Зчитування текстового файлу;
2) Токенізація тексту;
3) Стемінг токенів (слів);
4) Занесення стемінгованих слів до хеш-таблиці;
5) Застосування мультироздільникового коду до хеш-таблиці;
6) Збереження;

Алгоритм зчитування стисненого тексту виглядає наступним чином:

1) Зчитування стисненого тексту з файлу;
2) Декодування стисненного тексту;
3) Занесення декодованих даних до хеш-таблиці;
4) Зворотний стемінг кожного елементу хеш-таблиці;
5) Занесення слів до вказаної позиції;

Дані алгоритми за послідовністю та побудовою виглядають як будь-який інший алгоритм для збереження текстових даних. Тому ці алгоритми можна винести як окремий текстовий формат. Однак, кожен файл конкретного текстового формату мусить мати на початку одну і ту же комбінацію та/або назву формату, щоб можна було ідентифікувати програмі зчитування, який алгоритм застосовувати.

3.2 Дослідження результатів

Алгоритми у підрозділі 3.1 були застосовані до текстів різних розмірів та жанрів. Метою цих експериментів було дослідження впливу кількості та різноманітності слів на якість стиснення та збереження даних.

Під час виконання програми стиснення були отримані наступні часові результати (таблиця 2). Тестування проводилось на текстовому файлі розміром 800 тисяч слів. Загальний час кодування близько 20-25 секунд, а час декодування 9-10 секунд.

Подія	Час, с
Зчитування тексту та токенізація	10
Стемінг та додання до хеш-таблиці	10
Створення конструкції кодових слів	5
Кодування	0,02
Декодування	0,04
Зворотний стемінг	9

Таблиця 2 – Результати виконання роботи у часі (800 000 слів)

Якщо розглядати ці часові дані як загальному користувачу, то дійсно, часу витрачено забагато. Проте, не всі етапи повинні виконуватися відразу. Деякі події можуть виконуватися паралельно із роботою над текстом.

Наприклад, коли користувач пише свій текст, то з кожним новим словом відбувається одна ітерація стемінгу та занесення результату до хеш-таблиці. У цьому випадку збереження файлу займатиме лише 5 секунд. Тобто, коли користувач пише слово, то вже пропускається момент зчитування тексту, бо відкритий текстовий файл вже зчитаний і знаходиться в оперативній пам’яті, та токенізація, бо з кожним натисканням пробілу, можна враховувати, що це токен (слово). А у випадку, коли користувач видаляє слово, то не проблема видалити зі списку слів за допомогою методів мов програмування, які вже вважаються оптимізованими та витрачають мінімум часу для здійснення певної дії.

Результати стиснення даних, що наведені на рисунку 3, показують розмір стисненого файлу, розмір бібліотеки (кількість слів) та кількість унікальних слів.

```
creating library...
100000 200000 300000 400000 500000 600000 700000 Number of words: 790018
Size of library: 9262
L=2; Cn=0 s=0
L=3; Cn=1 s=0
L=4; Cn=3 s=0
L=5; Cn=7 s=0
L=6; Cn=14 s=0
L=7; Cn=26 s=0
L=8; Cn=46 s=0
L=9; Cn=79 s=0
L=10; Cn=133 s=0
L=11; Cn=221 s=0
L=12; Cn=364 s=0
L=13; Cn=596 s=0
L=14; Cn=972 s=0
L=15; Cn=1581 s=0
L=16; Cn=2567 s=0
L=17; Cn=4163 s=0
L=18; Cn=6746 s=0
L=19; Cn=10926 s=0
L=20; Cn=17690 s=0
L=21; Cn=28635 s=0
entropy=8.46029
R=-nan(ind)  formed!!!!!!!!!!!!!!!!!!!!!!!! 0 0 0 100000 200000 300000 400000 500000 500000 rank[0]=1943
Text to ranks donesize: 790018
different words: 9262

I235 bytes: 648008
I235 av codeword length: 8.83449
entropy=8.46029
S=223 scdc_L= 684979 SCDC av codeword length: 9.33853
s1=0 s2=0 r1=0 r2=0 r3=0 r4=0
decode_SCDC time: 0.022492 scdc_L=684979 decode_C_fast time: 0 decode_I_fast time: 0.0422563

 uu=13690800 27381600 98554
```

Рисунок 3 – Результат виконання програми

У додатку В наведений частковий результат стисненого файлу у бітовому вигляді. Так, зберігання окремих одиниць та нулів займатиме в рази більше місця, а в реалі одне число займає 1 біт, а якщо конвертувати ці бітові значення в юнікод, то можна отримати символи та числа, які репрезентують стисненні дані за допомогою мультироздільникових кодів та функцій хешування.

Для більше детальної оцінки результатів, були обрані кілька десяток книг різних жанрів та обсягом. Як помітно за таблиці 3, опираючись на обсяг текстів, ефективність стиснення не сильно різниться, хоча можна зробити поспішний висновок, що тексти з більшим обсягом мають краще стиснення.

Розмір тексту	Ефективність стискання
Малий (до 250 000 слів)	4%
Середній (250 000 - 1 000 000 слів)	3%
Великий (більше 1 000 000 слів)	6%

Таблиця 3 – Ефективність стискання за розміром тексту

З таблиці 4 можна побачити порівняння за жанрами. З даної таблиці можна дійти висновку, що ефективність стискання даних залежить від жанрової належності.

Жанр тексту	Ефективність стискання
Художня література	6.5%
Наукова література	3%
Історична література	3.5%

Таблиця 4 – Ефективність стискання за жанром тексту

Однак, після об’єднання обох результатів та більш детального огляду вхідних даних (тобто текстів), я дійшов висновку, що ефективність стискання даних залежить від кількості унікальних слів. Чим менше унікальних слів, тим менше елементів у хеш-таблиці, тим менше займатиме місця кінцевий файл.

А якщо розглядати жанр літератури, то дійсно у більш наукових текстах йде менше повторень слів, ніж у художній літературі. Цей фактор також може бути ключовим для сфер застосування цього алгоритму.

Також цей метод був перевірений на наборах чисел та випадкових даних. Так як числа не піддаються стемінгу, то ця частина автоматично ігнорується. Для зберігання великих чисел у хеш-таблиці також немає сенсу, бо їхня довжина може вийти за рамки середньої довжини слова і майже дублюються. Тому у економічних, математичних та аналітичних сферах цей алгоритм буде неактуальним. В додаток цього можна сказати, що у вищезгаданих сферах використовуються свої локальні методи зберігання даних, а саме бази даних та матричні таблиці.

Звідси випливає питання, чи є ентропія стисненого та хешованого тексту кращою за ентропію тексту и чистому вигляді. Після проведення кількох додаткових перевірок та експериментів, були отримані певні результати (таблиця 5), що показують ефективність.

Метод	Показник ентропії
MD-код	8.9%
MD-код + хешування	8.9%
MD-код + стемінг + хешування	8.4%

Таблиця 5 – Показники ентропії різними методами

Так як хешування виділяє статичний простір для збереження даних, незалежно від початкового розміру. Саме тому в середньому виходить, що

довжина хешованого слово дорівнює середній довжині слова в тексті, а точніше цілій мові.

Під час стемінгу слова скорочуються та іноді виникають випадки, що загальний обсяг стемінгованого слову із кінцівкою займають менше місця ніж початкове слово, і саме це є фактором пониженої ентропії.

З цього робиться логічний висновок, що у цьому випадку стемінг дійсно необхідний, хоча на менших обсягах тексту різниці та сенсу використання фактично немає.

ВИСНОВКИ

Після проведення кінцевих досліджень та аналізів, можна побачити, що алгоритм стиснення текстових даних, розроблений у цій роботі, має потенціал стати альтернативою для текстового формату «.txt», так як зберігає чистий текст без стилів, розмірів, кольорів тощо.

Із додатковою обробкою цього алгоритму та при правильному підведенню до стандарту розширення, дійсно можна вивести формат текстових файлів, який опиратиметься на мультироздільникові коди та стемінг тексту.

У майже всіх випадків, цей алгоритм стискає дані та займає менше місця при збережені. Однак, є декілька недоліків. Через стемер на виході отримується текст повністю у нижньому реєстрі. Цю проблему можна розв’язати, переробивши алгоритм стемінгу. Також алгоритм стискання, що був розроблений у цій роботі, не показує ефективності, а може навіть займати більше місця, коли дуже малі текстові дані, тобто текст складається з кілька десятків слів. Окрім цього, якщо відбувається робота з числами, або йдеться зберігання одного цілого рядка (слова з кілька сотень символів), то алгоритм не функціонує як необхідно.

Отже, можна дійти висновку, що алгоритм стиснення стемінгованих текстів, реалізований на мультироздільникових кодах, чудово працює із творами, книгами, художньою та науковою літературами, однак абсолютно не ефективний на базах даних з числами та довгими рядками символів.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. I. Zavadskyi, A. Anisimov; “Reverse Multi-Delimiter Compression Codes,” Data Compression Conference Proceedings, 910587 – pp. 173–182, 2020.
2. D. Huffman, “A method for the construction of minimum-redundancy codes,” Proc. IRE, vol. 40, pp. 1098–1101, 1952.
3. R. Capocelli, A. D. Santis, “Regular universal codeword sets,” IEEE Transactions Information Theory, vol. 32, no. 1, pp. 129–133, 1986.
4. I. Zavadskyi, A. Anisimov, “A family of data compression codes with multiple delimiters,” in Proc. Prague Stringology Conference, pp. 71–84, 2016.
5. M. Bates, "Models of natural language understanding", Proceedings of the National Academy of Sciences of the United States of America, 92 (22): pp. 9977–9982, 1995.
6. Leon E. Dostert, “The Georgetown–I.B.M. experiment”, 124–135, 1955.
7. J. Searle, "Minds, Brains and Programs", Behavioral and Brain Sciences, 3(3): 417–457, 1980.
8. Y. Goldberg, "A Primer on Neural Network Models for Natural Language Processing", Journal of Artificial Intelligence Research. 57: 345–420, 2016.
9. G. Filip, J. Krzysztof, W. Agnieszka, W. Mikolaj, “Text Normalization as a Special Case of Machine Translation,” Proc. of the International Multiconference on Computer Science and Information Technology; 51–56, 2006.
10. G. A. Miller, “The cognitive revolution: A historical perspective”. Trends in Cognitive Sciences. 7 (3): 141–144, 2006.
11. C. Truesdell, “The Tragicomical History of Thermodynamics” 1822–1854, Springer, New York, ISBN 0-387-90403-4, pp 215, 1980.
12. J. L. Dawson, “Suffix Removal for Word Conflation”, Bulletin of the Association for Literary and Linguistic Computing, 2(3): 33–46, 1974.

13. J. B. Lovins, “Development of a Stemming Algorithm”, Mechanical Translation and Computational Linguistics, 11, 22—31, 1968.
14. M. F. Porter, “An Algorithm for Suffix Stripping”, Program, 14(3): 130–137, 1980.
15. B. Stroustrup, “The C++ Programming Language (Third ed.),” ISBN 0-201-88954-4, 1997.
16. D. Knuth, “The Art of Computer Programming. 3: Sorting and Searching (2nd ed.),” Addison-Wesley. pp. 513–558, ISBN 978-0-201-89685-5, 1998.
17. E. Reingold, J. Nievergelt, D. Narsingh, “Combinatorial Algorithms: Theory and Practice”, Englewood Cliffs, New Jersey: Prentice Hall. pp. 38–41, ISBN 0-13-152447-X, 1977.
18. J. Lewis, “Computer Science Illuminated. Jones and Bartlett”, ISBN 0-7637-4149-3, 2006.

ДОДАТОК А
Псевдокод алгоритму декодування MD-коду

Вхід: кодований текст.

Результат: індекси декодувати слова в масиві слів.

```
1 i ← 0, j ← 0, p ← 0
2 while i < length of encoded text
3       (p1, p2, p3, p4, j) ← TABj[Text[i]]
4        p ← p + p1
5       if p2 ≥ 0
6               output (p)
7               if p3 ≥ 0
8                       output (p2)
9                       if p4 ≥ 0
10                              output (p3)
11                              p ← p4
12                      else
13                              p ← p3
14              else
15              p ← p2
16      i ← i + 1
```

ДОДАТОК Б
Приклади роботи стемеру Портера

Слово – Стемінг		Слово – Стемінг		Слово – Стемінг	
consign	consign	conspicuous	conspicu	knees	knee
consigned	consign	conspicuously	conspicu	knell	knell
consigning	consign	conspiracy	conspiraci	knelt	knelt
consignment	consign	conspirator	conspir	knew	knew
consist	consist	conspirators	conspir	knick	knick
consisted	consist	conspire	conspir	knif	knif
consistency	consist	conspired	conspir	knife	knife
consistent	consist	conspiring	conspir	knight	knight
consistently	consist	constable	constabl	knightly	knight
consisting	consist	constables	constabl	knights	knight
consists	consist	constance	constanc	knit	knit
consolation	consol	constancy	constanc	knits	knit
consolations	consol	constant	constant	knitted	knit
consolatory	consolatori	knack	knack	knitting	knit
console	consol	knackeries	knackeri	knives	knive
consoled	consol	knacks	knack	knob	knob
consoles	consol	knag	knag	knobs	knob
consolidate	consolid	knave	knave	knock	knock
consolidated	consolid	knaves	knave	knocked	knock
consolidating	consolid	knavish	knavish	knocker	knocker
consoling	consol	kneaded	knead	knockers	knocker
consolingly	consol	kneading	knead	knocking	knock
consols	consol	knee	knee	knocks	knock
consonant	conson	kneel	kneel	knopp	knopp
consort	consort	kneeled	kneel	knot	knot
consorted	consort	kneeling	kneel	knots	knot
consorting	consort	kneels	kneel	lie	lie

ДОДАТОК В
Частина бінарного результату стиснення

«1111111111100000100110000010011011110100001110100001111101000011
1010000111000001111110000011110001010000110110100100101000110110
0000011000000111111101001111110100111111111100000101111110000010
1111111111100000011111110100111111010011111111101001110100111001
0011100011001000000111011100100110000001100100000011110000000111110
0100000011100100000011110111110110000001100010000110111100100000111
0010000001111111011111111100100000011100100000011101111111111111011
0111111111111111111111110000000111110000000110010001110100111001001
1100011101110001010101010110111111100101001111001010011111111011
0110010000111110111111110111111011111001000111001000111100010101
0101011000101010101011011111000011100001111111111111010001111000
1010101010110001010101010111111011111000011100001111111111100010
1010101011000101010101011011111011000001101010111110001010101010
1100010101010101111011011111111111111111111100010100101100010100
10110010001110100111111110100011111101000111000000100110000010011
1000111001111001011010100111011001000011111110011111100111101010
1010110111110110000011010101111111100111111001110111111111111111
1111101110000000111101111101100000011000100001110100111101000011
10100001111111100011000000000111111110000000111110000000001001111
00000001001111001010011101111010101001110101010011111111110011010
10100111100001011001111100000001001111110101011100101001110100110
11110101001010011000000111101000011101000011101111101100000011010
10111110100001110100001110010000001100000000011111111000000110111
00000001001111001010011110000101100111110000000100110111111111111
10011111111100110101010011111110101011100101001110000011011110101
0010100111000010110011111000000100110111000000111101111110110000
0110…»