КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА

Факультет комп'ютерних наук та кібернетики
Кафедра математичної інформатики

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

(підпис)

«__»_____20__р.

Дипломна робота
на здобуття ступеня бакалавра

за спеціальністю 122 Комп'ютерні науки
на тему:

ЗАСТОСУВАННЯ ЧАСТОТНО-ПОЗИЦІЙНОГО АНАЛІЗУ ДЛЯ СТИСНЕННЯ ПРИРОДНОМОВНИХ ТЕКСТІВ

Виконав студент 4-го курсу
Стречень Матвій Володимирович _____
(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 46 сторінок, 8 лістингів, 1 алгоритм, 6 рисунків, 9 таблиць, 8 джерел та посилань.

КОДУВАННЯ ДАНИХ, СТИСКАННЯ ДАНИХ, РЕВЕРСИВНІ МУЛЬТИРОЗДІЛЬНИКОВІ КОДИ, АНАЛІЗ ПРИРОДНОМОВНИХ ТЕКСТІВ

Об’єктом роботи є алгоритми стиснення природномовних текстів. Предметом роботи є реверсивні мультироздільникові коди та RPBC-коди.

Метою дослідження є тестування гіпотез і створення покращених версій алгоритмів стискання, призначених для стиснення природномовних текстів.

Методи розроблення: інтегроване середовище розробки для мови програмування Python PyCharm (включно з інтерпретатором CPython). У якості метода дослідження виступало експериментальне порівняння стискальної ефективності розроблених варіацій алгоритму з існуючими, між собою та з теоретично можливою межею на наборах файлів Проєкту Гутенберг.

Результати роботи: виконано загальний огляд вже існуючих реалізацій алгоритму, побудовано власні реалізації, зроблено аналіз отриманих результатів роботи алгоритму, результати проілюстровано відповідними таблицями.

Отримані результати можна використати як фундамент для гіпотез щодо ймовірних напрямків подальшого покращення стискальної ефективності при роботі з природномовними текстами, а описані методи можуть бути застосовані для оптимізації текстів з властивостями, подібними до властивостей природномовних текстів.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ

англ. – англійською

$\mathbb{N}$ - множина натуральних чисел

N – кількість елементів у множині можливих вхідних даних алгоритму кодування

|S| - кількість елементів у скінченній множині S

RPBC – скорочена назва до Enhanced Byte Codes with Restricted Prefix Properties

РМК – скорочена назва реверсивних мультироздільникових кодів

ВСТУП

Оцінка сучасного стану об’єкта розробки.

Стрімке збільшення об’ємів даних у світі мотивує шукати нові алгоритми кодування для того чи іншого набору обмежень. До прикладу, ЦЕРН генерує [1] 115 петабайт даних щорічно. Мережа Інтернет генерує такий самий об’єм даних щосекундно. За таких умов гостро стоїть питання стиснення даних, однак одного лише коефіцієнту стискання недостатньо для порівняння алгоритмів стиснення.

При цьому варто виокремлювати різні формати, що в них представлена інформація. Оптимальні алгоритми стиснення для зображення чи відеоряду сильно відрізняються від тих, що застосовуються до текстових файлів чи документів. Більше того, навіть для конкретно текстових документів окремі підходи можуть краще проявляти себе, якщо заздалегідь відомо, що вмістом є суто природномовний текст, а не, скажімо, таблиця у форматі comma-separated values (значення, записані через кому). Таким чином, коректним є твердження: що більше заздалегідь відомо про представлення інформації в кодованій послідовності, то краще можна підібрати алгоритм, який би найефективніше стискав цей конкретний формат представлення даних.

Коефіцієнт стиснення – основна характеристика алгоритму стискання. Якщо розмір вхідного алфавіту – степінь двійки, найкраще (відносно максимального ступеню стиснення, тобто двійкової ентропії) проявляють себе коди Хафмана [2]. У інших випадках з нині відомих найвищу ступінь стискання мають арифметичні коди.

Актуальність роботи та підстави для її виконання.

Існуючі алгоритми стиснення при роботі з природномовними текстами типово ігнорують особливості конкретної природної мови. Окремі наукові роботи роблять крок у напрямку частотного аналізу текстів. Так, у своїй статті El Daher, A. та Connor, J. [3] розглядають статистичну модель для англійської мови і використовують коди Хафмана для стискального алгоритму, однак

ігнорують контекст слів, беручи до уваги виключно частотну характеристику навчальної вибірки.

Однак, природні мови хоч і досить різноманітні за своєю структурою, все ж володіють деякими характеристиками, які можна використати при стисненні текстів. По-перше, природномовний текст зазвичай чітко розбивається на синтаксичні одиниці – речення, які можуть бути піддані аналізу незалежно. По-друге, в певних мовних групах речення зазвичай має досить чітку структуру (як-то у германських і романських мовах), що впливає на частоту потрапляння слів на певні місця у реченні.

При цьому, лише певний аналіз текстів має мало сенсу, якщо не розглядати його у парі з конкретними алгоритмами стиснення.

Позаяк коди Хафмана та арифметичні коди не є оптимальними в жодній з вищезгаданих категорій (алгоритми кодування досить неефективні, файли-результати достатньо однорідні і не мають чітко вираженої структури, що заважає пошуку, а про стійкість до перешкод мова не йде - навіть один інвертований біт може привести до неправильного розшифрування усього подальшого тексту), альтернативні алгоритми кодування все ще розглядаються за певних обмежень.

Реверсивні мультироздільникові коди, розглянуті у статті [4] від 24 лютого 2017 року за авторства Анісімова А.В., Завадського І.О. гарно проявляють себе у якості компромісу між ефективністю стиснення та вищезгаданими характеристиками пошуку та складності декодування.

Крім того, існують варіації алгоритмів стискання, які працюють на рівні байтів (а не окремих бітів), що сильно покращує швидкодію як кодування, так і декодування. Так, хоч коефіцієнт стиснення в RPBC [5] і поступається реверсивним мультироздільниковим кодам (вже не кажучи про вищезгадані коди Хафмана і арифметичні коди), значно виграє у швидкодії як кодування, так і декодування, а тому також буде розглянутий в рамках цієї роботи.

Мета й завдання роботи. Метою дослідження є тестування гіпотез і створення покращених версій алгоритмів стискання, призначених для стиснення природномовних текстів.

- Дослідити теоретичну базу існуючих алгоритмів;
- Дослідити існуючі реалізації алгоритмів стискання природномовних текстів;
- Розробити альтернативні реалізації алгоритму кодування-декодування природномовних текстів, порівняти результати стискання з існуючими
- Зробити висновки щодо ефективності тих чи інших напрямків покращення алгоритмів стиснення природномовних текстів.

Об'єкт, методи й засоби розробки. Об’єктом даного дослідження є алгоритм декодування реверсивних мультироздільникових кодів.

Методи розроблення: інтегроване середовище розробки для мови програмування Python PyCharm (включно з інтерпретатором CPython).

У якості метода дослідження виступало експериментальне порівняння стискальної ефективності розроблених варіацій алгоритму з існуючими, між собою та з теоретично можливою межею на наборах файлів Проєкту Гутенберг.

Можливі сфери застосування. Результати виконання роботи можуть бути використані як для розробки безпосередньо програмного забезпечення для стискання природномовних текстів, так і для подальших досліджень у напрямку використання особливостей природних мов для покращення стискальної ефективності алгоритмів.

Взаємозв’язок з іншими роботами. Роботу було виконано на базі та в порівнянні з уже існуючими реалізаціями алгоритмів декодування реверсивних мультироздільникових кодів та RPBC при стисканні природномовних текстів.

Крім того, даній роботі передувала курсова робота, присвячена оптимізації алгоритму декодування реверсивних мультироздільникових кодів. Згадане дослідження дає підстави вважати реверсивні мультироздільникові коди гарним компромісом між швидкодією, стискальною ефективністю, толерантністю до похибок та іншими важливими характеристиками при стисненні природномовних текстів.

РОЗДІЛ 1
АНАЛІЗ ІСНУЮЧИХ РЕАЛІЗАЦІЙ АЛГОРИТМУ ТА АНАЛОГІВ

1.1 Загальний підхід до кодування даних

У більшості випадків алгоритми кодування приймають на вхід послідовність натуральних чисел. З цією метою текст, що кодується, має бути відображений у таку послідовність.

Зазвичай для алгоритмів кодування розглядають два типи відображень:

- F: {множина символів} → {1, 2, …, N}, N ∈ ℕ
- F: {множина слів} → {1, 2, …, N}, N ∈ ℕ

Для кодування природомовних текстів (а також текстів з обмеженою кількістю слів та сильно нерівномірним розподілом частот слів, наприклад, програм) використовується саме другий тип відображень.

У роботі використовується саме він, тому спершу для текстів-тестів необхідно було провести попередню обробку, щоб отримати потрібне відображення. Ця передобробка включала в себе два етапи.

Перший етап – виділення окремих слів. У якості роздільника (по якому відбувалась розбивка на слова) при цьому використовувалися виключно пробіли, тобто переноси на новий рядок «склеювались» з попереднім чи наступним словом. Якщо кілька пробілів йшли підряд, у якості кодового слова використовувася порожній рядок, що йшов між ними. До прикладу, при розділенні рядка «слово, інше\n та третє» (між «та» та «третє» два пробіли, «\n» позначає символ переносу рядка) ми отримали б наступний набір слів-токенів: «слово,», «інше\n», «та», «», «третє».

Другий етап зазвичай являє собою простий підрахунок частотності кожного слова, беззалежно від контексту. Після цього встановлюється бієкція між унікальними словами-токенами з тексту та натуральними числами таким чином, щоб меншим числам відповідали частотніші слова, а більшим – слова,

що рідше зустрічались. У даній роботі цей етап передобробки був розширений, що дозвовлило показати певні покращення в стискальній ефективності текстів.

Тепер вхідний текст з допомогою отриманої бієкції перетворюється на послідовність чисел, що і буде аргументом алгоритму кодування. Результатом алгоритму буде набір байт – закодований текст.

Декодування відбувається у зворотному порядку. Спершу отримуємо послідовність натуральних чисел, застосовуючи алгоритм декодування до закодованого тексту. Отримавши послідовність чисел, з допомогою тієї самої бієкції відновимо початкову послідовність слів. Оскільки дана робота передбачає внесення змін до другого етапу передобробки/кодування, то і відповідний етап декодування буде певним чином розширений. Конкретні зміни як до алгоритму кодування, так і до алгоритму декодування будуть розглянуті у подальших розділах.

1.2 Реверсивні мультироздільникові коди

Сімейство кодувань $R_{m_1,\dots,m_t}$ (реверсивні мультироздільникові кодування) задаються наступним чином.

Нехай маємо зростаючу послідовність чисел $M = m_1, m_2, \dots, m_t$. Тоді множина слів кодування $R_{m_1,\dots,m_t}$ складається зі всіх слів вигляду $01^{m_i}\ \forall m_i \in M$, а також усіх двійкових послідовностей, що задовольняють наступні умови:

- слово починається з префіксу $01^{m_i}\ \forall m_i \in M$
- слово не закінчується на $01^{m_i}\ \forall m_i \in M$
- слово не містить підрядка $01^{m_i}\ \forall m_i \in M$

Стискальна ефективність алгоритму при цьому напряму залежить від вибору множини М. Крім того, для ліпшої стискальної ефективності слід забезпечити відповідність частотніших слів тексту коротшим двійковим послідовностям з множини $R_{m_1,\dots,m_t}$. Щоб забезпечити це, слід у явному вигляді побудувати цю множину (а точніше, деяку підмножину з найменших за довжиною кодів). Для цього застосуємо прийом інкрементальної побудови.

Будемо вважати, що підмножина, що складається з кодів довжини до L – 1 вже побудована і ми намагаємось додати до множини усі слова довжиною L. Використаємо допоміжну множину K, що містить усі невід'ємні числа, що не перевищують $m_t + 1$ і при цьому не містяться в M. Тоді виконаємо три прості кроки для переходу на наступний рівень:

1. Для кожного $k_i \in K$ додамо до множини кодових слів усі слова довжини $L - k_i - 1$ у конкатенації з суфіксом 01^{k_i}
2. Додамо до множини всі слова довжини L – 1 з суфіксом $01^r, r > m_t$, додавши до них "1"
3. Якщо $L - 1 \in M$, то додамо 01^{L-1} до набору.

Маючи кодові слова, упорядковані за зростанням довжини, можемо закодувати слова вхідного тексту за їх частотою у самому тексті.

Як вже було сказано, стискальна ефективність алгоритму сильно залежить від множини M. В рамках цієї роботи буде розглянуто різні набори параметрів M та проведено аналіз стискальної ефективності в залежності від емпіричного розподілу частот кодових слів, а також від безпосередньо вибору множини M.

В таблиці 1 наведено найменші (довжини, що не перевищує 6) кодові слова (роздільники та суфікси) для кодувань $R_{2,3,5}$, $R_{2,\infty}$ та $R_{2,4,\infty}$. Тут запис $R_{m_1,\dots,m_t,\infty}$ є скороченим позначенням до $R_{m_1,\dots,m_t,(m_t+1),(m_t+2),\dots}$, тобто крім скінченного набору роздільників $01^{m_1}, 01^{m_2}, \dots, 01^{m_t}$ додається нескінченний набір роздільників $01^{m_t+1}, 01^{m_t+2}, \dots$

Довжина	$R_{2,3,5}$		$R_{2,\infty}$		$R_{2,4,\infty}$	
	Роздільник	Суфікс	Роздільник	Суфікс	Роздільник	Суфікс
3	011		011		011	
4	011	0	011	0	011	0
	0111		0111			

5	011 011 0111	01 00 0	011 011 0111 01111	00 01 0	011 011 01111	00 01
6	011 011 011 0111 0111 011111	000 010 001 00 01	011 011 011 0111 0111 01111 011111	000 001 010 00 01 0	011 011 011 01111 011111	000 010 001 0 0

Таб. 1

Як видно з наведеної таблиці, від вибору множини M залежить кількість кодових слів малої довжини – чим більше роздільників, тим більше коротких кодових слів. Однак у більшої кількості роздільників є обернений ефект, який виражається у меншій варіативності довгих кодових слів. Як буде показано у роботі, при більш рівномірних розподілах (коли найчастотніше слово зустрічається порівняно рідко, а сам розподіл має пологий «хвіст»), має сенс давати більшу варіативність довшим кодовим словам, жертвуючи кількома кодовими словами невеликої довжини.

Важливою перевагою реверсивних мультироздільникових кодів є також достатньо простий (щодо обчислювальної складності та загальної швидкодії) алгоритм декодування.

Маючи множину $R_{m_1,\dots,m_t}$, можна побудувати скінченний автомат для декодування. Природньо, що реалізації скінченного автомату будуть відрізнятися в залежності від множини M, однак для кодів з мінімальним роздільником 011 є уніфікований алгоритм вирівняного по байтам декодування. Для його реалізації потрібна передпідрахована таблиця TAB, яка фактично буде виконувати роль матриці переходів скінченного автомату.

1. $i \leftarrow 0, j \leftarrow 0, p \leftarrow 0$

```
2.  ПОКИ i < довжини закодованого тексту:
3.      (p1, p2, p3, p4, j) ← TAB_j[ТЕКСТ[i]]
4.      p ← p + p1
5.      ЯКЩО p2 ≥ 0:
6.          ВИВЕСТИ(p)
7.          ЯКЩО p3 ≥ 0:
8.              ВИВЕСТИ(p2)
9.              ЯКЩО p4 ≥ 0:
10.                 ВИВЕСТИ(p3)
11.                 p ← p4
12.             ІНАКШЕ:
13.                 p ← p3
14.         ІНАКШЕ:
15.             p ← p2
16.     i ← i + 1
```

Алгоритм 1

Слід прокоментувати рядок 3 наведеного алгоритму. Може виникнути питання – чому чотирьох значень p (тобто p_1, p_2, p_3, p_4) достатньо? Чи може виникнути ситуація, коли все-таки знадобиться 5 значень? У загальному випадку така ситуація дійсно може виникнути, але як було згадано раніше, ми розглядаємо лише коди, у яких мінімальний роздільник має довжину три. Відповідно, на один байт у гіршому випадку може потрапити 1 біт попереднього кодового слова, два кодових слова довжиною три та 1 біт початку наступного кодового слова – таким чином, кодових слів, що перетинаються з конкретним байтом щонайбільше 4.

Щодо таблиці TAB – її розмір пропорційний кількості біт у максимальному за довжиною кодовому слові. До прикладу, якщо кількість слів у словнику досягає 288 тис. слів, довжина кодових слів не перевищує 34 біти, а тому для збереження таблиці TAB достатньо 600 Кб пам’яті – мізерні, по сучасним мірким, об’єми.

Значними перевагами реверсивних мультироздільникових кодів є:

- Гарна стискальна ефективність при нерівномірних розподілах частот кодових слів
- Непогана швидкодія кодування і декодування

- Наявність чітко виражених роздільників, що робить можливість пошук по шаблону
- Глобальна толерантність до помилок. Інвертований біт у закодованій послідовності хоч і дещо зіпсує результат локально, після певної кількості біт жодним чином не буде впливати на подальші результати декодування (на відміну від кодів Хафмана або арифметичного кодування, де подібна ситуація повністю зіпсує вихідні данні)

1.3 Реалізація алгоритму кодування РМК

Задля полегшення розробки і тестування гіпотез щодо підвищення стискальної ефективності алгоритмів, було вирішено використовувати мову Python для більшої частини написаних програмних засобів. У додатку А наведено реалізацію алгоритму кодування з допомогою вищезгаданої ітеративної процедури генерації кодових слів.

1.4 Реалізація алгоритму декодування РМК

Відштовхуватимемося від наступної реалізації алгоритму декодування реверсивних мультироздільникових кодів $R_{2,3,5}$ мовою C++. Відмінність від Алгоритму 1 полягає у тому, що замість “j” TAB зберігає адресу початку потрібного «рівня» TAB. Це дає змогу зекономити операції обчислення адрес щоітерації, що у накопиченні дає дієве прискорення.

Структура елементу TAB наведена у Лістингу 1. Базова реалізація алгоритму наведена у Лістингу 2.

Наведена реалізація активно використовує глобальні змінні та адресну арифметику, тому вже є досить ефективною відносно теоретичної межі ефективності.

```
1   struct TAB_I235 {
2       int p1, p2, p3, p4;
3       struct TAB_I235 * al;
4   };
```

Лістинг 1

```
1   int decode_i235() {
2       uint p=0, OUTPUT_COUNT = 0;
3       TAB_I235 P = {0,0,0,0,(struct TAB_I235*) T235 + 256};
4       for(int i=0; i < ilen; i++) {
5           P = P.al[codes[i]];
6           p += P.p1;
7           if(P.p2 >= 0) {
8               out_i235[OUTPUT_COUNT++] = Dict_i235 + p;
9               if(P.p3 >= 0) {
10                  out_i235[OUTPUT_COUNT++] = Dict_i235 + P.p2;
11                  if(P.p4 >= 0) {
12                      out_i235[OUTPUT_COUNT++] = Dict_i235 + P.p3;
13                      p = P.p4;
14                  } else
15                      p = P.p3;
16              } else
17                  p = P.p2;
18          }
19      }
20      return OUTPUT_COUNT;
21  }
```

Лістинг 2

Дана реалізація використовувалась у роботі для валідації коректності кодування.

1.5 Алгоритм RPBC

Дане кодування було описане у статті «Enhanced byte codes with restricted prefix properties» 2005-го року. Суть кодування заключається у представленні кодових слів у вигляді послідовностей з рівно одного, двох, трьох або чотирьох байтів (схожу схему використовує система кодування UTF-8, однак вона орієнтована на кодування символів, а також була створена з урахуванням оберненої сумісності зі стандартом US-ASCII [6], що не спонукало авторів до максимального використання простору можливих кодів).

Інформація про довжину послідовності отримується зі значення першого байту, після чого зчитується відповідна кількість байт і формується кодове слово. Кодування має три параметри - (v1, v2, v3), які керують кількістю одно-, дво-, трьо- та чотирибайтових кодових слів серед усієї множини.

```
1   int rpbc_encode(int length, const uint * numbers, uchar * output) {
2       int cur, x, j = 0;
3       for(int i = 0; i < length; i++) {
4           x = numbers[i];
5           if(x < v1)
6               output[j++] = x;
7           else
8           if(x < v1 + v2*R) {
9               output[j++]= (int)((x - v1) / R) + v1;
10              output[j++]= (x - v1) % R;
11          } else
12          if(x < v1 + v2*R + v3*R2) {
13              t = x - v1 - v2*R;
14              output[j++] = (int)(t / R2) + v1 + v2;
15              output[j++] = (int)((t % R2) / R);
16              output[j++] = t % R;
17          } else {
18              t = x - v1 - v2*R - v3*R2;
19              output[j++]=(int)(t / R3) + v1 + v2 + v3;
20              output[j++]=(int)((t % R3) / R2);
21              output[j++]=(int)((t % R2) / R);
22              output[j++]= t % R;
23          }
24      }
25      return j;
26  }
```

Лістинг 3

У цій роботі у якості базових параметрів використовуються (v1, v2, v3) = (220, 35, 1) як такі, що добре зарекомендували себе з точки зору наближення до ентропії при звичайному підході до стиснення текстів. Були розглянуті і інші варіації набору параметрів.

У лістингу 3 наведено наведено функцію кодування RPBC. Тут R, R2 та R3 позначають $256, 256^2$ та 256^3 відповідно.

Алгоритм декодування RPBC оперує над байтами та невеликими таблицями, через що є надзвичайно ефективним відносно сучасних ЕОМ.

У Лістингу 4 наведено функцію декодування RPBC.

```
int rpbc_decode(const uchar * input, string ** output, int L_coded) {
    int b, k = 1, offset;
    for(int i = 0; i < L_coded;) {
        b = input[i++];
        offset = 0;
        for(int j = 1; j <= suffix[b]; j++)
            offset = offset * R + input[i++];
        output[k++] = Dict_i235 + offset + first[b];
    }
    return k;
}
```

Лістинг 4

Як можна бачити з лістингу, алгоритм надефективний з точки зору кількості допоміжних даних, що використовуються – необхідні лише кілька змінних, а також таблиці suffix та first розмірності 256 – тобто вони вміщаються у 8 кеш-ліній процесора на кожну.

РОЗДІЛ 2
ОГЛЯД МОЖЛИВИХ НАПРЯМКІВ ПОКРАЩЕННЯ АЛГОРИТМУ СТИСНЕННЯ

2.1 Особливості германської мовної групи

При розробці підходів для стиснення природномовних текстів необхідно звузити коло пошуку, тобто задати певну категорію мов, для яких ми робимо ті чи інші припущення. При цьому типово обирають категорію, до яких належать розповсюджені мови (тобто мови міжнародного спілкування – англійська, іспанська, французька, мандаринська і т.д.).

У даній роботі розглядаються підходи, що застосовні у першу чергу до англійської мови, а також до германської мовної групи. Серед цієї мовної групи характерні наступні особливості [7]:

- Загальна тенденція мов до аналітизму, тобто передачі сенсу в основному через службові слова, а не через залежні морфеми
- Наявність артиклів (зазвичай ці одиниці мови мають високу частоту зустрічання, тому графік частотного розподілу має сильно виражений «шпиль»)
- Мала кількість варіацій слів (що є наслідком високого рівня аналітизму цих мов)
- Типовий SPO-порядок слів у реченні, тобто спершу іде суб’єкт, потім предикат, потім об’єкт. Хоча інверсії і наявні, зазвичай зустрічаються у питальних і підрядних реченнях, яких значно менше, ніж звичних предикативних речень.

2.2 Частотно-позиційний аналіз

У більшості алгоритмів стиснення природномовних текстів використовується виключно частотний аналіз. Такий підхід хоча і добре

рекоменує себе у загальному випадку, все ж не завжди досягає максимальних можливих результатів.

Як уже було згадано, германська мовна група типово має SPO-структуру. Це означає, що кількість слів, що могли б стояти на першому місці у реченні набагато менша за загальну кількість слів у словнику. Те саме стосується (хоч і в меншій мірі) слів на другій позиції і далі.

Більше того, розподіл для перших позицій у реченні може суттєво відрізнятися від подальших, що дає підстави застосовувати інші параметри для РМК або RPBC для кодування конкретно цих позицій.

2.3 Сепарація речень

Окремим питанням стоїть підхід до сепарації речень. Звісно, що у письменності германських мов типовим розділовим знаком між реченнями є символ крапки («.»), знаки оклику («!»), питання («?»), а також трикрапка («…»). Однак, потрібно не тільки безпосередньо виокремити речення, слід певним чином межі речень кодувати і обробляти. У даній роботі було розглянуто три варіанти обробки.

Перший варіант передбачає кодування символу крапки як окремого кодового слова. При цьому, враховуючи особливості різних текстів, часто довелося б виділяти у окреме кодове слово перший символ після крапки. Наприклад, текст «Hello there… Hello» був би розбитий як [«Hello», «there», «.»] – перше речення, [«.»] – друге, «порожнє» речення, [«.»] – третє, також «порожне» речення, та [« », «Hello»]. Тут пробіл виділений окремим словом, оскільки наявність крапки не означає автоматичну наявність пробілу після неї. Так, текст міг би виглядати як «Hello there…\nHello» (тут \n означає перенос тексту), тоді б останнє речення виглядало б як [«\n», «Hello»]. Далі у роботі цей підхід буде згаданий як «Метод А».

Другий варіант передбачає об’єднання крапки і наступного за ним кодового слова у кортеж з подальшим кодуванням цього кортежу як одного цілого. Наприклад, при розбитті тексту «Let’s go! Come on.\nHurry!» ми б отримали наступну розбивку на речення та кодові слова: [«Let’s», «go», («!», «

»)], [«Come», «on», («.», «\n»)] та [«Hurry», («!», EOF)] (тут EOF позначає кінець файлу). Далі у тексті роботи цей варіант буде згадуватись як «Метод Б».

Третій варіант враховує ті особливості розділових знаків, що вони зазвичай не відділені пробілом від попереднього слова. Таким чином, ми можемо трактувати останнє слово речення та розділовий знак як єдине ціле, однак все ще розпізнавати кодове слово як кінець речення у випадку, коли кодове слово містить символ крапки, знак оклику чи питання. Відтак, речення «Hello… How do you do? Fine?» буде розбито на набір речень як [«Hello…»], [«How», «do», «you», «do?»] та [«Fine?»]. Окремо слід зазначити, що трикрапка була трактована як частина одного слова – ми виокремлюємо кодові слова лише на основі пробільних символів, що до того ж спрощує обробку. Побічним ефектом такого підходу є неможливість розбити текст на окремі слова за відсутності пробілів, однак такі ситуації радше рідкість. У тексті роботи цей підхід буде іменуватись як «Метод В»

2.4 Адаптивне розширення словника

Окрім загальних мовних характеристик, можна скористатися також характеристиками конкретного тексту. Гіпотеза при цьому наступна: якщо слово вже зустрілося на першому/другому/n-ому місці речення у конкретному тексті, то ймовірність його наявності на тому самому місці у речені десь далі по тексту більша, ніж у навмання вибраного слова з словника слів на цій позиції, зібраного на основі навчальної вибірки. Наприклад, якщо в англомовному тексті ми зустріли речення в тексті зі структурою «He was …» (тобто опис людини у минулому часі, некролог чи щось подібне), то з великою ймовірністю ми зустрінемо займенник «He» у якому-небудь із наступних речень тексту, у реченнях накшталт «He did …». Так само дієслово «was» також має більші шанси опинитися на другому місці одного з наступних речень.

Використовуючи це припущення, можемо використати наступний алгоритм. Нехай маємо $S_1, S_2, \dots, S_k$ – множини слів, що *вже* зустрічалися на

1-ій, 2-ій, …, k-ій позиціях відповідно. Тоді при обробці і-го слова поточного речення можливі наступні ситуації:

- Слова наявне в S_i. Тоді ми просто записуємо код цього слова з перспективи множини S_i;
- Слова немає в S_i і його код перевищує $|S_i| + 1$. Тоді можемо також безпосередньо записати код цього слова (з перспективи загального словника), а також додати його до множини S_i;
- Слова немає в S_i і його код не перевищує $|S_i| + 1$. Тоді записуємо спершу код $|S_i| + 1$ (що буде прапорцем такої ситуації), після чого записуємо код слова з перспективи глобального словника і додаємо це слово у множини S_i.

Можна також дещо змінити підхід до кодування у разі відсутності слова у словнику S_i, не змінюючи перший пунк алгоритму, і застосувавши універсальний підхід до другого та третього випадків. Матимемо наступні ситуації:

- Слова наявне в S_i. Тоді ми просто записуємо код цього слова з перспективи множини S_i;
- Слова немає в S_i. Тоді можемо також безпосередньо записати код цього слова (з перспективи загального словника), зсунувши цей код на $|S_i| + 1$ (просто додавши до самого коду значення $|S_i| + 1$), а крім того, включити це слово до множини S_i.

Існують також два варіанти упорядкування кодових слів всередині множин $S_1, S_2, \dots, S_k$. Простіший шлях полягає в сортуванні за статистичними даними, отриманими з навчальної вибірки (тобто заздалегідь). Це загалом спрощує реалізацію, але не враховує особливості конкретного тексту. З іншого боку, можна постійно оновлювати частоти слів по мірі їх знаходження у тексті, підтримуючи поточний розподіл частот близьким до справжнього у конкретному тексті. Нарешті, можна застосувати гібридний підхід, задаючи

початкову вагу слова на основі навчальної вибірки і змінюючи її по мірі надходження нових слів.

РОЗДІЛ 3
СТАТИСТИЧНИЙ АНАЛІЗ ТЕСТОВОЇ ВИБІРКИ

3.1 Методика створення тестової вибірки

У якості матеріалів для тестової вибірки було вирішено обрати проєкт «Гутенберг». З понад 60 тисяч текстів [8], що присутні у каталозі проєкту, випадковим чином було обрано ~100 мегабайт англомовних матеріалів. Для зручності користування усі матеріали було сконкатеновано у один суцільний файл.

Тексти у проєкті Гутенберг форматовані для зручності виведення у термінал, через що довжина рядків файлів не перевищує 120 символів. Це дещо спотворює тестові дані, тож була проведена передобробка. Якщо перенос тексту був диктований саме обмеженнями довжини рядка, а не структурними особливостями окремого тексту, такий перенос видалявся, а послідовність дубльованих пробілів, якщо така є на місці видалення, замінялась на єдиний пробіл.

Окрім того, у деяких текстах зустрічались «некласичні» розділові знаки та символи пунктуації з розширеної таблиці Unicode (навіть якщо існував прямий аналог в таблиці ASCII). У таких випадках, задля збереження консистетності, розділові з таблиці Unicode замінювались на прямі аналоги з таблиці ASCII. Наприклад, символ «”» (має код U+201D) був замінений символом «”» (має код U+0022).

Нарешті, окремі тексти були вилучені через їх специфічний вміст. До прикладу, одна з книжок містила лінгвістичні викладки щодо грецької мови, у яких активно використовувався не тільки грецький алфавіт, а і допоміжні символи Unicode, а також фонетичні позначення та лігатури.

Окремо було також виділено вибірку меншого розміру з шевдськомовних текстів задля порівняння результатів при застосуванні описаних підходів щодо більш складних мов германської мовної групи.

3.2 Контекстно-незалежний частотний аналіз тестової вибірки

За контекстно-незалежного частотного аналізу текст розбивався на окремі слова і для кожного слова підраховувалась кількість входжень у текст. Роділові знаки при цьому або «склеювались» з сусіднім словом, або, у випадку ізоляції пробілами, вважалося окремим кодовим словом.

Графік розподілу частот зображено на рисунку 1. Слід зазначити, що шкала обох вісей на графіку логарифмічна.

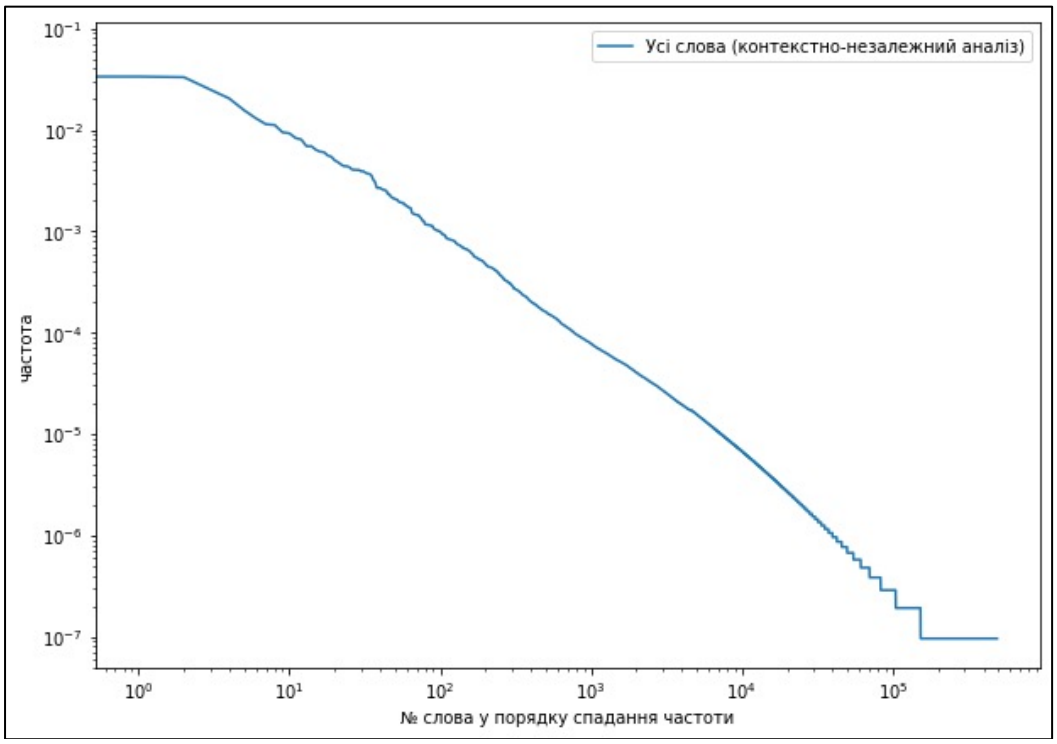

Рис. 1

Всього у вибірці виявилось 10’398’410 слів, унікальних – 491’952. Десятьма найчастішими словами у англомовній вибірці текстів є наступні слова: «the» (5,81%), «and» (3,35%), «of» (3,31%), «to» (2,48%) , «a» (2,03%), «in» (1,56%), «was» (1,29%), «I» (1,13%), «that» (1,11%), «his» (0,94%). Загалом частка цих 10 найчастотніших слів складає понад 23% серед усіх входжень слів.

6.59% слів містять символи кінця речення – крапку, знак оклику, знак питання чи знак перенесення картеки «\n». Таким чином, середня довжина речення – 15 кодових слів.

Бінарна ентропійна межа стиснення для цього словника виявилась на рівні 11,2717 біт на слово.

3.3 Частотно-позиційний аналіз тестової вибірки

3.3.1 Метод сепарації речень A

Як уже було згадано у розділі 2.3, метод сепарації речень A полягає в кодуванні символів кінця речення окремим кодовим словом. Для різних символів кінця речення при цьому виділялися різні кодові слова. Жодних додаткових перетворень при цьому не застосовувалось.

Порівняльний графік розподілу частот слів відносно їх позиції у реченнях наведено на рисунках 2 та 3.

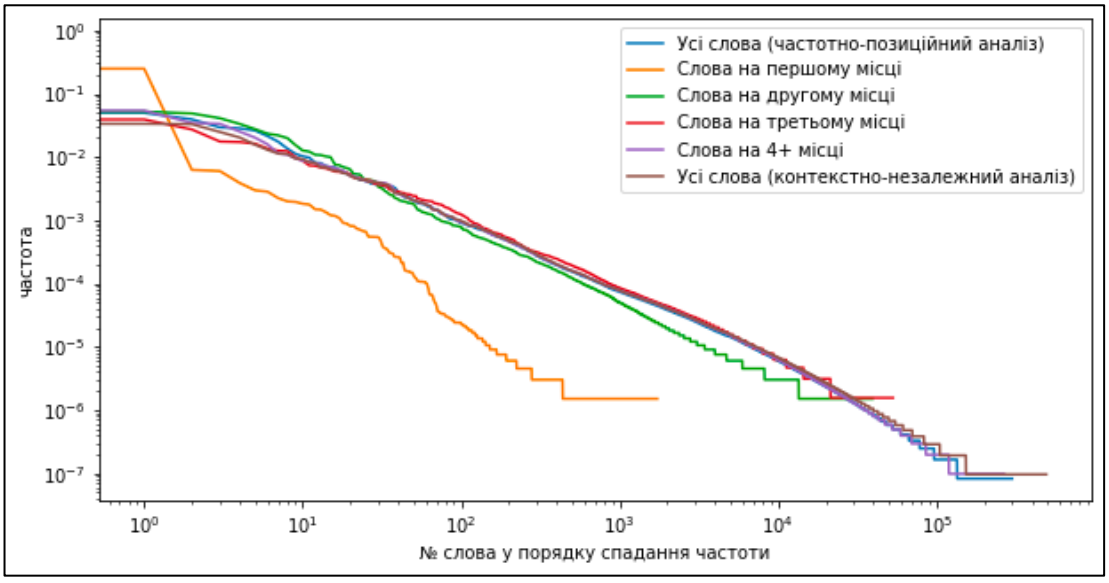

Рис. 2

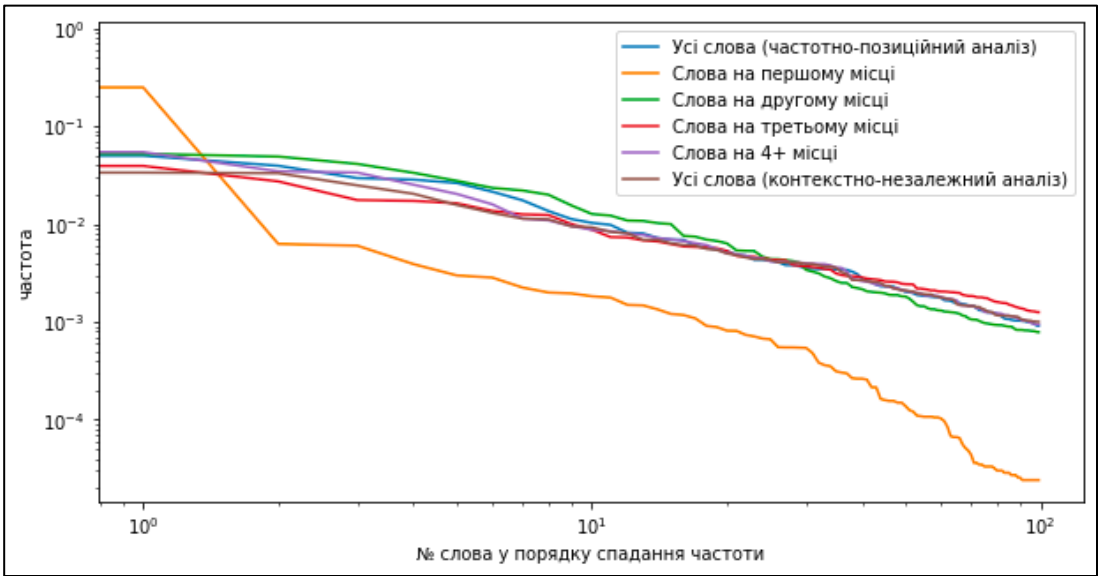

Рис. 3

При сепарації речень за рахунок трактування розділових знаків як окремих кодових слів загальний словник дещо відрізняється за частотами. Так, список найчастіших слів виглядає наступним чином: «.» (5,1%), «the» (4,99%), порожній рядок «» (3,94%), «and» (2,94%), «of» (2,84%), «\n» (2,63%), «to» (2,14%), «a» (1,74%), «in» (1,35%), «was» (1,12%). Ці 10 слів складають 28,8% від загальної кількості слів.

Бінарна ентропійна межа для цього словника складає 10,4191 (що на 8% менше за ентропійну межу словника при контекстно-незалежному аналізі), однак загальна кількість слів при цьому на 16% більша, що нівелює ефект зменшення ентропії. Утім, метою застосування частотно-позиційного аналізу було не розширення словника за рахунок розділових знаків, а навпаки, його зменшення в залежності від контексту. Тож суттєвими є характеристика словників слів, що стоять на першому, другому та третьому місцях у реченнях.

Так, словник слів, що зустрічаються на першому місці, суттєво менший за загальний – усього 1730 слів. Найчастішими при цьому є порожній рядок «» (69%) та символ перенесу рядка «\n» (24%). Таке зміщення частот відповідним чином відобразилось на теоретичній межі стиснення – ентропійна межа сягає 1.46325 бітів на слово.

Унікальних слів, що зустрічалися на другому місці у реченні у тестовій вибірці було 39380. Найчастішими при цьому є наступні: «\n» (18,8%), «The» (5,22%), «I» (4,88%), «He» (4,11%), «.» (3,33%), «It» (2,76%), «She» (2,34%), «2,20» (%), «But» (1,99%), «And» (1,54%). Бінарна ентропія приблизно дорівнює 7,9131 бітів на слово.

Словник слів з третіх позицій за результатами аналізу тестової вибірки складається з 53035 унікальнх слів. Найчастішими словами є наступні: «.» (6,96%), «was» (3,92%), «the» (2,72%), «I» (1,75%), «had» (1,72%), «is» (1,62%). Ентропійна межа складає 10,522.

Для слів, що знаходяться після третьої позиції у реченнях, ентропійна межа склала 10,5158. Найастішими при цьому є «the» (5,78%), «.» (5,42%), «and» (3,45%), «of» (3,35%), «to» (2,52%), «a» (2,02%), «in» (1,59%), «that» (1,13%), «was» (1,09%), «his» (0,95%).

3.3.2 Метод сепарації речень Б

У розділі 2.3 вже було описано відмінності методу Б від методу А. Передбачувано, що позиції слів у термінах методу Б буде зміщено на одиницю відносно методу А: те, що раніше було «словом на другій позиції», тепер стане «словом на першій позицій», адже тепер «перше слово» є складовою останнього слова (символу кінця речення – крапки, знаку оклику чи питання). Рисунок 4 зображує графік розподілу частот в контекстних словниках при застосуванні методу сепарації Б.

Так, словник слів, що зустрічаються на першому місці у речені має 39803 унікальних слова, найчастішими при цьому є: «\n» (18,83%), «The» (5,22%), «I» (4,89%), «He» (4,11%), «It» (2,76%), «She» (2,34%), «» (2,20%),

«But» (1,99%), «('.', ")» - суміщена крапка та порожній рядок (1,88%), «And» (1,55%). Бінарна ентропія складає 8,015 бітів на слово.

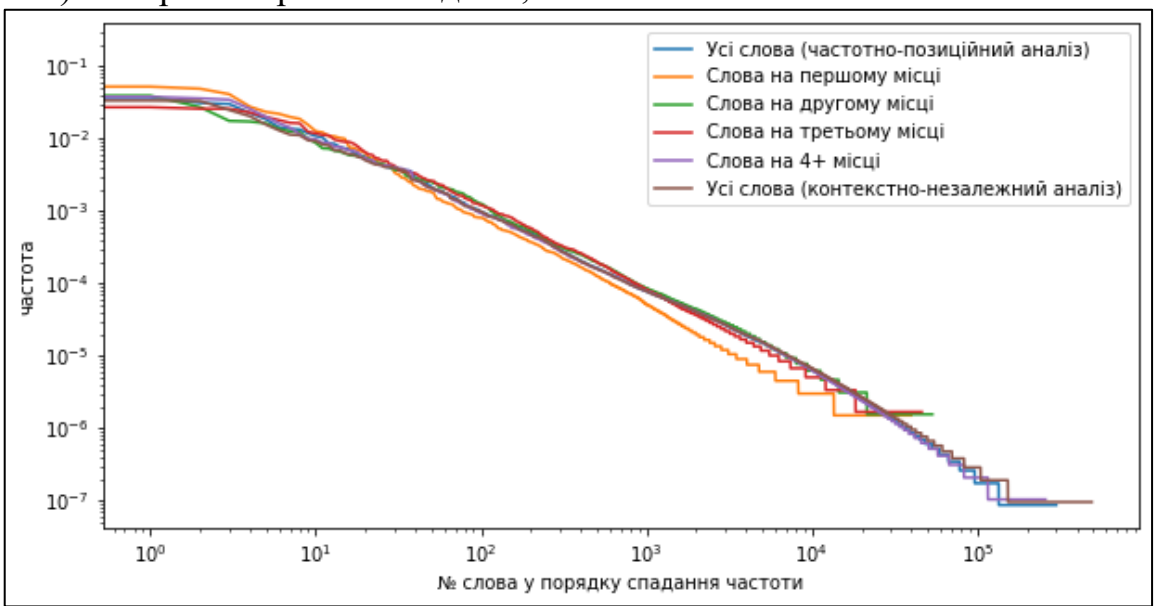

Рис. 4

Найчастішими словами на другій позиції в реченні були наступні: «('.', ")» - суміщена крапка та порожній рядок (5,74%), «was» (3,92%), «the» (2,73%), «I» (1,76%), «had» (1,72%), «is» (1,63%), «The» (1,36%), «he» (1,26%), «\n» (1,24%), «you» (1,00%). Ентропійна межа складає 10,612.

Слова на третій позиії мають наступні найчастотніші слова: «the» (3,96%), «('.', '\n')» (2,72%), «('.', ")» (2,59%), «was» (2,59%), «a» (2,30%), «and» (1,99%), «of» (1,80%), «to» (1,65%), «not» (1,63%), «I» (1,24%). Бінарна ентропія для цього словника складає 10,201 бітів на слово.

Словник слів на позиціях після третьої має ентропійну межу близько 10,575. Найчастішими при цьому є наступні слова та їх комбінації: «the» (5,89%), «('.', ")» (3,78%), «and» (3,60%), «of» (3,45%), «to» (2,58%), «a» (2,01%), «in» (1,63%), «('.', '\n')» (1,42%), «that» (1,15%), «was» (1,00%).

3.3.3 Метод сепарації речень В

Третій метод сепарації речень працює за достатньо відмінним від методів А і Б принципом, не виділяючи на символи сепарації речень окремі кодові слова без потреби. Таким чином, словники частот суттєво відрізняються за структурою і дещо відрізняються за частотою конкретних слів. На рисунку 5 зображено графік розподілу частот для даного методу загалом, а на рисунку 6 – лише для перших 100 слів з кожного словника.

Для словника слів, що зустрічаються на першому місці в реченні, маємо наступні найчастотніші слова: «The» (5,52%), «I» (5,25%), «He» (4,28%), «And» (3,46%), «It» (2,83%), «She» (2,37%), «But» (2,23%), «They» (1,32%), «You» (1,28%), «If» (1,11%). Таким чином, ці 10 займенників, сполучників та артиклів покривають 30% від усіх входжень слів на перших місцях. Бінарна ентропія при цьому складає 9,885 – більше, ніж у відповідних словників при

застосуванні методів А та Б. Однак це зовсім не означає гірший стискальний потенціал, що і буде продемонстровано у розділі 4.

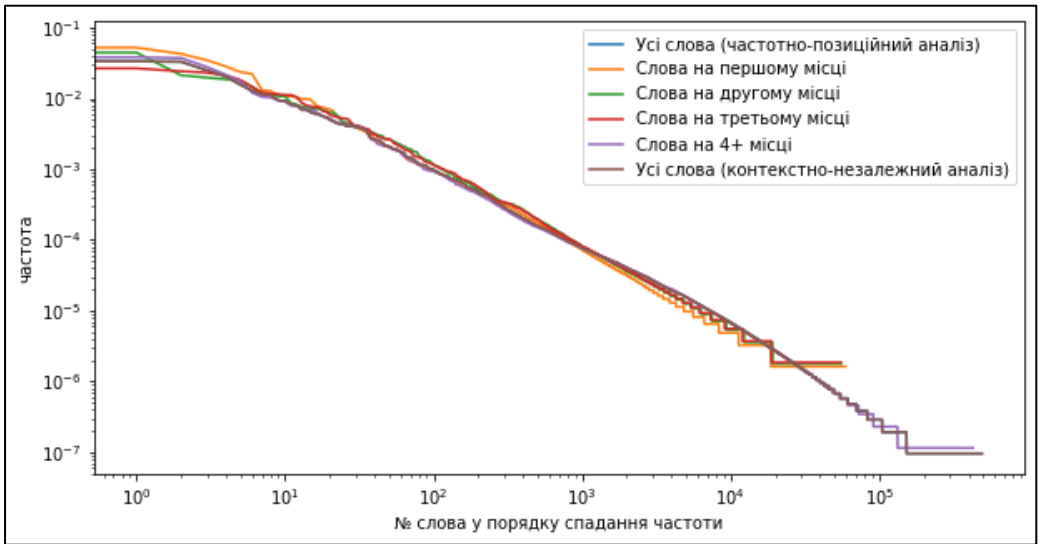

Рис. 5

Словник слів, що зустрічаються на другому місці, має ентропію на рівні 10,503. Список найчастіших слів виглядає наступним чином: «was» (4,81%), «the» (4,47%), «had» (2,12%), «is» (1,97%), «I» (1,88%), «he» (1,79%), «a» (1,47%), «it» (1,19%), «you» (1,15%), «and» (1,08%). При цьому варіативність словника дещо менша, ніж словника слів з перших позицій – усього 55443 унікальних слова у порівнянні з 59380. Розподіл при цьому хоч і має досить крутий «стрибок» частоти від другого до третього слова, все ж виглядає більш «пологим».

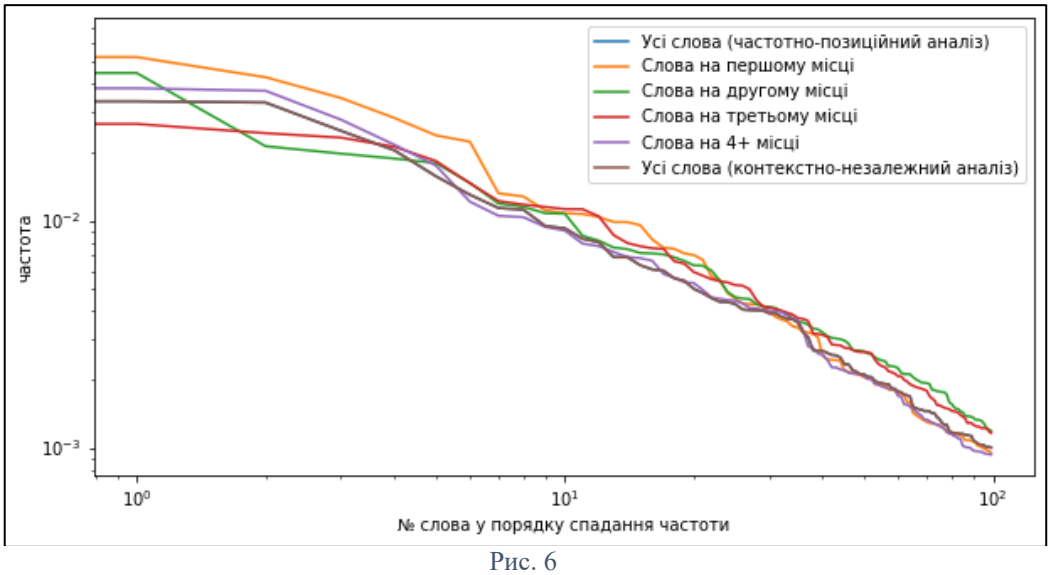

Рис. 6

Словник слів з третіх позицій має наступний «топ» найчастотніших слів: «the» (4,69%), «a» (2,66%), «of» (2,43%), «was» (2,32%), «to» (2,11%),

«not» (1,83%), «and» (1,46%), «I» (1,22%), «in» (1,18%), «had» (1,15%). Бінарна ентропія становить близько 10,571.

Серед слів на 4 та наступних місцях найчастотнішими є наступні: «the» (6,31%), «and» (3,82%), «of» (3,73%), «to» (2,78%), «a» (2,16%), «in» (1,75%), «that» (1,21%), «his» (1,05%), «was» (1,03%), «with» (0,94%). Ентропійна межа – 11,127.

Як можна побачити з наведеного вище суто теоретичного аналізу, контекст дійсно сильно впливає на частоту входжень тих чи інших слів, що дає підстави для застосування описаного методу при стисканні природномовних текстів. У наступному розділі буде наведено результати безпосереднього тестування описаних методик.

РОЗДІЛ 4
ОПИС ТА АНАЛІЗ ОТРИМАНИХ РЕЗУЛЬТАТІВ

4.1 Стиснення за допомогою реверсивних мультироздільникових кодів

4.1.1 Стиснення на основі методу сепарації A

У таблиці 2 наведено результати емпіричного підбору параметрів для найбільшої стискальної ефективності за допомогою реверсивних мультироздільникових кодів. Для кожного словника підбір проводився окремо. Як видно з таблиці, реверсивні мультироздільникові коди мають досить гарне наближення до ентропії в усіх випадках, окрім №2 – оскільки за своєю структурою мультироздільникові коди можуть містити лише кодові слова довжини починаючи від 2, наблизитись до ентропійної межі (яка складає 1,4632 біти на слово) неможливо з використанням цих кодів. Можливим виходом могло б бути використання арифметичних кодів або кодів Хафмана, однак у такому випадку усі переваги використання реверсивних мультироздільникових кодів нівелюються.

№	Коментар	Ентропія	Найкращий набір роздільників	Бітів на слово, у середньому	Приріст відносно ентропії	Бітів усього
1	Загальний словник, без розбиття на речення	11,2717	[2, 4, 5, 6]	11,5381	+2,36%	119977575
2	Слова на першому місці у реченні	1,4632	[1, 3, 4, 5]	2,5295	+72,87%	1675044
3	Слова на другому місці у реченні	7,9131	[2, 3, 4, 5]	8,0539	+1,78%	5333246
4	Слова на третьому місці в реченні	10,5222	[2, 4, 5, 6]	10,8640	+3,25%	6952813
5	Слова на 2+ місці в реченні	10,6538	[2, 4, 5, 6]	10,9017	+2,33%	124982511
6	Слова на 3+ місці в реченні	10,5895	[2, 4, 5, 6]	10,8343	+2,31%	117035154
7	Слова на 4+ місці в реченні	10,5158	[2, 4, 5, 6]	10,7582	+2,30%	109327748
8	Загалом слова у реченні	10,4191	[2, 3, 5, 6]	10,6464	+2,18%	129104681

Таб. 2

Параметри підбирались серед усіх можливих наборів роздільників з кількістю не більше за 4 та максимальним роздільником 11.

У таблиці 3 наведено результати при різних можливих варіантах комбінацій кодувань. Базова варіація – контекстно-незалежне кодування.

Комбінація «2+5» означає, що була застосовано контекстно-залежний аналіз з використанням словників «Слова на першому місці у реченні» та «Слова на 2+ місці в реченні» - відповідно другий і п’ятий рядок таблиці 2. За таким самим прицнипом комбінація «2 + 3 + 4 + 7» означає, що були застосовані словники слів, що розташовувались на першому, другому та третьому місцях, а також словник слів, що зустрічалися на позиції 4 або більше.

Комбінація	Бітів усього (менше – краще)	Відносно (1)	Відносно (8)
1	119846113	+0,00%	-7,17%
2 + 5	126657555	+5,68%	-1,90%
2 + 3 + 6	124043444	+3,50%	-3,92%
2 + 3 + 4 + 7	123288851	+2,87%	-4,50%
8	129104681	+7,73%	0,00%

Таб. 3

Як видно з результатів, наведених у таблиці 3, метод сепарації не тільки не покращує результати стиснення відносно базового варіанту (1) безконтекстного стиснення, а і погіршує їх, програючи на 2,87% у стискальній ефективності.

4.1.2 Стиснення на основі методу сепарації Б

У таблиці 4 наведено аналогічні результати для словників, отриманих за допомогою методу сепарації Б. Набори роздільників підбирались так само, як і у розділі 4.1.1.

№	Коментар	Ентропія	Найкращий набір роздільників	Бітів на слово, у середньому	Приріст відносно ентропії	Бітів усього
1	Загальний словник, без розбиття на речення	11,2717	[2, 4, 5, 6]	11,5381	+2,36%	119977575
2	Слова на першому місці у реченні	8,0151	[2, 3, 4, 5]	8,0151	+0,00%	5402096
3	Слова на другому місці у реченні	10,6122	[2, 4, 5, 6]	10,9726	+3,40%	7022312
4	Слова на третьому місці в реченні	10,2013	[2, 3, 5, 6]	10,5443	+3,36%	6249270
5	Слова на 2+ місці в реченні	10,6683	[2, 4, 5, 6]	10,9216	+2,37%	117978310
6	Слова на 3+ місці в реченні	10,5922	[2, 3, 5, 6]	10,8432	+2,37%	110191662
7	Слова на 4+ місці в реченні	10,5751	[2, 3, 5, 6]	10,8245	+2,36%	103586828
8	Загалом слова у реченні	10,7371	[2, 4, 5, 6]	10,9975	+2,43%	126080655

Таб. 4

У таблиці 5 так само наведено комбіновані результати при застосуванні різних наборів словників. Результати таблиці хоч і демонструють покращення відносно методу сепарації A, все ж програють безконтекстному стисненню на 1,9%.

Комбінація	Бітів усього (менше – краще)	Відносно (1)	Відносно (8)
1	119977575	+0,00%	-4,84%
2 + 5	123380406	+2,84%	-2,14%
2 + 3 + 6	122616070	+2,20%	-2,75%
2 + 3 + 4 + 7	122260506	+1,90%	-3,03%
8	126080655	+5,09%	0,00%

Таб. 5

4.1.3 Стиснення на основі методу сепарації B

У таблиці 6 наведено результати емпіричного підбору параметрів реверсивних мультироздільникових кодів для словників, отриманих методом сепарації B. Як і у попередніх двох випадках, роздільники вибирались з усіх можливих наборів з щонайбільше чотирьох роздільників, кожен з яких не перевищує 11.

№	Коментар	Ентропія	Найкращий набір роздільників	Бітів на слово, у середньому	Приріст відносно ентропії	Бітів усього
1	Загальний словник, без розбиття на речення	11,2717	[2, 4, 5, 6]	11,5381	+2,36%	119977575
2	Слова на першому місці у реченні	9,8855	[2, 3, 5, 6]	10,1434	+2,61%	6232602
3	Слова на другому місці у реченні	10,5032	[2, 4, 5, 6]	10,8588	+3,39%	6078537
4	Слова на третьому місці в реченні	10,5706	[2, 4, 5, 6]	10,9418	+3,51%	5887579
5	Слова на 2+ місці в реченні	11,1465	[2, 4, 5, 6]	11,4024	+2,30%	111560226
6	Слова на 3+ місці в реченні	11,1314	[2, 4, 5, 6]	11,3871	+2,30%	105037007
7	Слова на 4+ місці в реченні	11,1279	[2, 4, 5, 6]	11,3841	+2,30%	98883334
8	Загалом слова у реченні	11,2717	[2, 4, 5, 6]	11,5381	+2,36%	119977575

Таб. 6

У таблиці 7 наведено результати комбіновані результати при застосуванні різних наборів словників. Варто зазначити декілька моментів. По-перше, оскільки загальний підхід до обробки не змінився (тобто сепарація самих слів відбувається «по-старому»), то і варіанти 1 та 8 ідентичні з точки зору усіх

показників – адже по суті є ідентичними словниками. У той же час варіанти «2 + 5», «2 + 3 + 6» та «2 + 3 + 4 + 7» показують покращення відносно базового варіанту, досягаючи 2,41-відсоткового зменшення розміру результуючого файлу відносно базової реалізації.

Комбінація	Бітів усього (менше – краще)	Відносно (1)	Відносно (8)
1	119977575	0,00%	0,00%
2 + 5	117792828	-1,82%	-1,82%
2 + 3 + 6	117348146	-2,19%	-2,19%
2 + 3 + 4 + 7	117082052	-2,41%	-2,41%
8	119977575	0,00%	0,00%

Таб. 7

4.2 Стиснення за допомогою RPBC

Як було показано у розділі 4.1, методи сепарації A та Б працюють значно гірше за метод сепарації B, тому тестування частотно-позиційної оптимізації у комбінації з RPBC будуть проводитись лише з методом сепарації B.

№	Коментар	Ентропія	Найкращий набір роздільників	Бітів на слово, у середньому	Приріст відносно ентропії	Бітів усього
1	Загальний словник, без розбиття на речення	11,2717	[162, 85, 8]	12,6506	+12,23%	131545680
2	Слова на першому місці у реченні	9,8855	[182, 72, 1]	11,5537	+16,88%	7099144
3	Слова на другому місці у реченні	10,5032	[180, 74, 1]	11,9570	+13,84%	6693240
4	Слова на третьому місці в реченні	10,5706	[181, 73, 1]	12,0061	+13,58%	6460232
5	Слова на 2+ місці в реченні	11,1465	[162, 86, 7]	12,5349	+12,46%	122640928
6	Слова на 3+ місці в реченні	11,1314	[159, 89, 7]	12,5364	+12,62%	115638016
7	Слова на 4+ місці в реченні	11,1279	[157, 91, 7]	12,5433	+12,72%	108952352
8	Загалом слова у реченні	11,2717	[162, 85, 8]	12,6506	+12,23%	131545680

Таб. 8

Таблиці 8 та 9 ідентичні за структурою змісту до відповідних таблиць з розділу 4.1, однак алгоритм підбору параметрів відрізняється (позаяк простір параметрів для перебору ширший, а массовий підрахунок довжини кодового слова легший). Як видно з таблиці 8, RPBC коди забезпечують значно гіршу стискальну ефективність. (+12% відносно ентропії, у той час як у реверсивних

мультироздільникових кодів було +2-3%). Однак RPBC коди все одно мають свої переваги, які уже розглядались в розділі 2.

Результати, наведені у таблиці 9, свідчать про те, що частотно-позиційний аналіз з методом сепарації B гарно працює з RPBC.

Комбінація	Бітів усього (менше – краще)	Відносно (1)	Відносно (8)
1	131545680	0,00%	0,00%
2 + 5	129740072	-1,37%	-1,37%
2 + 3 + 6	129430400	-1,61%	-1,61%
2 + 3 + 4 + 7	129204968	-1,78%	-1,78%
8	131545680	0,00%	0,00%

Таб. 9

ВИСНОВКИ

Розроблено ряд гіпотез щодо можливих напрямків покращення алгоритмів стиснення природномовних текстів на основі RPBC та реверсивних мультироздільникових кодів. Наведено та протестовано реалізації алгоритмів з залученням зазначених гіпотез. Для розглянутих реалізацій проведено тестування на вхідних даних різного характеру. Отримані при тестуванні дані подано у вигляді таблиць та графіків для наочного сприйняття. Гіпотези, висунуті на етапі проектування алгоритмів частково підкріплюються цими даними, частково спростовуються.

Отримані результати можна використати не тільки для подальшого удосконалення згаданих алгоритмів, а і для розробки узагальнених підходів до стиснення природномовних текстів.

ДОДАТОК А

Лістинг коду енкодерів для реверсивних мультироздільникових кодів та RPBC

```
class ReversesMDCodes:
    def __init__(self, delim: list):
        delim = sorted(delim)
        self.delim = delim
        self._M = delim
        self._t = len(delim)
        self._K = [
            i for i in range(max(self._M) + 2)
            if i not in self.delim
        ]
        self._q = len(self._K)
        self._M_t = delim[-1]
        self._M_t_suffix = '1' * (self._M_t + 1)

        self.codes: t.List[str] = []
        self.code_groups: t.List[t.List[str]] = [[]]

    def generate_codes(self, limit=100):
        while len(self.codes) < limit:
            L = len(self.code_groups)
            self.code_groups.append(None)
            current_code_group = []
            for i in self._K:
                if L - i - 1 < 0:
                    continue
                current_code_group += [
                    code + '0' + '1' * i
                    for code in self.code_groups[L - i - 1]
                ]

            current_code_group += [
                code + '1'
                for code in self.code_groups[L - 1]
                if code.endswith(self._M_t_suffix)
            ]
            if L - 1 in self._M:
                current_code_group.append('0' + '1' * (L - 1))

            self.code_groups[-1] = current_code_group
            self.codes += current_code_group

    def __getitem__(self, item: int):
        if item >= len(self.codes):
            self.generate_codes(limit=item + 1)
```

```
        return self.codes[item]

class RPBC:
    _R = 256
    _R2 = 256 * 256
    _R3 = 256 * 256 * 256
    num_to_bits: t.List[str]

    def __init__(self, v1: int, v2: int, v3: int):
        self._generate_num_to_bits()
        self.v1 = v1
        self.v2 = v2
        self.v3 = v3

    def _generate_num_to_bits(self):
        self.num_to_bits = ['']
        for i in range(8):
            self.num_to_bits = (
                    ['0' + code for code in self.num_to_bits] +
                    ['1' + code for code in self.num_to_bits]
            )

    def __getitem__(self, x: int):
        result = []
        if x < self.v1:
            result.append(x)
        elif x < self.v1 + self.v2 * self._R:
            result.append((x - self.v1) // self._R + self.v1)
            result.append((x - self.v1) % self._R)
        elif x < self.v1 + self.v2 * self._R + self.v3 * self._R2:
            _t = x - self.v1 - self.v2 * self._R
            result.append((_t // self._R2) + self.v1 + self.v2)
            result.append((_t % self._R2) // self._R)
            result.append(_t % self._R)
        else:
            _t = x - self.v1 - self.v2 * self._R - self.v3 * self._R2
            result.append((_t // self._R3) + self.v1 + self.v2 + self.v3)
            result.append((_t % self._R3) // self._R2)
            result.append((_t % self._R2) // self._R)
            result.append(_t % self._R)
        print(result)
        return ''.join(self.num_to_bits[byte] for byte in result)
```

ДОДАТОК Б

Лістинг коду текстових аналізаторів

```
import typing as t

class BaseTextAnalyzer:
    _available_count_dicts: t.List[str] = []
    def __init__(self):
        self.reset()

    def reset(self):
        for key in self._available_count_dicts:
            setattr(self, key, defaultdict(lambda: 0))

    def analyze(self, text: str):
        raise NotImplementedError()

    @property
    def stats(self):
        return {
            key: Stats(getattr(self, key))
            for key in self._available_count_dicts
        }

class ContextlessTextAnalyzer(BaseTextAnalyzer):
    _available_count_dicts = ['all_words_contextless']
    all_words_contextless: t.Dict[str, int]

    def analyze(self, text: str):
        text = text.split(' ')
        for word in text:
            self.all_words_contextless[word] += 1
        self.all_words_contextless[__TEXT_END__] += 2

class BaseContextfullTextAnalyzer(BaseTextAnalyzer):
    _available_count_dicts = [
        'all_words',
        'words_1st_place',
        'words_2nd_place',
        'words_3rd_place',
        'words_2_plus_place',
        'words_3_plus_place',
        'words_4_plus_place',
    ]
    all_words: t.Dict[str, int]
    words_1st_place: t.Dict[str, int]
    words_2nd_place: t.Dict[str, int]
    words_3rd_place: t.Dict[str, int]
```

```
    words_2_plus_place: t.Dict[str, int]
    words_3_plus_place: t.Dict[str, int]
    words_4_plus_place: t.Dict[str, int]


    def parse_sentence(self, sentence: str) -> list:
        end = '.'
        if sentence.endswith('?') or sentence.endswith('!'):
            end = sentence[-1]
            sentence = sentence[:-1]
        sentence = sentence.replace('\n', '\n ')
        sentence = sentence.split(' ')
        sentence.append(end)
        return sentence

    def calculate_sentence_stats(self, sentence: list):
        for word in sentence:
            self.all_words[word] += 1
        for word in sentence[1:]:
            self.words_2_plus_place[word] += 1
        for word in sentence[2:]:
            self.words_3_plus_place[word] += 1
        for word in sentence[3:]:
            self.words_4_plus_place[word] += 1
        try:
            self.words_1st_place[sentence[0]] += 1
            self.words_2nd_place[sentence[1]] += 1
            self.words_3rd_place[sentence[2]] += 1
        except IndexError:
            pass

class ContextfullTextAnalyzerTypeA(BaseContextfullTextAnalyzer):
    def analyze(self, text: str):
        text = text.replace('?', '?.').replace('!', '!.')
        text = text.split('.')

        for sentence in text:
            sentence = self.parse_sentence(sentence)
            self.calculate_sentence_stats(sentence)

        self.words_1st_place[__TEXT_END__] += 2
        self.words_2nd_place[__TEXT_END__] += 2
        self.words_3rd_place[__TEXT_END__] += 2

class ContextfullTextAnalyzerTypeB(BaseContextfullTextAnalyzer):
    def analyze(self, text: str):
        text = text.replace('?', '?.').replace('!', '!.')
        text = text.split('.')
        not_first_sentence = False

        for sentence, sentence_next in zip(text, text[1:] + [__TEXT_END__]):
```

```
            sentence = self.parse_sentence(sentence)
            sentence_next = self.parse_sentence(sentence_next)
            sentence[-1] = (sentence[-1], sentence_next[0])

            self.calculate_sentence_stats(sentence[not_first_sentence:])
            not_first_sentence = True

        self.words_1st_place[__TEXT_END__] += 1
        self.words_2nd_place[__TEXT_END__] += 1
        self.words_3rd_place[__TEXT_END__] += 1

class ContextfullTextAnalyzerTypeC(BaseContextfullTextAnalyzer):
    def split_sentences(self, text: str) -> t.Iterable[t.List[str]]:
        text = text.split(' ')
        current_sentence = []
        for word in text:
            current_sentence.append(word)
            if '.' in word or '?' in word or '!' in word:
                yield current_sentence
                current_sentence = []
        if current_sentence:
            yield current_sentence

    def analyze(self, text: str):
        for sentence in self.split_sentences(text):
            self.calculate_sentence_stats(sentence)

        self.words_1st_place[__TEXT_END__] += 2
        self.words_2nd_place[__TEXT_END__] += 2
        self.words_3rd_place[__TEXT_END__] += 2
```

ДОДАТОК В

Лістинг коду шукача оптимального набору параметрів для енкодерів реверсивних мультироздільникових кодів та RPBC

```
def find_best_params_rmdc(stats: Stats, max_delim_count: int, max_delim: int):
    def find_best_params(delims: list = []):
        best_params = None
        best_score = None

        if delims:
            rmdc = ReversesMDCodes(delims)
            score = sum(
                len(rmdc[i]) * cnt
                for i, (_, cnt) in enumerate(stats.stats_sorted)
            )
            best_params = delims.copy()
            best_score = score
        if len(delims) < max_delim_count:
            for i in range((delims or [0])[-1] + 1, max_delim + 1):
                delims.append(i)
                score, params = find_best_params(delims)
                if best_score is None or (score is not None and score < best_score):
                    best_score = score
                    best_params = params.copy()
                delims.pop()
        return best_score, best_params

    best_score, best_params = find_best_params()
    return best_score, best_params


def find_best_params_rpbc(stats: Stats):
    def make_prefix_sums(stats: Stats):
        res = [0]
        for _, cnt in stats.stats_sorted:
            res.append(res[-1] + cnt)
        return res

    def get_score_rpbc(prefix_sums, v1, v2, v3):
        bounds = [
            0,
            v1,
            v1 + v2 * 256,
            v1 + v2 * 256 + v3 * 256 * 256,
            (256 - v1 - v2 - v3) * 256 * 256 * 256 + (v1 + v2 * 256 + v3 * 256 * 256),
        ]
        if len(prefix_sums) >= bounds[-1]:
            return 2**63
```

```
    bounds = [min(bound, len(prefix_sums) - 1) for bound in bounds]
    return (
            (prefix_sums[bounds[1]] - prefix_sums[bounds[0]]) * 8 +
            (prefix_sums[bounds[2]] - prefix_sums[bounds[1]]) * 16 +
            (prefix_sums[bounds[3]] - prefix_sums[bounds[2]]) * 24 +
            (prefix_sums[bounds[4]] - prefix_sums[bounds[3]]) * 32
    )
prefix_sums = make_prefix_sums(stats)
best_params = None
best_score = None
for v1 in range(1, 256):
    for v2 in range(1, 256 - v1):
        for v3 in range(1, 256 - v1 - v2):
            score = get_score_rpbc(prefix_sums, v1, v2, v3)
            if not best_score or best_score > score:
                best_score = score
                best_params = [v1, v2, v3]
return best_score, best_params
```

ДОДАТОК Г

Лістинг коду повноцінного енкодера-декодера.

```
class ContextfullCompressorRMDC:
    str_to_byte: t.Dict[str, int]
    def __init__(
            self,
            stats_and_params: t.List[t.Tuple[Stats, t.List[int]]]
    ):
        self.encoders = [
            ReversesMDCodes(params)
            for _, params in stats_and_params
        ]
        self.stats = [
            deepcopy(stats)
            for stats, _ in stats_and_params
        ]
        self.make_byte_mapping()

    def make_byte_mapping(self):
        self.byte_to_str = ['']
        for i in range(8):
            self.byte_to_str = [
                '0' + s for s in self.byte_to_str
            ] + [
                '1' + s for s in self.byte_to_str
            ]
        self.str_to_byte = {
            key: value
            for value, key in enumerate(self.byte_to_str)
        }

    def compress(self, text: str) -> bytearray:
        result = bytearray()
        result_str = ''
        for sentence in self.split_sentences(text):
            for i, word in enumerate(sentence):
                state = min(i, len(self.encoders) - 1)
                try:
                    word_code = self.stats[state].word_to_code[word]
                except KeyError:
                    result_str += self.handle_exceptional_str(word, state)
                else:
```

```
                result_str += self.encoders[state][word_code]
            appendix, result_str = self.str_to_bytes(result_str)
            result += appendix
        appendix, _ = self.str_to_bytes(result_str, fill=True)
        result += appendix
        return result

    def decompress(self, encoded_text: bytearray) -> str:
        state = 0
        current_str = ''
        result = []
        for byte in encoded_text:
            current_str += self.byte_to_str[byte]
            while True:
                try:
                    code, offset = self.encoders[state].partial_decode(
                        current_str,
                        next_delim_b=self.encoders[min(state + 1,
len(self.encoders) - 1)].delim,
                        next_delim_a=self.encoders[0].delim,
                    )
                except IndexError:
                    break
                current_str = current_str[offset:]
                if code >= len(self.stats[state].stats_sorted):
                    raise NotImplementedError()
                word = self.stats[state].stats_sorted[code][0]
                if '.' in word or '?' in word or '!' in word:
                    state = 0
                else:
                    state += 1
                    state = min(state, len(self.encoders) - 1)
                if word == __TEXT_END__:
                    return ' '.join(result)
                result.append(word)
        raise ValueError("No __TEXT_END__ has been reached")

    def handle_exceptional_str(self, s: str, state: int) -> str:
        res = ''
        encoder = self.encoders[state]
        res += encoder[len(self.stats[state].stats_sorted)]
        for c in s:
            res += encoder[ord(c)]
```

```
    res += encoder[0]
    return res

def str_to_bytes(self, s: str, fill=False) -> t.Tuple[bytearray, str]:
    if fill and len(s) % 8 != 0:
        s += '0' * (8 - len(s) % 8)
    res = bytearray([
        self.str_to_byte[s[i * 8 : i * 8 + 8]]
        for i in range(0, len(s) // 8)
    ])
    if len(s) % 8 == 0:
        return res, ''
    return res, s[-(len(s) % 8):]

def split_sentences(self, text: str) -> t.Iterable[t.List[str]]:
    text = text.split(' ')
    current_sentence = []
    for word in text:
        current_sentence.append(word)
        if '.' in word or '?' in word or '!' in word:
            yield current_sentence
            current_sentence = []
    current_sentence += [__TEXT_END__]
    current_sentence += [__TEXT_END__]
    yield current_sentence
```

ПЕРЕЛІК ДЖЕРЕЛ ТА ПОСИЛАНЬ

1) Офіційний ресурс ЦЕРН [Електронний ресурс] – Режим доступу до ресурсу: https://home.cern/science/computing/storage.

2) D. Huffman. “A Method For Construction Of Minimum Redundancy Codes”. In: Proc. IRE 40 (1952), pp. 1098–1101

3) El Daher, A., & Connor, J. (2016). Compression Through Language Modeling.

4) A. V. Anisimov and I. O. Zavadskyi, "Variable-Length Prefix Codes With Multiple Delimiters," in *IEEE Transactions on Information Theory*, vol. 63, no. 5, pp. 2885-2895, May 2017, doi: 10.1109/TIT.2017.2674670.

5) Culpepper J.S., Moffat A. (2005) Enhanced Byte Codes with Restricted Prefix Properties. In: Consens M., Navarro G. (eds) String Processing and Information Retrieval. SPIRE 2005. Lecture Notes in Computer Science, vol 3772. Springer, Berlin, Heidelberg

6) UTF-8, a transformation format of ISO 10646, RFC 3629, November 2003

7) Андрушенко О.Ю. Вступ до германського мовознавства. – Житомир: ЖДУ, 2010. – 178 с.

8) Офіційний сайт Project Gutenberg [Електронний ресурс] – Режим доступу до ресурсу: https://www.gutenberg.org