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 бакалавра

за спеціальністю 122 Комп'ютерні науки

на тему:

УДОСКОНАЛЕННЯ КОМПРЕСІЇ ТЕКСТІВ НА ОСНОВІ МУЛЬТИРОЗДІЛЬНИКОВИХ КОДІВ

Виконав студент 4 курсу
Морозюк Антон Юрійович _____
(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

ЕНТРОПІЙНЕ КОДУВАННЯ, КОДИ ФІБОНАЧЧІ, КОДИ ХАФФМАНА, МУЛЬТИРОЗДІЛЬНИКОВІ КОДИ, РЕВЕРСИВНІ МУЛЬТИРОЗДІЛЬНИКОВІ КОДИ, СТИСНЕННЯ БЕЗ ВТРАТ, СТИСНЕННЯ ТЕКСТОВИХ ДАНИХ.

Об'єктом дослідження є кодування текстових даних за допомогою реверсивних мультироздільникових кодів та WRT. В роботі було проведено аналіз кодування англомовних тестів за допомогою мульроздільникових кодів і WRT, а також їх комбінування з архіваторами *bzip2*, *7zip*, *zstd* та *gzip*.

Метою дослідження є порівняння ефективності кодування текстів за допомогою *WRT* та мультироздільникових кодів в контексті використання їх як передобробки для інших архіваторів, а також перевірити ефективність застосування прийомів *WRT* для покращення кодування реверсивними мультироздільниковими кодами.

Методом досліджень є розробка реалізації алгоритмів *WRT* та стискання за допомогою реверсивних мультироздільникових кодів на мові програмування *C++* з використанням компілятору *g++*, а також написання відповідних тестів для перевірки ефективності даних кодувань. Також для розробки було використано фреймворк *Google Test* і бібліотеку *Boost*. Реалізації архіваторів *bzip2*, *7zip*, *zstd* та *gzip* було взято з відкритих джерел.

Результатами роботи є порівняння ефективності стискання за допомогою реверсивних мультироздільникових кодів та *WRT* у якості передобробки для архіваторів *bzip2*, *7zip*, *zstd* та *gzip*, а також перевірка ефективності застосування підходів *WRT* для покращення ефективності стискання за допомогою реверсивних мультироздільникових кодів.

Отримані результати є ще одним аргументом на користь використання реверсивних мультироздільникових кодів для стискання текстових даних, а також можуть використовуватися у подальшому для підвищення їх ефективності.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ

WRT – Word Replacing Transformation, алгоритм описаний у [13].

EOL – End-Of-Line encoding – один з етапів WRT.

$R_{2,3,5}$ − реверсивні мультироздільникові коди з роздільниками довжин 2, 3 і 5.

$R_{2,4,inf}$ − реверсивні мультироздільникові коди з роздільниками довжин 2, 4 і нескінченність.

ВСТУП

Оцінка сучасного стану об’єкту розробки. У сучасному світі завдяки мережі Інтернет з’явився доступ до величезних об’ємів інформації. Крім того, цей об’єм зростає експотенційно. В зв’язку з цим все важливішими стають методи, якими опрацьовується ця інформація. Одним з таких важливих методів є стискання інформації, адже без цього її передавання і зберігання було б значно складнішим. Не менш важливим є і пошук у закодованих даних, адже без цього для кожного їх використання довелося б розкодовувати увесь об’єм наявної інформації.

При цьому для стискання даних важлива як швидкість кодування/декодування так і його ефективність. Зазвичай, ці величини обернено залежать одна від одної – що швидше працює кодування, тим гірше воно стискає.

Одним з перших розповсюджених ентропійних методів стискання даних були коди Хаффмана, за допомогою яких кожен символ вихідного алфавіту кодувався послідовністю бітів тим коротшою, чим він частіше зустрічався. Їх недоліком було те, що максимально ефективними вони було на алфавітах, кратних степеням двійки.

Пізніше з’явилося арифметичне кодування, що дало змогу кодувати кожен символ дробною кількістю бітів, але через складність алгоритму їх було дуже важко реалізувати, також вони працювали повільно.

На сьогодні існує дуже багато різноманітних методів стискання даних і здебільшого в пріоритет ставиться швидкість кодування/декодування, а не ефективність стискання.

Актуальність роботи та підстави для її виконання. Розглянуті в даній роботі реверсивні мультироздільникові коди є дуже перспективним об’єктом досліджень серед інших алгоритмів стискання даних – вони значно швидше працюють ніж їхній аналог(коди Фібоначчі) при цьому показуючи схожу чи кращу степінь стискання. Хоча вони і не є достатньо швидкими в порівнянні з

алгоритмами, що зараз широко використовуються, проте подальше покращення мультироздільникових кодів може привести до того, що вони стануть ефективнішими за популярні алгоритми.

Варто зазначити, що кожен алгоритм стиснення завжди проходить цілу низку різних покращень та оптимізацій перед тим як стати широковживаним, тому навіть незначне покращення алгоритму є важливим.

Мета й завдання роботи. Метою роботи є порівняти *WRT* та реверсивні мультироздільникові коди як методи передобробки для інших алгоритмів стискання даних, зокрема *bzip2*, *7zip*, *zstd* та *gzip*. Також спробувати використати підходи *WRT* для покращення ефективності кодування мультироздільниковими кодами і перевірити ефективність таких покращень експериментально.

Об'єкт і методи розроблення. Об'єктом даної роботи є застосування мультироздільникових кодів і *WRT* в якості передобробки для архіваторів *bzip2*, *7zip*, *zstd* та *gzip*. Під час розробки особливу увагу було приділено зручності тестування нових підходів на різних типах та об'ємах вхідних даних, а також зручність включення/виключення різних етапів *WRT*.

Можливі сфери застосування. Результати дослідження можуть бути використані у будь-яких системах, що працюють зі стисненням текстових даних, для яких швидкодія не є критичною і можна нею пожертвувати заради покращення ефективності стиснення. Також результати даної роботи можуть стати підставою перейти до реверсивних мультироздільникових кодів тим, хто використовує алгоритми на основі *WRT* для стискання текстових даних.

Взаємозв'язок з іншими роботами. Дана робота базується на кодах описаних у [12] та алгоритмі передобробки тексту, описаному в [13].

РОЗДІЛ 1. ХАРАКТЕРИСТИКА АЛГОРИТМІВ СТИСКАННЯ ДАНИХ

1.1. Методи стискання даних

Стискання даних (англ. *data compression*) – алгоритмічне перетворення даних з метою зменшення їх обсягу [2, 4, 6]. Застосовується для більш раціонального використання пристроїв зберігання та передачі даних. Синонімами є упаковка даних, компресія, стискуюче кодування, кодування джерела. Зворотна дія називається відновленням даних (розпакуванням, декомпресією).

Стискання базується на надлишку інформації, яка міститься у вхідних даних. Цей надлишок зазвичай виникає через одну з двох причин:

- у природномовному тексті зустрічається багато повторень (одні й ті ж самі слова та словосполучення використовуються кілька разів);
- деякі символи зустрічаються частіше за інші.

У першому випадку можна замінити часто повторювані слова та фрази на коротші, таким чином скоротивши загальний обсяг. А через те, що одні символи зустрічаються частіше за інші можна на їх кодування витрачати менше бітів, ніж зазвичай, при цьому на символи, що зустрічаються рідше буде витрачатись більше бітів, ніж зазвичай (такий підхід називається ймовірнісне стиснення).

У другому випадку скорочення обсягу даних досягається за рахунок заміни даних, які часто зустрічаються, короткими кодовими словами, а рідкісних – навпаки довгими (ентропійне кодування).

Варто зауважити, що стискати будь-які дані з випадковим розподілом символів неможливо. Також неможливо стискати зашифровані дані, оскільки вони зазвичай виглядають як випадкова послідовність бітів. Тому перед стисканням текстових даних потрібно наперед чітко знати, що вхідний текст не

є випадковим набором символів, інакше будь які алгоритми стискання лише збільшать його розмір.

Методи стискання даних можна розділити на два типи: з втратами та без втрат [2, 6].

Методи стискання без втрат (*loseless*) гарантують, що декодувати дані будуть в точності збігатися з вихідними.

Методи стиснення з втратами (*lossy*) можуть спотворювати вихідні дані, наприклад за рахунок видалення несуттєвих частини даних, після чого повне відновлення неможливо.

Перший тип стиснення застосовують, коли дані важливо відновити після стиснення в неспотвореному вигляді, це важливо для текстів, числових даних, тощо. Декодування таких даних, за визначенням, нічого не видаляє з вихідних даних. Стиснення досягається тільки за рахунок іншого, більш економічного, представлення даних.

Другий тип стискання застосовують, в основному, для відео, зображень і звуку. За рахунок втрат може бути досягнута більш висока ступінь стиснення. У цьому випадку втрати при стисканні означають несуттєве спотворення зображення (звуку), які в цілому не перешкоджають нормальному сприйняттю, але при порівнянні оригіналу та відновленої після стиснення копії можуть бути помічені. Проте ми все ще можемо розрізнити подану інформацію. Це пов’язано з тим, що людина завдяки особливостям своїх органів чуттів може коректно сприймати інформацію низької якості.

Очевидно, що стиснення з втратами недоцільно застосовувати до текстових даних, оскільки частина даних буде втрачена. І лише стискання без втрат має сенс для текстів, оскільки, на відміну від, наприклад, зображень, для них втратах хоча б частини тексту буде критичною, тоді як для зображень це не буде важливим.

Крім того варто зазначити, що алгоритми стискання з втратами також називаються необоротними, а без втрат – оборотними.

Крім того, можна виділити наступні методи стискання інформації:

- методи стискання загального призначення (*general-purpose*), методи, які не залежать від фізичної природи вхідних даних і, як правило, орієнтовані на стискання текстів, програм, об'єктних модулів і бібліотек, тощо, тобто даних, які в основному і зберігаються в ЕОМ;
- спеціальні методи стискання (*special*), які орієнтовані на стиснення даних відомої природи, наприклад, звуку, зображень, тощо. Такі методи за рахунок знання специфічних особливостей даних, що підлягають стисненню, досягають істотно кращої якості та / або швидкості стискання, ніж при використанні методів загального призначення.

За визначенням, методи стискання загального призначення – без втрат, а спотворюють вихідні дані тільки спеціальні методи стиснення. Як правило, спотворення допустимі тільки при обробці різних сигналів (звуку, зображення, даних з фізичних датчиків), коли відомо, яким чином і до якої міри можна змінити дані без втрати їх якостей для кінцевого споживача.

Основним критерієм відмінності між алгоритмами стискування є саме наявність або відсутність втрат. У загальному випадку алгоритми стиснення без втрат універсальні в тому сенсі, що їх застосування можливо для даних будь-якого типу, в той час як можливість застосування стиснення з втратами повинна бути аргументована. Для деяких типів даних спотворення в принципі не припустимі. Зокрема:

- символічні дані, зміна яких неминуче призводить до зміни їх семантики: програми та їх вихідні тексти, двійкові масиви, тощо;
- життєво важливі дані, зміни в яких можуть призвести до критичних помилок (наприклад, одержувані з медичної вимірювальної апаратури або контрольних приладів літальних, космічних апаратів, тощо).

В той же час, багаторазово можуть піддаватися стисненню і відновленню проміжні дані при багатоетапній обробці графічних, звукових і відеоданих.

1.2. Критерії оцінки методів стискання

Основними властивостями будь-якого алгоритму стискання даних є [1, 4]:

- якість (коефіцієнт або ступінь) стискання;
- швидкість кодування і декодування, що визначаються часом, що витрачається на кодування і декодування даних;
- обсяг необхідної пам'яті.

Розглянемо кожен з цих критеріїв більш детально.

Коефіцієнт стиснення – це основна характеристика алгоритму стискання. Вона визначається як відношення обсягу вихідних нестиснутих даних до обсягу отриманих стиснутих, тобто:

$$k = \frac{S_o}{S_c}$$

де k – коефіцієнт стискання;

S_o – обсяг вихідних даних;

S_c – обсяг стиснутих даних.

Таким чином, чим значення коефіцієнта стиснення, тим алгоритм ефективніше. Варто зазначити:

- якщо $k = 1$, то алгоритм не здійснює стиснення, тобто вихідне повідомлення дорівнює за обсягом вхідному;
- якщо $k < 1$, то алгоритм створює повідомлення більшого розміру, ніж вихідне, тобто, здійснює «непотрібну» роботу.

Ситуація, коли $k < 1$ цілком можлива при стисканні. Принципово неможливо отримати алгоритм стиснення без втрат, який за будь-яких даних утворював би на виході дані меншого або рівного обсягу. Обґрунтування цього факту полягає в тому, що оскільки кількість різних повідомлень обсягом n біт становить рівно 2^n, кількість різних повідомлень з обсягом меншим або рівним n (при наявності хоча б одного повідомлення меншого обсягу) буде менше 2^n. Це означає, що неможливо однозначно зіставити всі вихідні повідомлення стисненим: або деякі вихідні повідомлення не матимуть стиснутого

представлення, або декільком вихідним повідомленнями буде відповідати одне і те ж стиснуте, а значить їх не можливо відрізнити. Однак навіть коли алгоритм стиснення збільшує розмір вихідних даних, легко добитися того, щоб їх обсяг гарантовано не міг збільшитися більш, ніж на 1 біт. Тоді навіть у найгіршому випадку отримаємо нерівність:

$$k \geqslant \frac{S_o}{S_o + 1}$$

Відбувається це в такий спосіб: якщо обсяг стиснутих даних менше обсягу вихідних, повертаємо стиснуті дані, додавши до них «1», інакше оновлюємо вихідні дані, додавши до них «0».

Коефіцієнт стискання може бути постійним (деякі алгоритми стиснення звуку, зображення тощо, наприклад *A*-закон, *μ*-закон, *ADPCM*, усічене блокове кодування), так і змінним. В останньому випадку він може бути визначений або для кожного окремого повідомлення, або оцінено за деякими критеріями:

- середній (зазвичай визначається за деяким тестовим набором даних);
- максимальний (випадок найкращого стискання);
- мінімальний (випадок найгіршого стискання).

При цьому коефіцієнт стискання з втратами дуже залежить від допустимої похибки стиснення або якості, яка зазвичай виступає як параметр алгоритму. У загальному випадку постійний коефіцієнт стискання здатні забезпечити лише методи стиснення даних з втратами.

В області стиснення даних, як це часто трапляється, діє закон рівноваги: алгоритми, що використовують більше ресурсів (часу і пам’яті), зазвичай досягають кращої якості стиснення, і навпаки: менш ресурсомісткі алгоритми за якістю стиснення, як правило, поступаються більш ресурсоємним.

Таким чином, побудова оптимального з практичної точки зору алгоритму стискання даних є досить нетривіальним завданням, так як необхідно досягнути досить високої якості стиснення при невеликому обсязі використовуваних ресурсів.

Зрозуміло, що критерії оцінки методів стиснення з практичної точки зору сильно залежать від передбачуваної області застосування. Наприклад, при використанні стиснення в системах реального часу необхідно забезпечити високу швидкість кодування і декодування; для вбудованих систем критичний параметр – обсяг необхідної пам'яті; для систем довгострокового зберігання даних – якість стиснення і / або швидкість декодування і т. д.

Надійність програмних систем і комплексів дуже важлива і забезпечується як безпомилковістю програмування і дизайну, так і характеристиками використаних алгоритмів. Якщо кількість помилок в основному визначається повнотою й якістю тестування (а також кваліфікацією і культурою програмування) і мало залежить від волі розробника, то вибір алгоритмів – цілком керований і контрольований процес.

Для забезпечення кінцевого і заздалегідь відомого часу стиснення, необхідно, щоб алгоритм володів добре детермінованим часом роботи і заздалегідь відомим об’ємом необхідної пам’яті.

Якщо з теоретичної точки зору поліноміальні алгоритми, що володіють поліноміальною або експоненціальною складністю, вважаються хорошим рішенням проблеми, то на практиці прийнятні тільки алгоритми з лінійною або лінійно-логарифмічною тимчасовою складністю, причому вкрай бажано, щоб середній час роботи (на типових даних) було лінійним.

1.3. Класифікація алгоритмів стискання даних без втрат

Основою всіх методів стискання інформації є проста ідея: якщо відображати елементи, що часто використовуються короткими кодами, а ті, що зрідка використовуються – довгими, то для зберігання певного обсягу інформації знадобиться менший обсяг пам’яті, ніж якби всі елементи відображалися кодами однакової довжини. Даний факт є давно відомим. Саме Азбука Морзе, винайдена в 1838 році була першим відомим випадком стиснення даних. В ній символи, що часто зустрічаються представлені

короткими послідовностями точок і тире, а символи, що іноді зустрічаються, навпаки – довгими.

На рис. 1.1 представлена класифікація найбільш розповсюджених алгоритмів стискання даних без втрат, які використовуються для стискання текстової інформації [6].

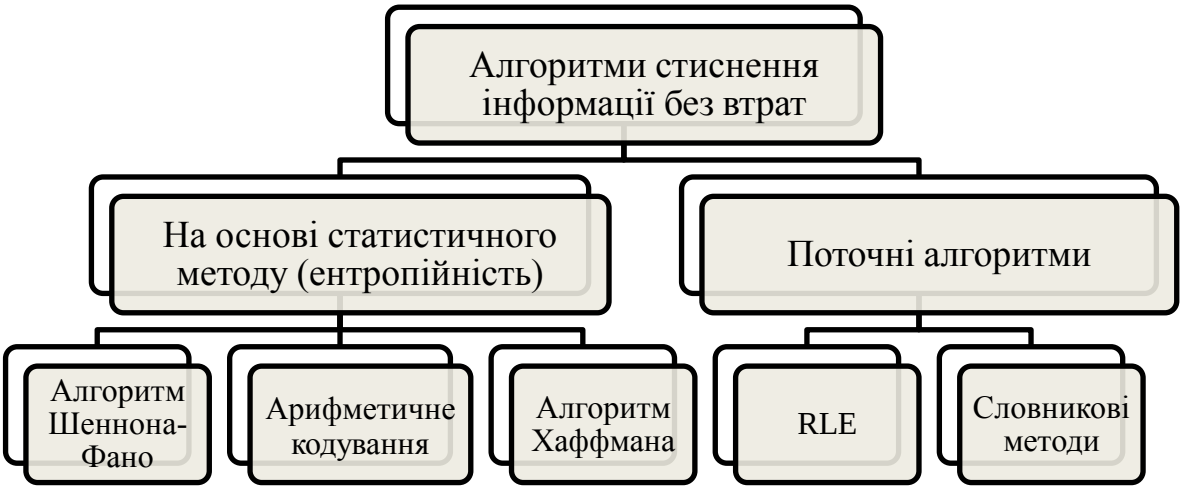

Рисунок 1.1 – Класифікація алгоритмів стискання інформації

У загальному випадку можна виділити два базові варіанти, на яких будуються алгоритми стиснення.

Перша група методів – це статистичні методи стиснення. Ентропійне кодування – це кодування послідовності значень з можливістю однозначного відновлення з метою зменшення обсягу даних (довжини послідовності) за допомогою усереднення імовірностей появи елементів в закодованій послідовності.

При цьому передбачається, що до кодування окремі елементи послідовності мають різну ймовірність появи. Після кодування в результуючій послідовності ймовірності появи окремих символів практично однакові (ентропія на символ максимальна).

Згідно теореми Шеннона, існує межа стиснення без втрат, яка залежить від ентропії джерела. Чим більш передбачувані одержувані дані, тим краще їх

можна стиснути. Випадкова незалежна рівноймовірна послідовність стискання без втрат не піддається.

У сорокових роках XX століття вчені, які працювали в галузі інформаційних технологій, ясно зрозуміли, що можна розробити такий спосіб зберігання даних, при якому простір буде витрачатися більш економно. Клод Шеннон, вивчаючи нюанси розходжень між семантикою (*semantics*) (що значить деяка сутність) і синтаксисом (*syntax*) (як виражається деяка сутність), розробив більшість базових понять цієї теорії. Розуміння того, що одне і те ж значення (семантика) може бути реалізовано різними способами (синтаксис), приводить до закономірного питання: «Який спосіб вираження чогось є найбільш економічним?» Пошук відповіді на це питання привів Шеннона до думки про ентропію, яка співвідноситься з кількістю корисної інформації, що міститься у файлі. Методи стиснення намагаються збільшувати ентропію файлу, тобто зменшувати довжину файлу, зберігаючи при цьому всю інформацію.

Пізніше, коли в 1949 році мейнфрейм-комп'ютери почали завойовувати популярність, Клод Шеннон і Роберт Фано винайшли кодування, назване на їх честь – алгоритм Шеннона-Фано. Це шифрування привласнює коди символів у масиві даних за теорією ймовірності [4].

Даний метод вирізняється своєю простотою. При цьому беруться вихідні повідомлення $m_{(i)}$ та ймовірності їх появи $P_{(m\,(i)}$. Повідомлення впорядковуються так, щоб ймовірність і-го повідомлення була не більша $i + 1$ -го. Цей список ділиться на дві групи з приблизно рівною інтегральної імовірністю. Кожному з повідомленням з групи 1 присвоюється 0 в якості першої цифри коду. Повідомленням з другої групи ставляться у відповідність коди, що починаються з 1. Кожна з цих груп ділиться на дві аналогічним чином і додається ще одна цифра коду. Процес триває до тих пір, поки не будуть отримані групи, що містять лише одне повідомлення. Кожному з повідомленням в результаті буде присвоєно код x з довжиною – $lg\ (P_{(x)})$. Це слушно, якщо можливий поділ на підгрупи з абсолютно рівною сумарною

ймовірністю. Якщо ж це неможливо, деякі коди будуть мати довжину – $lg\,(P_{(x)}) + 1$. Тому, алгоритм Шеннона-Фано не гарантує оптимального кодування.

Арифметичне кодування – один з алгоритмів ентропійного стиснення. На відміну від алгоритму Хаффмана, не має жорсткої постійної відповідності вхідних символів групам біт вихідного потоку. Це дає алгоритму велику гнучкість в поданні дрібних частот зустрічальності символів.

Використовуючи метод арифметичного кодування, можна досягти майже оптимального представлення для заданого набору символів та їх ймовірностей. Алгоритми стиснення даних, що використовують у своїй роботі метод арифметичного кодування, перед безпосереднім кодуванням формують модель вхідних даних на підставі кількісних або статистичних характеристик, а також, знайдених з кодованої послідовності повторень або патернів – будь якої додаткової інформації, що дозволяє уточнити ймовірність P появи символу в процесі кодування. Очевидно, що чим точніше визначена або передвіщена ймовірність символу, тим вища ефективність стиснення.

На відміну від алгоритму Хаффмана, метод арифметичного кодування показує високу ефективність для дрібних нерівномірних інтервалів розподілу ймовірностей кодованих символів. Однак у випадку рівноймовірного розподілу символів, наприклад для рядка біт 010101 ... 0101 довжини S метод арифметичного кодування наближається до коду Хаффмана і навіть може займати на один біт більше. Як правило, перевершує алгоритм Хаффмана по ефективності стиску, дозволяє стискати дані з ентропією, меншою 1 біта на кодований символ, але деякі версії мають патентні обмеження від компанії IBM.

Алгоритм Хаффмана – адаптивний алгоритм оптимального префіксного кодування алфавіту з мінімальною надлишковістю. Був розроблений в 1952 році аспірантом Массачусетського технологічного інституту Девідом Хаффманом при написанні ним курсової роботи. В даний час використовується в багатьох програмах стиснення даних.

Ідея, яка покладена в основу алгоритму Хаффмана досить проста та заснована на частоті появи символу в послідовності [3]. Тому цей алгоритм відносять до алгоритмів статичстичного аналізу. Варто додати, що символом вважається деякий повторюваний елемент початкового ряду – як друкований знак, так і будь-яка бітова послідовність. Отже, замість того, щоб кодувати усі символи однаковою кількістю бітів, кодування відбувається наступним чином: символи, які зустрічаються частіше, кодуються меншим числом біт, а символи, які зустрічаються рідше – навпаки більшим. Ця ідея заснована на спостереженні, що деякі символи зі стандартного набору в довільному тексті можуть зустрічатися частіше середнього періоду повтору, а інші рідше. Відповідно, якщо використати короткі послідовності бітів для розповсюджених символів, то сумарний об’єм файлу зменшиться. Результатом такої систематизації даних буде двійкове дерево. Алгоритм є двічі прохідним, тобто реалізується в два етапи. На першому етапі будується частотний словник та генеруються коди. Під час другого етапу відбувається саме кодування.

Класичний алгоритм Хаффмана на вході отримує таблицю частот повторюваності символів в повідомленні. Далі на підставі цієї таблиці будується дерево кодування Хаффмана (Н-дерево). Отже, алгоритм Хаффмана володіє властивістю префіксності та мінімальною збитковістю. Для отримання кодів необхідно виконати наступні дії:

1. Символи вхідного алфавіту утворюють список вільних вузлів. Кожен лист має вагу, яка може дорівнювати або ймовірності, або кількості входжень символу в повідомлення, що стискається.

2. Вибираються два вільних вузла дерева з найменшими вагами.

3. Створюється їх батько з вагою, рівною їх сумарній вазі.

4. Батько додається в список вільних вузлів, а два його нащадка видаляються з цього списку.

5. Однією у дузі, що виходить з батьків, ставиться у відповідність біт 1, а інший – біт 0.

6. Кроки, починаючи з другого, повторюються до тих пір, поки в списку вільних вузлів не залишиться тільки один вільний вузол. Він і буде вважатися коренем дерева.

Класичний алгоритм Хаффмана має ряд істотних недоліків. По-перше, для відновлення вмісту стиснутого повідомлення декодер повинен знати таблицю частот, якою користувався кодер. Отже, довжина стиснутого повідомлення збільшується на довжину таблиці частот, яка повинна надсилатися перед даними, що може звести нанівець всі зусилля зі стиснення повідомлення. Крім того, необхідність наявності повної частотної статистики перед початком власне кодування вимагає двох проходів за повідомленням: одного для побудови моделі повідомлення (таблиці частот і Н-дерева), іншого для власне кодування. По-друге, надмірність кодування повертається в нуль лише в тих випадках, коли ймовірності кодованих символів є зворотними ступенями числа 2. По-третє, для джерела з ентропією, що не перевищує 1 (наприклад, для двійкового джерела), безпосереднє застосування коду Хаффмана є безглуздим.

Кодування довжин серій (англ. *Run-length encoding, RLE*) або кодування повторів – простий алгоритм стиснення даних, який оперує серіями даних, тобто послідовностями, в яких один і той же символ зустрічається кілька разів поспіль [7]. При кодуванні рядок однакових символів, що складають серію, замінюється рядком, що містить сам повторюваний символ і кількість його повторів.

Існують деякі модифікації алгоритму, які дозволяють вирішити проблему з ефективністю, якщо зустрічається велика кількість неповторюваних символів. Для цього можна зробити так, що байт, який кодує кількість повторів, буде також зберігати інформацію про їх наявність. Наприклад, якщо перший біт дорівнює 1, то наступні сім вказуватимуть на кількість повторів відповідного символу, а якщо перший біт дорівнює 0, то наступні біти вказують на кількість символів, які потрібно брати без повторів. Алгоритм розрахований на зображення з великими областями, заповненими одним кольором. Сюди

відносять так звану ділову графіку: гістограми, діаграми, графіки. При застосуванні до такого виду зображень алгоритм буде давати високу ефективність.

Тільки в 1970-х з появою інтернету і онлайн-сховищ, були реалізовані повноцінні алгоритми стиснення. Коди Хаффмана динамічно генерувалися на базі вхідної інформації. Ключова відмінність між кодуванням Шеннона-Фано і кодуванням Хаффмана полягає в тому, що в першому дерево ймовірності будувалося знизу вгору, створюючи неоптимальний результат, а в другому - зверху вниз.

Словникові методи – розбиття даних на слова і заміна їх на індекси в словнику [1, 2]. Ці методи є найбільш поширеними для стиснення даних в даний час. Є природним узагальненням RLE. У найбільш поширеному варіанті реалізації словник поступово доповнюється словами з вихідного блоку даних в процесі стиснення. Основним параметром будь-якого словникового методу є розмір словника. Чим більше словник, тим більше ефективність. Однак для неоднорідних даних надмірно великий розмір може бути шкідливий, тому що при різкій зміні типу даних словник буде заповнений неактуальними словами. Для ефективної роботи даних методів при стисненні потрібна додаткова пам’ять. Істотною перевагою словникових методів є проста і швидка процедура розпакування. Додаткова пам’ять при цьому не вимагається. Така особливість вкрай важлива, якщо необхідний оперативний доступ до даних. До методів стиснення з використанням словника відносяться наступні алгоритми: *LZ77/78*, *LZW*, *LZO*, *DEFLATE*, *LZMA*, *LZX*, *ROLZ*.

У 1977 два дослідники з Ізраїлю Абрахам Лемпел і Якоб Зів висунули ідею формування «словника» загальних послідовностей даних. При цьому стискання даних здійснюється за рахунок заміни записів відповідними кодами зі словника. Існують два алгоритми, в даний час відомі як *LZ77* і *LZ78*. Вони вже не вимагають включення словника даних в архів, тому що якщо ви формуєте словник певним способом, програма декодування може його відновлювати безпосередньо з ваших даних. На жаль, *LZ77* і *LZ78* витрачають

багато часу на створення ефективного словника. Алгоритм *LZ77* коротко називають варіантом «пам'ятай недавню історію», а *LZ78* «пам'ятай те, що використовував», завдяки головним ідеям, які вони використовують.

Алгоритм *LZW (Lempel-Ziv-Welch)* – це універсальний алгоритм стиснення даних без втрат, створений Abraham Lempel, Jacob Ziv і Terry Welch. Він був опублікований Велчем в 1984 році, в якості покращеної реалізації алгоритму *LZ78*. Алгоритм розроблений так, щоб його можна було швидко реалізувати, але він не обов'язково оптимальний, оскільки він не проводить ніякого аналізу вхідних даних.

Даний алгоритм при стисненні динамічно створює таблицю перетворення рядків: певним послідовностям символів ставляться у відповідність групи біт фіксованої довжини (зазвичай 12-бітові). Таблиця ініціалізується усіма 1-символьними рядками (в разі 8-бітних символів – це 256 записів). Внаслідок кодування, алгоритм переглядає текст символ за символом, і зберігає кожен новий, унікальний 2-символьний рядок в таблицю у вигляді пари код / символ, де код посилається на відповідний перший символ. Після того як новий 2-символьний рядок збережений у таблиці, на вихід передається код першого символу. Коли на вході читається черговий символ, для нього по таблиці знаходиться рядок максимальної довжини, що вже зустрічався раніше, після чого в таблиці зберігається код цього рядка з наступним символом на вході; на вихід видається код цього рядка, а наступний символ використовується в якості початку наступного рядка. Алгоритму декодування на вході потрібно тільки закодований текст, оскільки він може відтворити відповідну таблицю перетворення безпосередньо по закодованому тексту.

На момент своєї появи алгоритм *LZW* давав кращий коефіцієнт стискання, для більшості додатків, ніж будь-який інший добре відомий метод. Він став першим методом стиснення даних, який широко використовується на комп'ютерах.

РОЗДІЛ 2. МУЛЬТИРОЗДІЛЬНИКОВІ КОДИ ТА МЕТОДИ ЇХ ОПТИМІЗАЦІЇ

2.1. Мультироздільникові коди

Стискання інформації є галуззю, яка досить добре досліджена в інформатиці. Найпоширеніший клас методів стискання даних ґрунтується на використанні частотних кодів. Основна ідея цього методу полягає в тому, що словам вхідного алфавіту з більшими частотами потрапляння відповідають коротші кодові слова. Якщо кількість слів у вхідному алфавіті дорівнює степеню двійки, найвищий ступінь стискання забезпечують коди Хафмана, в іншому випадку найефективнішими є арифметичні коди. Ці коди дають дуже щільне наближення до границі Шеннона, яка визначає теоретичну межу можливостей методів стискання.

Проте, окрім коефіцієнта стискання, важливими є й інші характеристики зазначених методів, зокрема обчислювальна складність кодування, декодування та пошуку в стиснутих файлах, а також стійкість до перешкод, що можуть виникати в каналах зв’язку та на носіях інформації. Коди Хафмана та арифметичні коди не є оптимальними з точки зору жодної характеристики, крім коефіцієнта стискання. Зокрема, вони є зовсім нестійкими до перешкод, оскільки помилка навіть в одному біті може призвести до викривлення всієї частини повідомлення, що йде за цим бітом, а також роблять неможливим пошук даних у стиснутих файлах без їхньої деархівації. Ці недоліки вирішують тегові коди Хафмана, які, на жаль, досить суттєво погіршують коефіцієнт стискання.

Альтернативним варіантом є коди Фібоначчі [8]. Це універсальний код для натуральних чисел (1, 2, 3 ...), що використовує послідовності бітів. Оскільки комбінація 1…1 заборонена в системі числення Фібоначчі, її можна використовувати як маркер кінця запису. Для складання коду Фібоначчі по

запису числа необхідно переписати цифри в зворотньому порядку (так, що старша одиниця виявляється останнім символом) і приписати в кінці ще раз 1. Тобто, кодова послідовність має вигляд:

ε 2 ε 3... ε n 1,

де n – номер самого старшого розряду з одиницею.

Як і тегові коди, коди Фібоначчі є синхронізованими, тобто кодові слова в них закінчуються спеціальними послідовностями – роздільниками (у кодах Фібоначчі вони мають вигляд 1…1), а отже, помилка в окремому біті з потоку закодованих слів може призвести до спотворення лише того слова, якому цей біт належить, або, якщо біт належить роздільникові, ще й наступного слова. Однак вплив помилки не може поширитися далі наступного роздільника. Найпоширенішою галуззю застосування кодів Фібоначчі є стискання природничих текстів, під час якого кожне слово тексту замінюється певним кодовим словом. За прикладними дослідженнями, код *Fib3* дає в 2–3 рази краще наближення до границі Шеннона, ніж коди *ETDC* та *SCDC*, однак має гіршу швидкість кодування, декодування та пошуку в стиснутих файлах.

Певним недоліком кодів Фібоначчі є брак властивості негайного розділення [8, 9]. Це значить, що, якщо в закодованому тексті знайдена бітова послідовність, яка відповідає певному кодовому слову, ще не гарантовано, що вона є цим кодовим словом, оскільки коди Фібоначчі не є суфіксними: одні кодові слова можуть бути суфіксами інших (хоча ці коди є префіксними). Більше того, найкоротші кодові слова в кодах Фібоначчі складаються лише з одного роздільника вигляду 1…1 і тому можуть «зливатися» в довгі послідовності з одиниць. Тому для відокремлення одного кодового слова від іншого варто перевіряти в загальному випадку необмежену кількість бітів (до тих пір, поки не буде знайдено початок серії одиниць).

Нині з’явився новий клас префіксних кодів – мультироздільникові коди [9, 10]. Їх так назвали тому, що кодові слова можуть відокремлюватися не одним, а кількома роздільниками вигляду 01…10. Деякі представники цього сімейства кодів у разі застосування до стискання природничих текстів мають

коефіцієнт стискання на 10–30 % ближчий до границі Шеннона, ніж для коду *Fib3*, на 20 % вищу швидкість і в чотири рази менші витрати пам'яті декодувального алгоритму, при цьому є значно кращими з огляду на термінове розділення: для виявлення в стиснутому файлі точного положення кодового слова досить продивитися лише невелику фіксовану кількість бітів, які передують відповідній цьому слову бітовій послідовності.

У статтях І. Завадського [9, 10] наводиться наступне визначення мультироздільникових кодів. Припустимо, що m_1 або… або m_t – деякі цілі числа, такі, що $0 < m_1 <...< m_t$. Означення мультироздільникового коду $D_{m1...\,mt}$ містить усі слова вигляду 1…10 із m_1 або… або m_t одиницями, а також усі слова, що задовольняють наступні умови:

- слово не починається з послідовності 1…10, що містить m_1 або m_2 або… або m_t одиниць;
- слово закінчується послідовністю 01…10, що m_1 або m_2 або… або m_t одиниць;
- слово не містить послідовності 01…10 із m_1 або m_2 або… або m_t одиницями ніде, крім кінця.

Роздільниками в такому коді є послідовності вигляду 01…10, що містять m_1 або m_2 або… або m_t одиниць, однак код містить також слова вигляду 1…10 із m_1 або m_2 або… або m_t одиниць, які утворюють роздільник разом із останнім нулем попереднього кодового слова в тексті.

Взагалі за результатами досліджень багатьох вчених коди із більшою кількістю роздільників мають гіршу асимптотичну щільність, утім містять більше коротких слів. Подібну закономірність пов'язують із довжиною роздільника: чим коротшою вона є, тим більше коротких слів містить код, але тим гіршу асимптотичну щільність він має. З огляду на мету стискання текстової інформації, найбільш ефективними виходять коди з роздільниками довжини 2.

Що стосується кодування/декодування мультироздільникових кодів, то під час кодування використовується словник, у якому слова тексту відсортовані

за спаданням частот, а кодові слова відсортовані за зростанням довжин. Під час декодування застосовується зворотний алгоритм: необхідно побудувати відображення з множини кодових слів на множину слів тексту. З метою часової оптимізації декодування слова вихідного тексту доцільно зберігати за допомогою структури даних, що забезпечує швидкий доступ до своїх елементів, а найефективнішою з таких структур є масив із числовими індексами.

Отже, мультироздільникові коди з метою застосування до стискання текстової інформації дають кращий коефіцієнт стискання, характеризуються вищою швидкістю алгоритму негайного розділення, а також вищою швидкістю та меншими ємнісними витратами алгоритму декодування, ніж коди Фібоначчі [9, 10].

2.2. Реверсивні мультироздільникові коди

Якщо слова мультироздільникового коду записувати справа наліво, то отримаємо непрефіксний, але однозначно декодовний код, який називається реверсним [11, 12]. Його однозначна декодовність випливає з однозначної декодовності прямого мультироздільникового коду, оскільки будь-який алгоритм декодування прямого коду є цілком застосовним і до реверсного, якщо розглядати закодований файл справа наліво.

Нехай $M = \{m_1, m_2, \dots, m_t\}$ – множина натуральних чисел, розміщених в порядку зростання.

Тоді реверсивний мультироздільниковий код $R_{m_1,m_2,..,m_t}$складатиметься з усіх слів вигляду $01^{m_i}, i = 1, \dots, t$ і всіх інших слів, що задовольняють наступні правила:

- слово починається з $01^{m_i}0$ для деякого $m_i \in M$;
- для будь-якого $m_i \in M$ слово не закінчується на 01^{m_i};
- для будь-якого $m_i \in M$ слово не може містити послідовності $01^{m_i}0$ ніде, крім префіксу.

З такого означення випливає, що роздільниками у реверсивному мультироздільниковому коді $R_{m_1,m_2,..,m_t}$ є послідовності вигляду $01^{m_i}0$. При цьому код також містить слова вигляду $01^{m_i}, i = 1, \dots, t$, які є коротшими за роздільники, але при цьому утворюють їх разом з першим нулем наступного слова.

Далі розглянемо код $R_{2,4,5}$ так як він показує найкращі результати серед аналогів.

Основною перевагою реверсивних мультироздільникових кодів над звичайними є можливість легко згенерувати усі кодові слова динамічно. Далі покажемо як це можна зробити.

Нехай потрібно побудувати словник слів довжини L, при тому що вже є всі слова меншої довжини і $К = \{k_1, \dots, k_q\}$ – зростаюча послідовність усіх цілих чисел до m_t, що не належать M. Тоді результату можна досягти наступними кроками:

1. Для і від 1 до q повторюємо всі групи кодових слів довжин $L - k_i - 1$ і додаємо до них послідовності виду $01^{k_i}, k_i \in K$.
2. Повторюємо групу слів довжини $L - 1$ з суфіксом $01^r, r > m_t$ і додаємо до кожного кодового слова «1».
3. Якщо $L = m_i + 1$ для деякого $i \in \{1, \dots, t\}$, то додаємо слово 01^{m_i}.

На рисунку 2.1 показано приклад генерації кодових слів для $L = 8$.

Для декодування реверсивних мультироздільникових кодів можна використати наступний автомат (рис. 2.2) [11].

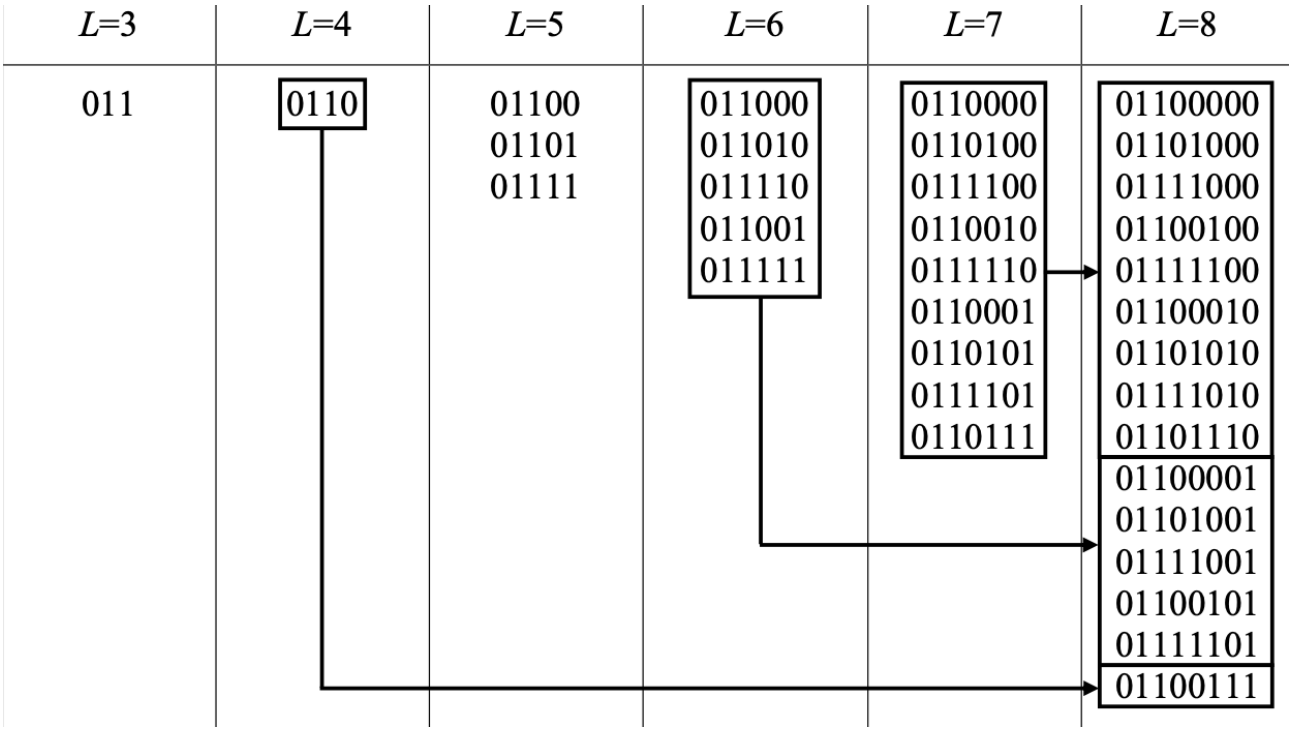

Рисунок 2.1 – Генерація кодових слів довжини менше 9

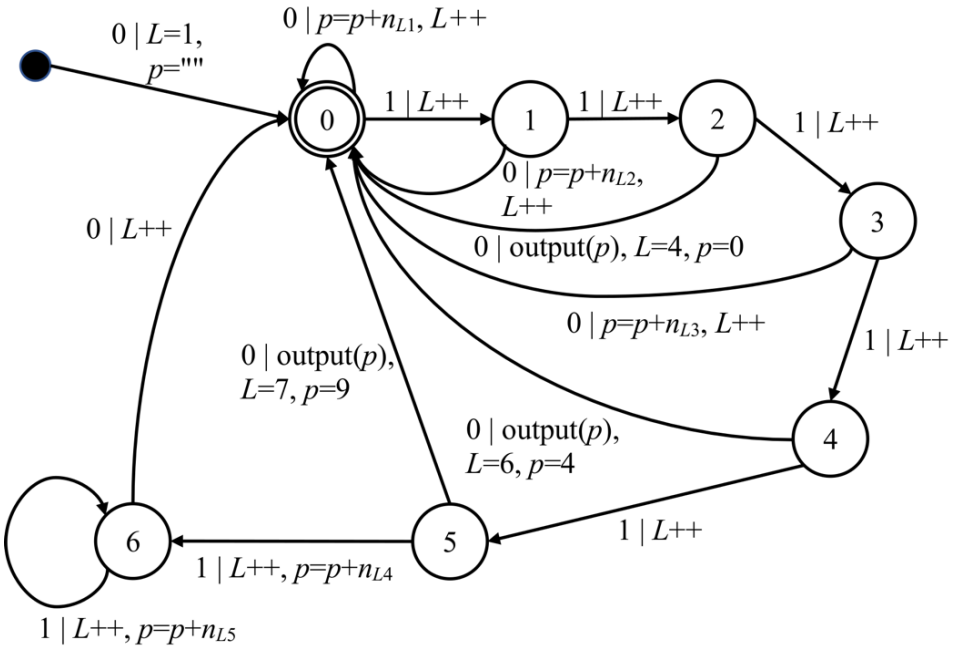

Рисунок 2.2 – Декодувальний автомат для коду $R_{2,4,5}$

Декодувальний автомат обробляє кодові слова в потоці починаючи зі стану, який відмічений замальованим колом і закінчуючи в стані 0, після роздільника на початку наступного кодового слова. Отож до закодованого файлу потрібно дописати в кінці роздільник (наприклад, 0110) для того, щоб декодування закінчилося нормально. Автомат розпізнає кодові слова біт за

бітом і обчислює значення двох змінних: L – довжина кодового слова і p – результуючий індекс кодового слова в їх масиві, відсортованих за зростанням. Для кожного переходу вхідний біт написано зліва над стрілкою, а операції над змінними – справа, притому їх порядок визначальний.

Розглянемо як працює цей автомат. Будь-яке кодове слово з $R_{2,4,5}$ може бути подано у вигляді конкатенації бітових послідовностей вигляду $01^t,\ t \geq 0$. Після обробки такої послідовності і 0 після неї автомат переходить у стан 0. Бітові послідовності 011, 01111 та 011111 можуть зустрічатися лише як префікси кодового слова. Після їх обробки автомат переходить у стан 0 з станів 2, 4 або 5, виводить індекс попереднього кодового слова p та ініціалізує p новим значенням 0, 4 або 9, що відповідає індексу кодових слів 011, 01111, 011111 відповідно. Також, після переходів $2 \to 1$, $4 \to 1$ та $5 \to 1$, L ініціалізується відповідною довжиною кодового слова, тоді як на всіх інших переходах воно просто збільшується на 1.

Після обробки послідовності 01…10 з 0, 1, 3, 6 або більше одиницями автомат переходить в стан 0 зі станів 0, 1, 3, 6 відповідно і додає до p значення n_{L_i}. Якщо i менше 5, то n_{L_i} просто дорівнює зміщенню кодового слова в масиві усіх кодових слів через додавання до нього бітової послідовності виду 01…10 з k_i одиницями $k_i \in K = \{0, 1, 3, 6\}$, коли довжина результуючого кодового слова L. Значення n_{L_5} дорівнює зміщенню кодового слова довжини $L - 1$ з 6 чи більше одиницями в кінці, в масиві усіх кодових слів через додавання одиничного біту, утворюючи вихідне кодове слово довжини L.

Але використання такого автомату для декодування тексту біт за бітом може бути дуже повільним. Проте, можна виконати байтову квантифікацію цього автомату. Це відбувається шляхом пропускання через автомат різних вхідних байтів, які утворюють різні вихідні числа. Після цього виникає таблиця переходів $TAB[a_{start}][L_{start}][x]$, яка складається з трійок (p, a_{fin}, L_{fin}),

де x це байт закодованого тексту,

a_{start} – початковий стан автомату,

a_{fin} – кінцевий стан автомату,

L_{start} – довжина розкодованої частини останнього кодового слова, який опрацьовувався до x,

L_{fin} – довжина розкодованої частини останнього кодового слова після опрацювання x,

p – вивід автомату, згенерований після проходження через нього x.

Побайтовий алгоритм декодування для коду $R_{2,4,5}$ наведено на рис. 2.3. Таблиця переходів для нього побудована за принципом, який описано вище, але для прискорення розкодування використовується багато одновимірних $TAB_j[x]$ замість однієї тривимірної. Індекс j цієї таблиці залежить від a_{start} та L_{start} наступним чином $j = L_{start} * 7 + a_{start}$ (множимо на 7, бо у автомата 7 станів). Аналогічно замість a_{fin}, L_{fin} використовується j_{fin}. Число p представлено чотирма числами $p_1,\ p_2,\ p_3,\ p_4$ індексами декодованих слів або їх декодованих частин у масиві усіх кодових слів (просто зрозуміти, що в одному байті не може бути більше чотирьох кодових слів чи їх частин, якщо найкоротший роздільник має вигляд 011).

```
1.  i ← 0, j ← 0, p ← 0
2.  while i < length of encoded text
3.      (p1, p2, p3, p4, j) ← TAB_j[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
```

Рисунок 2.3 – Побайтовий алгоритм декодування для коду $R_{2,4,5}$

Головним недоліком швидкодії цього алгоритму є робота з TAB_j – на неї витрачається ледь не половина усіх асемблерних інструкцій. Для того, щоб уникнути використання цієї таблиці виконаємо наступне: для кожного конкретного випадку створимо асемблерну вставку, яка буде виконувати усі дії і переходитиме до наступної вставки.

Таким чином ми ніби замінимо TAB_j на «масив» асемблерних вставок і переходи між ними потребуватимуть значно менше команд. Крім того, можна буде заощадити на *if*, оскільки ми будемо знати значення $p_1,\ p_2,\ p_3,\ p_4$.

Реверсні мультироздільникові коди мають такий саме коефіцієнт стискання, як і прямі мультироздільникові коди зі словником достатньо великого розміру. Проте, реверсні коди вирішують проблему немонотонності словника, яка властива прямим мультироздільниковим кодам, при цьому зберігають всі їх позитивні властивості, такі як швидкість декодування, стійкість до помилок та можливість пошуку інформації в закодованому тексті без його розкодування.

2.3. WRT

Одним з привабливих способів підвищення ступеня стискання тексту є заміна слів посиланнями на заздалегідь створений текстовий словник. У попередньому розділі було описано декілька найбільш поширених методів стискання тексту. В цьому розділі ми зосередимся на відомих методах попередньої підготовки тексту для досягнення кращого стискання [13].

Концепція заміни слів коротшими кодовими словами з заданого статичного словника має, принаймні, два недоліки. По-перше, словник повинен бути досить великим (не менше десятків тисяч слів) і підходити тільки для однієї мови. По-друге, ніякі «більш високого рівня», наприклад, пов'язані з граматикою, кореляцІЇ не враховуються. Незважаючи на ці недоліки, такий підхід до стиснення тексту виявляється привабливим. Переваги схем стиснення

тексту на основі словників *WRT* є простота створення словника (за умови наявності достатньої кількості навчального тексту на даній мові), ясність ідеї, висока швидкість обробки, співпраця з широким діапазоном існуючих компресорів і конкурентоспроможна ступінь стиснення.

Перетворення великих букв (*Capital conversion (CC)*) – це широко відома техніка попередньої обробки тексту. Вона заснована на спостереженні, що такі слова, як, наприклад, *compression* і *Compression*, майже еквівалентні, але загальні моделі стискання, наприклад, звичайний *PPM*, не можуть відразу розпізнати цю подібність. Ідея *CC* полягає в тому, щоб перетворити велику літеру, з якої починається слово, в її рядковий еквівалент і позначити цю зміну символом *flag*. Крім того, варто використовувати ще один *flag* для позначення перетворення слова з повністю великих літер в його малу форму. Обидві форми використовуються в *StarNT*.

Використання різних алфавітів для кодових слів і оригінальних слів були описані Смирновим [14]. Оскільки байти кодових слів і вихідних оригінальних слів неможливо переплутати, компресори *PPM* і *BWCA* більш точні в своїх прогнозах.

Слова в словнику *StarNT* упорядковано відповідно до їх довжини, а всередині груп слів однакової довжини використовується алфавітний порядок. Можна скоротити вихідний словник шляхом відкидання найбільш рідкісних слів. Крім того, ми допускаємо до 4 байт на кодове слово і використовуємо підалфавіт для чотирьох байтів кодового слова. Байти кодового слова видаються в зворотному порядку, тобто діапазон для останнього байта завжди має 101 значення. Такий новий словник покращує продуктивність стиснення у всіх протестованих компресорах.

Фундаментальний закон стискання свідчить, що більш часті повідомлення повинні бути представлені більш короткими кодами, ніж менш часті повідомлення. Оскільки кодові слова в *StarNT* мають змінну довжину, очевидно, що кілька найбільш популярних слів повинні бути закодовані одним байтом, тоді як менш часто використовувані слова повинні бути представлені

парами байтів і т.д . З цієї точки зору, в ідеалі слова повинні бути відсортовані відповідно до їх частоти в даному тексті. Але це складний процес, тому краще спочатку впорядкувати словник по частоті, а потім сортувати невеликими групами відповідно до словникового порядку слів, тобто сортувати справа наліво. Таке рішення покращує контекстне прогнозування на останньому етапі.

Маючи окремі алфавіти для вихідних слів і кодових слів, легко закодувати тільки префікс слова, якщо префікс збігається з якимось словом в словнику, але все слово не збігається. Оригінальний *StarNT* не може кодувати префікси слів в якості вихідного кодового слова і суфікс вихідного слова будуть склеєні та будуть неоднозначні при декодуванні. Для цієї модифікації, *StarNT* потрібна додаткова програма, наприклад, *fsep*, щоб розділити кодове слово і суфікс вихідного слова. Проте, перевага такого методу дуже незначна.

Заміна частих послідовностей з q послідовних символів, тобто q-грам, одиночними символами [13] є молодшим братом підстановки, заснованої на словах, і належить до числа найбільш часто використовуваних методів підстановок. Основна ідея полягає в тому, щоб в контекстно-орієнтованих компресорах сприяти більш швидкому навчанню контекстної статистики, і практично збільшити в компресорах *LZ77*, щоб в основному заощадити біти на наборах і довжинах послідовностей. Довжина послідовності q зазвичай не перевищує 4. Природно представляти q-грами символами, які практично ніколи не використовуються в звичайних текстах.

Кодування в кінці рядка (*End-of-Line (EOL)*) – це метод попередньої обробки тексту, заснований на спостереженні, що символи кінця є «артикуляційними» і, таким чином, псують передбачення наступних символів. Загальна ідея полягає в тому, щоб замінити символи *EOL* на пробіли і кодувати інформацію, що дозволяє виконати зворотну операцію, в окремому потоці. Ця ідея була реалізована в багатьох існуючих компресорах.

Розділові знаки, такі як крапки і коми, зазвичай ставляться відразу після слова, без розділового пробілу. Це означає, що якщо за даними словом іноді ставиться кома, а іноді пробіл, то кодування першого символу не зміниться.

Вставка одного пробілу перед знаком пунктуації усуває це протиріччя. Звичайно, сам знак все ще повинен бути закодований. але його контекст тепер більш здійсненний, і в цілому втрати від розширення тексту більш ніж компенсуються. Ця ідея вже використовувалася в декількох компресорах, наприклад, *PPMN* і *Durilca* [13].

Компресори з сімейства *LZ77* значно відрізняються від *PPM* і *BWCA*. В основному, вони розбирають вхідні дані на потік співпадаючих послідовностей і окремих символів. Збіги *LZ* кодуються шляхом зазначення їх оцінок, тобто відстані до попереднього появи послідовності в обмеженому минулому, і довжини, в той час як літери кодуються безпосередньо, в більшості варіантів з кодуванням Хафманна. Компресори *LZ77* не прогнозують символи на їх контекстної основі, вони роблять це тільки в дуже низькому порядку рудиментним чином. Сильна сторона *LZ77* полягає в лаконічному кодуванні довгих співпадаючих слів, але на практиці більшість збігів досить короткі, і знову ж проста схема кодування не може конкурувати з сучасними компресорами, хоча і забезпечує високу швидкість декодування. Отже, текстовий препроцесор, адаптований для компресорів *LZ77*, повинен намагатися:

- зменшити кількість окремих символів для кодування;
- зменшити обсяг (в байтах) послідовностей збігів;
- зменшити довжину (в байтах) послідовності збігів;
- практично збільшити інтервал в якому збігаються послідовності.

послідовності.

Ці міркування приводять до висновку, що майже будь-яка ідея щодо скорочення виведення препроцесора також корисна для кінцевого обчислення. Це не відноситься до *PPM* або *BWCA*.

Найбільш серйозним недоліком попередньої обробки тексту на основі словників (включаючи *WRT*) є необхідність зберігання досить великого словника. Це позначається як на обсязі пам’яті, так і на розмірі пакета попередньої обробки. Тому дуже важливо, щоб словник був по можливості

компактним, але в той же час придатним для швидкості обробки. Існує безліч структур даних, які можуть бути використані для зберігання словника, але вимоги до швидкості і обсягу пам'яті виключають деякі з них.

Компактне представлення словника насправді означає дві пов’язані, але різні речі. Перша – використовувати якомога менше пам’яті під час обробки (і, якщо можливо, без шкоди для швидкості), друга – зберігати словник на диску в максмально компактному архіві. Остання мета дуже важлива, якщо багато словників (наприклад, для різних мов) поширюються разом з програмою архівації. Тривіальний спосіб мінімізувати як використання оперативної пам'яті, так і дискового сховища – зменшити кількість записів в словнику. І знову ж таки, самий банальний спосіб обрізки словника – відкинути самі рідкісні слова, тобто урізати словник після m перших записів, $m < n$, де n – кількість слів у словнику. Очевидно, що якщо m набагато менше n, то втрати при стисканні будуть помітні.

Отже, статичний препроцесор на основі словника, названий трансформацією заміни слів (*WRT*), призначений для стиснення тексту. WRT значно відрізняється від своїх конкурентів як в продуктивності подальшого стискання, так і в швидкості. Існує два варіанти перетворення: один підходить для *PPM* і *BWCA* компресорів, і один, адаптований для компресорів *LZ77*. Вихідні дані *WRT-LZ77* є більш компактним і іноді можна порівняти за розміром з результатом *gzip*, виконуваного на необробленому тексті, що означає, що орієнтована на *LZ77* версія *WRT* може сприйматися як окремий і дуже швидкий спеціалізований компресор.

Для цього методу подання словника можливе як в основній пам’яті так і на диску. При цьому словник, представлений у вигляді мінімального автомата кінцевих станів, перевершує рішення на основі ланцюгового хеша і потрійного дерева пошуку як по швидкості, так і (в кілька разів) за займаною пам'яті. Що стосується подання на диску, зручного для поширення програмного забезпечення, то супровід кожного слова кількістю пропусків і запуск *PPMonstr*

(добре пристосованого до нестаціонарних даних) з високим порядком є найкращим рішенням, що перевершує варіанти відомого стиснення фронту.

Кілька ідей щодо підвищення ступеня стискання тексту вимагають подальшого вивчення, а саме:

- слова в словнику можуть бути позначені тегами частин мови і / або категорій, і експерименти Смирнова [14] показали. що така додаткова інформація в словнику позитивно впливає на стискання. Недоліком цього підходу є те, що підготовка словників для різних мов стає дуже трудомісткою.
- можливо, вдасться досягти дещо більшої міри стиснення, розширивши словник кількома фразами, які часто зустрічаються в даній мові.
- було б цікаво зібрати словники для різних мов і простежити вплив представлених ідей, застосованих окремо.

В цілому, для багатомовних програм стискання тексту коротке компактне представлення словників має вирішальне значення і залишається на сьогодні справжньою проблемою.

РОЗДІЛ 3. ЕКСПЕРИМЕНТАЛЬНА ПЕРЕВІРКА ЕФЕКТИВНОСТІ

3.1. Особливості реалізації кодувань

Реалізація алгоритму кодування реверсивними мультироздільниковими кодами базується на [12].

Для того, щоб було зручно тестувати стискання реверсивними мультироздільниковими кодами після WRT останній потрібно реалізувати таким чином, щоб окремі його етапи можна було легко включити/відключити. Для цього було введено окрему глобальну змінну preprocFlag окремі біти якої відповідали за наявність певного етапу. Тоді легко можна організувати макрос для перевірки чи включений певний етап алгоритму:

$$\#define\ IF\_OPTION(option)\ ((preprocFlag\ \&\ option)! = 0)$$

Аналогічно можна написати і макроси для включення/виключення цієї опції:

$$\#define\ TURN\_OFF(option)\ \{if\ (preprocFlag\ \&\ option)\ preprocFlag -= option;\}$$

$$\#define\ TURN\_ON(option)\ \{if\ ((preprocFlag\ \&\ option) == 0)\ preprocFlag += option;\}$$

Значення різних опцій було теж реалізовано за допомогою дефайнів:

$\#define\ OPTION\_SPACE\_AFTER\_CC\_FLAG$ 2

$\#define\ OPTION\_CAPTIAL\_CONVERSION$ 4

$\#define\ OPTION\_TRY\_SHORTER\_WORD$ 8

$\#define\ OPTION\_USE\_NGRAMS$ 16

$\#define\ OPTION_WORD_SURROROUNDING_MODELING$32

$\#define\ OPTION\_SPACE\_AFTER\_EOL$ 64

$\#define\ OPTION\_EOL\_CODING$ 128

$\#define\ OPTION\_NORMAL\_TEXT\_FILTER$ 256

$\#define\ OPTION\_USE\_DICTIONARY$ 512

3.2. Реалізація тестування ефективності

Для того, щоб зручно було проводити тестування одного і того ж тексту різними видами реверсивних мультироздільникових кодів, було вирішено використати фреймворк *Google Test*.

Крім того, оскільки усі алгоритми кодування працюють по схожому принципу(створення словника зі слів тексту, генерація відповідної кількості кодових слів та власне кодування) доречно створити уніфікований метод ініціалізації об’єктів цього класу та роботи з ними задля того щоб уникнути непотрібного нагромадження коду та слідувати принципам ООП.

Спочатку потрібно створити інтерфейс для компресора – класу, що буде відповідати за стискання кодових слів. Потім окремі види компресорів для реверсивних мультироздільникових кодів будуть наслідувати цей інтерфейс. На рисунку 3.1 представлений вигляд цього інтерфейсу.

```
class Compressor {
 protected:
  virtual ResizableWritableBitStream EncodeToStream(const std::vector<int64> &values) = 0;

 public:

  virtual ~Compressor() = default;

  virtual std::string GetName() = 0;

  virtual std::string EncodeValue(int64 value) {
    return Stringify() << EncodeToStream({value});
  }

  virtual std::vector<uint8> Encode(const std::vector<int64> &values) = 0;

  virtual std::vector<CodeWord> EncodeToCodeWords(const std::vector<int64> &values) = 0;

  virtual std::vector<Block> EncodeToBlocks(const std::vector<int64> &values) = 0;

  virtual std::vector<int64> Decode(const std::vector<uint8> &data) = 0;
};
```

Рисунок 3.1 – Інтерфейс компресору

Основними методами, які нам потрібні є *Decode* та *Encode* – вони відповідають за декодування та кодування текстів відповідно. При цьому в якості аргументу перший із них приймає окремі байти, а другий – порядкові номери слів тексту у словнику з усіх наявних слів, відсортованому за

зменшенням частоти появи у тексті. Значення вони повертають навпаки – після кодування повертаються байти, після декодування – кодові порядкові номери слів із словника.

Після цього варто задуматись як саме будуть створюватись нові об’єкти класів-компресорів. Доречним є застосування фабрики (рис. 3.2).

```
static std::unique_ptr<CodewordBasedCompressor> R235(int64 code_word_num) {
  return CreateCompressor(code_word_num, true, CodeType::R_2_3_5);
}

static std::unique_ptr<CodewordBasedCompressor> R24Inf(int64 code_word_num) {
  return CreateCompressor(code_word_num, true, CodeType::R_2_4_INF);
}

static std::unique_ptr<CodewordBasedCompressor> R2Inf(int64 code_word_num) {
  return CreateCompressor(code_word_num, true, CodeType::R_2_INF);
}

static std::unique_ptr<CodewordBasedCompressor> R2(int64 code_word_num, bool monotone) {
  return CreateCompressor(code_word_num, monotone, CodeType::R_2);
}
```

Рисунок 3.2 – Методи фабрики

Таким чином, один раз ініціалізувавши фабрику ми легко зможемо отримати доступ до різних компресорів.

3.3. Результати проведених тестів

Було протестовано використання *WRT* та реверсивних мультироздільникових кодів у якості передобробки для архіваторів *bzip2*, *7zip*, *zstd* та *gzip*. Для тестування було використано два англомовних тести різних розмірів, результати тестувань наведені у таблицях 3.1 та 3.2, розміри файлів після стискання наведені в байтах.

Також було перевірено гіпотезу, що *WRT* покращить подальшу обробку реверсивними мультироздільниковими кодами (табл. 3.3).

Таблиця 3.1 – Результати на першому тексті (100 Мб)

Архіватор	Алгоритм				
	WRT	$R_{2,3,5}$	$R_{2,4,inf}$	$R_{2,3,5}+WRT$	$R_{2,4,inf}+WRT$
zstd	27013054	26632750	26542084	26707385	26540581
7zip	26907293	26525674	26416726	26575442	26474216
bzip2	29105692	27068378	27041980	27137157	27109259
gzip	34691381	27360164	27217726	27421123	27284298

Таблиця 3.2 – Результати на другому тексті (1 Гб)

Архіватор	Алгоритм				
	WRT	$R_{2,3,5}$	$R_{2,4,inf}$	$R_{2,3,5}+WRT$	$R_{2,4,inf}+WRT$
zstd	246950124	246772108	244996413	247881520	245872210
7zip	247488538	245617877	244719289	245152894	244232600
bzip2	254602221	252729702	252285798	255099222	255170698
gzip	295543634	256081997	254861323	262053467	261120812

Таблиця 3.3 – Результати тесту мультироздільникових кодів з *EOL*

Архіватор	Алгоритм			
	Перший текст		Другий текст	
	$EOL+R_{2,3,5}$	$EOL+R_{2,4,inf}$	$EOL+R_{2,3,5}$	$EOL+R_{2,4,inf}$
zstd	26632150	26541023	247911253	245977124
7zip	26525055	26423653	245623189	244712936
bzip2	27068673	27041185	252730431	252290841
gzip	27360229	27218290	256083755	254862744

Як видно з таблиці 3.3, лише деякі етапи *WRT* доцільно використовувати для покращення реверсивних мультироздільникових кодів. При цьому навіть вони покращують ефективність лише на текстах середньої довжини. Тим не менше, отримані результати дають можливі напрями для подальших їх оптимізацій.

ВИСНОВКИ

У роботі було реалізовано алгоритми кодування за допомогою реверсивних мультироздільникових кодів та алгоритм *WRT*, перевірено ефективність їх використання у якості передобробки для архіваторів *bzip2*, *7zip*, *zstd* та *gzip*. Також було протестовано ефективність застосування їх разом, включаючи застосування лише певних етапів з *WRT*.

Експерименти були проведені з двома текстами англійською мовою, розміром за допомогою власної реалізації алгоритмів $R_{2,3,5}$, $R_{2,4,inf}$ та *WRT*. Архіватори були взяті з відкритих джерел.

В результаті експеременті було показано що реверсивні мультироздільникові коди $R_{2,3,5}$ та $R_{2,4,inf}$ краще показують себе при архівації великих тестів за допомогою архіваторів *bzip2*, *7zip*, *zstd* та *gzip* ніж алгоритм *WRT*, також було показано ефективність застосування деяких перетворень з *WRT* як передобробку до алгоритму кодування реверсивними мультироздільниковими кодами.

Загалом було встановлено, що після обробки *WRT* кількість кодових слів стає меншою, що мало б покращити роботу реверсивних мультироздільнкових кодів, але на практиці результати були гірші через те, що перетворення *WRT* збільшує ентропію файлу.

Отримані результати можуть слугувати аргументом для застосування реверсивних мультироздільникових кодів як передобробки для потужних архіваторів, оскільки було експериментально доведено ефективність такого підходу. Також результати використання окремих етапів *WRT* разом з реверсивними мультироздільниковими кодами можна використати для покращення останніх.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Алгоритмы: построение и анализ, 2-е издание / [Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К.]. – М. : Вильямс, 2005. – 1296 с.

2. Сэломон Д. Сжатие данных, изображения и звука / Д. Сэломон. – М. : Техносфера, 2004. – 368 с.

3. Левитин А. В. Жадные методы: Алгоритм Хаффмана // Алгоритмы: введение в разработку и анализ / А. В. Левитин. – М.: Вильямс, 2006. – 424 с.

4. Лидовский В. В. Теория информации: Учебное пособие / В.В.Лидовский – М. : Компания Спутник+, 2004. – 111 с.

5. Кенцл Т. Форматы файлов Internet Перев. с англ. / Кенцл Т. – СПб : Питер, 1997. – 320 с.

6. Методы сжатия данных / [Ватолин Д., Ратушняк А., Смирнов М., Юкин В.]. – М. : ДИАЛОГ-МИФИ, 2013 – 381 с.

7. Реализации алгоритмов/ Кодирование длин серий [Електронний ресурс] – Режим доступу: https://ru.wikibooks.org/wiki/Реализации_алгоритмов/Кодирование_длин_серий.

8. Воробьев Н. Н. Числа Фибоначчи [Популярные лекции по математике] / Н. Н. Воробьев. – М. : 1978 – 144 с.

9. Завадський І. О. Мультироздільникові коди / Завадський І. О. // Наукові записки НаУКМА. – 2015. – Т. 177 : Комп'ютерні науки. – С. 69-76.

10. Variable-length Prefix Codes with Multiple Delimiters / Anatoly V. Anisimov, Member, IEEE and Igor O. Zavadskyi. // IEEE Transactions on Information Theory. – 2017. – Volume: 63, Issue: 5. – P. 2885–2895

11. Застосування мультироздільникових кодів до архівування природномовних текстів / Анісімов А.В., Завадський І.О., Чудаков Т.С. // Cyb. and comp. eng. – 2020. – № 4 (202). – С. 5–24

12. Reverse Multi-Delimiter Compression Codes / I Zavadskyi, AV Anisimov // 2020 Data Compression Conference (DCC). – 2020. – P. 173–182

13. Revisiting dictionary-based compression / Przemysław Skibiński Szymon Grabowski // Software: Practice and Experience. – 2005. – Volume 35, Issue 15. – P. 1455-1476

14. Smirnov M.A. Techniques to enhance compression of texts on natural languages for lossless data compression methods // Proceedings of V session of post-graduate students and young scientists of St. Petersburg, State University of Aerospace Instrumentation. – Saint-Petersburg, Russia. – 2002. (In Russian) (http://www.compression.ru/download/articles/text/smirnov 2002 pos tagging/smirnov 2002 pos tagging.pdf.)