Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

**Алгоритм Brotli від Google для стискання веб-даних**

**Текстова частина до курсової роботи**

**за спеціальністю „Програмна інженерія” 6.050103**

#### Керівник курсової роботи

Кандидат фіз.-мат. наук

доц. *Глибовець А.М.*

*(прізвище та ініціали)*

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*(підпис)*

“\_\_\_\_” \_\_\_\_\_\_\_\_\_\_ 2016 р.

Виконав студент ФІН-2

*Ткач Н.В.*

*(прізвище та ініціали)*

“\_\_\_\_” \_\_\_\_\_\_\_\_\_\_ 2016 р.

Київ 2016

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ

Зав.кафедри інформатики

док.фіз.-мат.наук,проф. \_\_\_\_\_\_\_\_\_\_\_\_М.М.Глибовець (підпис)

„\_\_\_\_”\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_2016 р.

**ІНДИВІДУАЛЬНЕ ЗАВДАННЯ**

на курсову роботу

студенту  ***Ткачу Назарію Васильовичу*** факультету***\_інформатики 2\_*** курсу

ТЕМА Алгоритм Brotli від Googlе для стискання веб-даних

Вихідні дані:

* Розбір особливостей алгоритму
* Порівняння з іншими алгоритмами
* Тестування та аналіз роботи алгоритму на різних даних

Зміст ТЧ до курсової роботи:

Індивідуальне завдання

Анотація

Вступ

1. Теоретичні відомості.

2.Особливості роботи алгоритму та його реалізації

3. Практичне застосування та порівняльний аналіз

Висновки

Список посилань

Додаток А

Додаток Б

Додаток В

Додаток Г

Додаток Ґ

Додаток Д

Додаток Е

Дата видачі „\_\_\_” \_\_\_\_\_\_\_\_\_ 2016 р. Керівник \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(підпис)

Завдання отримав \_\_\_\_\_\_\_\_\_\_

(підпис)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **ЗМІСТ**  **Календарний план виконання роботи**  **Анотація**  ………………………………………………………………………............1 **ВСТУП ………………………………………………………………………………….2** **РОЗДІЛ 1: ТЕОРЕТИЧНІ ВІДОМОСТІ……………………………....3**  1.1. Загальні положення теорії інформації……………………………………………3  1.2. Аналогічні алгоритми ……………………………………………………………..4  1.3 Мета створення ………………………………...……………………………….......5  1.4. Представлення стиснених даних………………………...………………………..6  1.5.Префіксні коди в алгоритмі ……………………………….……………………..10  **РОЗДІЛ 2: ОСОБЛИВОСТІ РОБОТИ АЛГОРИТМУ ТА ЙОГО РЕАЛІЗАЦІЇ…………………………………………………………......15**  2.1. Кодування відстаней та команд блоку…………………..……………………...15  2.2. Моделювання контексту ………………………………..…………………….....19  2.3. Статичний словник………………………………………..……………………...20  2.4. Формат потоку та мета-блоків……………….………….……………………….23  **РОЗДІЛ 3: ПРАКТИЧНЕ ЗАСТОСУВАННЯ ТА ПОРІВНЯЛЬНИЙ АНАЛІЗ…………………………………………………………23**  3.1 Методи та критерії порівняльного аналізу………………………………………23  3.2 Створення автономних частин стиснених даних……………………………….24  3.3 Безпека…………………………..……………………………...……………….....25  3.4 Тестування роботи алгоритму на корпусах даних та веб-даних…………….....26  3.5 Результати та висновки …………………………..……………………………....30  **ВИСНОВКИ…………………………………………………………….33**  **СПИСОК ПОСИЛАНЬ** …………………………………………………………..34  **Додаток А**…………………………………………………………………………....35  **Додаток Б**……………………………………………………………………………37  **Додаток В**……………………………………………………………………………38  **Додаток Г**……………………………………………………………………………39  **Додаток Ґ**…………………………………………………………………………….39  **Додаток Д**……………………………………………………………………………39  **Додаток Е**…………………………………………………………………………….40  Перелік прийнятих скорочень  СR – (від англ. compression ratio )- відношення стискання.  СS – (від англ. сompression speed) – швидкість стискання.  DS – (від англ. decompression speed) – швидкість відновлення. | **Стор.** | | 4 | | 5 | | 6 | | 8 | | 8 | | 12 | | 14 | | 15 | | 16 | | 22 | | 22 | | 23 | | 24 | | 26 | | 30 | | 30 | | 31 | | 32 | | 33 | | 34 | | 37 | | 38 | |  | 40 | |  | 42 | |  |  | |  |  | |  |  | |

**Тема: *Алгоритм Brotli від Google для стискання веб-даних***

**Календарний план виконання роботи:**

|  |  |  |  |
| --- | --- | --- | --- |
| №  п/п | Назва етапу курсової роботи | Термін виконання етапу | Примітка |
| 1. | Отримання завдання на курсову роботу. | 13.10.2015 |  |
| 2. | Огляд технічної літератури за темою роботи. | 02.11.2015 |  |
| 3. | Виконати аналіз існуючих досліджень за темою | 23.11.2015 |  |
| 4. | Розібрати принцип роботи алгоритму | 09.12.2015 |  |
| 5. | Дослідити особливості реалізації | 21.12.2015 |  |
| 6. | Порівняти з аналогічними алгоритмами | 19.01.2016 |  |
| 7. | Провести тестування з алгоритму на різних корпусах документів за різних параметрів | 24.02.2016 |  |
| 8. | З’ясувати ефективні методи роботи з алгоритмом | 23.03.2016 |  |
| 9. | Створення слайдів для доповіді та написання доповіді. | 20.04.2016 |  |
| 10. | Аналіз отриманих результатів з керівником, написання доповіді. | 27.04.2016 |  |
| 11. | Корегування роботи. | 3.05.2016 |  |
| 12. | Остаточне оформлення пояснювальної роботи та слайдів. | 6.05.2016 |  |
| 13. | Захист курсової роботи | 12.05.2016 |  |

Студент **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

Керівник **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**“\_\_\_\_\_\_”\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

Анотація

У даній курсовій роботі розглянуто принцип роботи та елементи реалізації новітнього алгоритму для стискання веб-даних Brotli, розробленого програмними інженерами компанії Google Джуркі Алакуджалою та Євгенієм Ключіковим. Порівняно результати роботи алгоритму з іншими популярними алгоритмами. Досліджено, яким чином зміни до налаштувань алгоритму впливають на роботу алгоритму з даними які належать до різних популярних веб-форматів .

ВСТУП

У зв’язку з швидким збільшенням кількості веб-документів в мережі та якісних змін в їхньому наповненні, з’явилася потреба в більш ефективних алгоритмах стискання, які моли б за менший час стиснути веб-дані компактніше, аніж їхні аналоги до сьогодні. Це - пріоритетна задача для компаній які створюють веб-браузери, адже можливість набагато швидше постачати та опрацьовувати контент призводить до збільшення рівня конкурентоспроможності браузера, та відповідно до збільшення частки користувачів які на регулярній основі відвідують та використовують його.

З погляду звичайних розробників, кращі алгоритми стискання можуть бути використані в першу чергу при роботі з хмарними сервісами, коли ресурси для виконання програми та для збереження даних тарифікує компанія яка надає цей сервіс.

Алгоритм Brotli від Google – це новітній, в плані роботи з веб-даними за критеріями швидкості та якості зберігання даних, алгоритм. За багатьма показниками він значно перевершує аналогічні популярні алгоритми стискання .

Варто зазначити на майже повну відсутність великих досліджень цього алгоритму та варіантів його імплементації. Це, в свою чергу, підвищує актуальність та необхідність детального аналізу принципу роботи та методів які розробник зміг би використати цей алгоритм на практиці. З його поширенням та використанням у різних версіях найбільш популярних браузерів, все більша кількість розробників будуть змушені звернутися до цього алгоритму в своїй роботі .

Для його ефективного використання необхідне глибоке розуміння особливостей цього алгоритму та залежностей між тим, як налаштовано та реалізовано цей алгоритм та які результати на конкретних типах даних, що нас цікавлять, ми отримаємо в підсумку.

1.ТЕОРЕТИЧНІ ВІДОМОСТІ

**1.1Загальні положення теорії інформації**

Стиснення даних – процес кодування інформації за використання меншої кількості бітів , аніж в оригінальному представленні. Розрізняють стиснення без втрат та стиснення з втратами. В алгоритмах стиснення без втрат при кодуванні не втрачають жодної інформації. На противагу ним, в алгоритмах стиснення з втратами визначають надлишкову інформацію і видаляють її. В цій роботі розглянуто принцип та реалізацію роботи алгоритму стиснення без втрат та його порівняння з іншими алгоритмами стиснення без втрат.

Більшість цих алгоритмів використовують статистичні моделі для вхідних даних аби отримати частотний розподіл символів та фраз у вхідному потоці. З цього будують модель яка встановлює однозначний бієктивний зв’язок(відображення) між символом та послідовністю бітів. Для аналізу якості стискання використовують поняття інформаційної ентропії – очікуваного середнього значення інформації в повідомленні. Завдяки ній встановлюють найбільше можливе значення відношення нестиснених даних до стиснених. Це відношення ще називають відношенням(коефіцієнтом) стиснення. Цей коефіцієнт використовують як один з основних критеріїв оцінки якості роботи того чи іншого алгоритму стиснення.

Алгоритми стиснення без втрат поділяють на загальні – ті, що можуть обробити будь-які типи даних ,незалежно від їхнього формату,та спеціальні, які здатні працювати лише з аудіо,відео,текстом,інформацією з ДНК, тощо.

Для оцінки якості стискання алгоритму використовують стандартні множини(корпуси) документів. Завдяки ним можна порівняти виконання різних алгоритмів за різних налаштувань параметрів на однакових стандартних документах.

. **1.2 Аналогічні алгоритми**

Стиснення даних є одним з найбільш важливих інструментів для прискорення роботи веб-сайтів. Стиснуті дані набагато легше та менш затратно передавати по каналам зв’язку, що зменшує час який необхідний для завантаження сторінки на стороні клієнта. Найбільш популярним на сьогодні форматом стиснення даних для веб-даних є gzip. Більшість з популярних алгоритмів стиснення даних без втрат використовують LZ77 та кодування Хаффмана.

LZ77[11] – алгоритм створений Абрахамом Лемпелем та Якобом Зівом в 1977 році. Цей алгоритм розглядає дані як єдину стрічку в якій підпослідовності байт закодовують посиланнями на позицію першої появи цієї стрічки. Як правило,указник має форму <довжина,відстань>, де довжина означає кількість знайдених ідентичних байтів, а відстань означає скільки байт розділяють нинішню стрічку та першу.

Кодування Хаффмана[7] – алгоритм стиснення даних без втрат розроблений Давидом Хаффманом в 50х роках. Його використовують для стискання різноманітних форматів,зокрема JPEG. В принципі дії алгоритму лежить використання певного типу префіксного кодування, де за заданим наперед алфавітом та текстом,символи розділяють залежно від їхніх частот появи у тексті. Потім їм надаються послідовності певної довжини в оберненій залежності від частоти їхньої появи. Це дозволяє максимально збільшити ентропію тексту, а отже, максимально мінімізувати витрати на передачу та зберігання інформації.

Загалом, це кодування можна представити у вигляді бінарного дерева, де листками є літерали алфавіту , а на кожному ребрі графу позначено 1 або 0. Для відновлення тексту декомпресор зчитує дерево з кореня до того моменту поки не зчитає літерал.

Для найпопулярнішого алгоритму gzip[4] використовують безпосередньо алгоритм стискання Deflate[3]. Він є комбінацією LZ77 та алгоритму Хаффмана. З першого алгоритму використовують пари довжина – відстань, де довжина має максимум 258 байт , а відстань - 32768 байт. Ця відстань також визначає розміри розсувного вікна для стиснення та відновлення. Для представлення літералів та відстаней використовують два окремі алфавіти, які кодують алгоритмом Хаффмана.

**1.3 Мета створення**

Створення даного алгоритму було продиктовано необхідністю покращити роботу веб-браузерів з обробки веб-контенту. Попередні алгоритми на кшталт Deflate, gzip, та Zopfli не відповідали вимозі стискання вхідного потоку даних великого обсягу. Зокрема,інженерами Google були виставлені наступні вимоги до проекту алгоритму:

* бути незалежним від виду процесору ,операційної системи,файлової системи,а також від типу кодування символів.
* можна застосувати до обміну даних
* підтримка сучасними веб-браузерами
* його можна застосувати до довільно великого обсягу даних, які послідовно записані у вхідному потоці, за використання наперед зазначених ресурсів фіксованого обсягу.
* стискує дані з відношенням стискання яке порівнюване з аналогічними відношеннями інших алгоритмів стиснення. Зокрема, має бути кращим за gzip.

Після реалізації алгоритму було виявлено, що процес відновлення даних Brotli швидший за сучасну імплементацію LZMA алгоритму. Проте, даний алгоритм не призначений для довільного доступу до будь-якої частини стиснених даних. Також, він не здатний стискати спеціалізовані формати даних, як-от растрову графіку, тощо. Нижче зазначено зручний для пояснення метод для представлення послідовності байтів як послідовностей бітів, та метод для пакування послідовностей бітів в байти назад. Далі під байтом буде визначено послідовність з 8 бітів збережених як одиниці інформації. Під стрічкою мається на увазі будь-яка послідовність байтів довільної довжини. У визначенні байту послуговуватимемося позначенням спочатку найбільш важливого біту зліва. Оскільки, число може бути більшим за 1 байт ,будемо розміщувати 1 байт числа зліва(найменш важливий байт),а решту – справа.

Отже,остаточна форма стиснених даних матиме наступні властивості:

* елементи даних спаковані в байти в порядку зростання бітового числа в байті, починаючи з найменш важливого біту .
* Префіксні коди розташовані з найбільш важливого біту, відповідно,при читанні стиснених даних «справа –наліво» фіксовані елементи стиснення будуть прочитані спочатку від найголовнішого до найменш важливого біту; для префіксних кодів - навпаки.

Приклад:значення 3-бітного числа - 6; значення 4-бітного – 2; префіксний код - 110; ще один префіксний код - 10; значення12-бітного числа - 3628.

байт 2 байт 1 байт 0

|11100010|11000101|10010110|

| | | | |

| | | | +------ значення 6

| | | +---------- значення 2

| | +-------------- префіксний код 110

| +---------------- префіксний код 10

+----------------------------- значення 3628

Рис 1.1

**1.4 Представлення стиснених даних**

Стисненні дані містять в собі заголовок та послідовність мета-блоків. Кожен мета-блок відновлює від 0 до 16 777 216 нестиснених байтів. Остаточні відновлені дані є конкатенацією всіх відновлених послідовностей з кожного мета-блоку.

Заголовок містить розмір розсувного вікна яке було використане під час стиснення. При відновленні це вікно має бути заповнене нестисненими даними для того, щоб відновити наступні байти. Розмір розсувного вікна є степеню двійки і може бути зміненим від 1 кілобайта до 16 мегабайт. Кожен мета-блок стиснений комбінацією з LZ77 та алгоритму кодування Хаффмана.

Результатом кодування Хаффмана є префіксні коди. Для кожного мета-блоку вони є незалежними від інших кодів з попереднього чи наступного блоку. Алгоритм LZ77, в свою чергу,використовує посилання до стрічок які повторилися в попередніх мета-блоках. Також,посилання на стрічку може звертатися до статичного словника вбудованого в реалізації алгоритму.

Кожен мета-блок складається з двох частин ,а саме, заголовку мета-блоку, який описує представлення частини даних яка стиснута, та цю ж частину стиснених даних. Частина стиснених даних складається з послідовності команд. В свою чергу, кожна з команд складається з двох частин: послідовності байтів , або ж стрічок(які до того не були розпізнані розсувним вікном як такі, що мають повторення), та указник на стрічку з повторенням. Її представлено у вигляді пари <довжина,відстань>. В команді може бути 0 літеральних байтів(стрічка порожня). Мінімальна довжина стрічки для копіювання дорівнює 2. Остання команда в блоці може містити лише літерали(байти) та не містить указник на стрічку з повторенням.

Кожна з команд стиснених даних представлення у вигляді 3 категорій префіксних кодів. Перша множина префіксних кодів призначена для довжин послідовностей літералів та відстаней до копій. Таким чином, один префіксний код визначає дві довжини:довжину самої послідовності та відстань до копії. Друга - тільки для літералів. Третя категорія тільки для відстаней. Префіксні коди для кожного блоку знаходяться перед стисненим даним в заголовку мате-блоку. Приклад команди як послідовності префікс- них кодів:

<Довжина вставки та копії ,літерал,літерал,….,відстань>

Довжина вставки визначає кількість літералів які йдуть одразу після неї. Відстань визначає як далеко слід звернутися назад аби повернутися до копії. Довжина копії визначає кількість байтів в копії. В результаті нестиснені дані мають вигляд :

<Літерал,літерал,…..,копія,копія>

Остання команда в мета-блоці може закінчуватися останнім літералом якщо загальна довжина нестиснених даних була повністю заповнена. В цьому випадку немає відстані в цій команді, і довжина копії не потрібна. Префіксних кодів може бути кілька для кожної категорії, і у цьому разі, префіксний код для наступного елемента цієї категорії визначений в контексті стиснених даних які передують цьому елементові. Частиною цього контексту є 3 поточні типи блоків, по-одному для кожної категорії. Для кожної категорії існує лічильник, який зберігає кількість елементів цієї категорії які слід декодувати в поточному типі блоку. Як тільки лічильник був розширений, новий тип блоку та новий лічильник дістають з потоку перед наступним елементом категорії, який буде використовувати новий тип блоку.

Тип вставки та копіювання блоку визначає який префіксний код використати для наступного елементу цього ж типу. Для літералів та відстаней, відповідні типи блоків використовують в комбінації з іншою контекстною інформацією для наступного елементу. Розглянемо наступний приклад:

(IaC0, L0, L1, L2, D0)(IaC1, D1)(IaC2, L3, L4, D2)(IaC3, L5, D3) (1.1)

Цей мета-блок має чотири команди ,записані в дужках для ясності, де кожна з 3 категорій символів з цими командам може бути інтерпретована використовуючи різні типи блоків. Ми розділяємо кожну категорію на свою власну послідовність аби показати приклад типів блоків . Кожна група з квадратним дужками використовує один тип блоку.

[IaC0, IaC1][IaC2, IaC3] (1.2)

- вставка та видалення : типи блоків 0 та 1

[L0, L1][L2, L3, L4][L5] (1.3)

- літерали : типи блоків 0 ,1 та

[D0][D1, D2, D3] (1.4)

* відстані :типи блоків 0 та 1

Наступні блоки в кожній категорії мають мати різні типи блоків, але їх можна потім використовувати заново в мета-блоці. Типи блоків нумеровані від 0 до максимального числа блоків (255) і перший блок кожної категорії має тип 0.Блочна структура мета-блоку представлена послідовністю команд для кожної категорії, де команда має вигляд:

<тип блоку,лічильник блоку>

Команди блоку представлені в стиснених даних до початку кожного нового блоку префіксним кодом для типів блоків та окремим префіксним кодом для лічильників блоків кожної категорії. Для вищеназваного прикладу прикладом мета-блоку є:

IaC0 L0 L1 LBlockSwitch(1, 3) L2 D0 IaC1 DBlockSwitch(1, 3) D1

IaCBlockSwitch(1, 2) IaC2 L3 L4 D2 IaC3 LBlockSwitch(0, 1) L5 D3

Тут команда xBlockSwitch(t, n) перемикає до типу блоку t, n елементів в блоці. Зазначимо, що в цьому прикладі DBlockSwitch(1, 3) розташована одразу попереду наступної необхідної відстані D1. Коли елемент категорії потрібен, і лічильник категорії дійшов до нуля, тоді зчитують новий тип блоку та лічильник з потоку прямо перед читанням наступного елементу. Команди блоку для перших блоків кожної категорії не є частиною мета-блоку стиснених даних. Натомість, перший тип блоку визначений як 0 ,та перший лічильник для кожної категорії закодований в заголовку мета-блоку. Всі префіксні коди для типів блоку та лічильників визначені в заголовку мета-блоку. Кожне значення категорії(вставка та копіювання , літерали, відстані) можуть бути закодовані одним з префіксних кодів в колекції префіксних кодів, які належать до тієї ж категорії що й в заголовку мета-блоку.

Кожен конкретний префіксний код залежить від типу блоку та контексту значення блоку.

У випадку літералів ,контекст - це два попередні байти нестиснених даних.У випадку відстаней , контекст це довжина копіювання з цієї ж команди. Для довжини вставки та копіювання не використовують контексту та префіксний код залежить лише від типу блоку . У випадку літералів та відстаней контекст має відображення на упорядковану множину від 0 до 63 для літералів та від 0 до 3 для відстаней.

Індекси для кожного типу блоку та контекстний ID (ідентифікатор) утворюють разом контекстне відображення яке записане в заголовок мета-блоку. Для прикладу, префіксний код який використовують для відновлення L2 залежить від типу блоку (1),і контекстний ID для літералів визначений завдяки 2 нестисненим байтам, якими було відновлено L0 та L1. Аналогічно, визначено префіксний код для відновлення D0,що належить типові блоку (0),та контекстний ID визначений завдяки довжині копіювання з IaC0.Префіксний код для відновлення IaC3 залежить лише від типу блоку (1).

Заголовок мета-блоку також містить кількість нестиснених байтів, закодованих в мета-блок та два додаткові параметри для представлення відстаней між однаковими стрічками:кількість постфіксних бітів ,та кількість прямих відстаней.

Стиснений мета-блок може бути відзначеним у заголовку як останній мета-блок , який закінчується в стисненому потоці. Мета-блок також може просто зберігати нестиснені дані прямо як байти, без кодування чи стрічок з копіями. В цьому випадку, заголовок мета-блоку містить лише інформацію стосовно кількості не стиснених байтів та позначки що мета-блок є нестисненим. Нестиснений мета-блок не може бути останнім мета-блоком.

**1.5 Префіксні коди в алгоритмі**

Префіксне кодування представляє символи з відомого алфавіту вигляді бітових послідовностей, кожен один код на один символ. Таким чином, різні символи можуть бути представлені бітовим послідовностями різної довжини , але програма-парсер може однозначно прочитати закодовану стрічку символ за символом. Префіксним кодом для символу є послідовність 0 та 1 для ребер що йдуть з кореня до лиcтка позначеного цим символом.

Приклад :

/\ Символ Код

0 1 ------ ----

/ \ A 00

/\ B B 1

0 1 C 011

/ \ D 010

A /\

0 1

/ \

D C

Рис .1.2

Парсер може відновити наступний символ зі стисненого потоку спускаючись по дереву з кореня, вибираючи на кожному кроці відповідне до наступного біту значення. За даним алфавітом з відомими частотами , алгоритм Хаффмана дозволяє будувати оптимальне префіксне кодування в якому для кожного символу з відомою частотою виділено найменшу можливу кількість бітів.

Префіксні коді, які використовують для кожного алфавіту в алгоритмі, є канонічними префіксними кодами які мають два додаткові правила. Всі коди певної бітової довжини мають лексикографічні послідовні значення , в тому ж порядку, що й символи які вони представляють. Коротші коди лексикографічно знаходяться раніше за довші коди .

Приклад на основі алфавіту ABC:

Символ Код

A 10

B 0

C 110

D 111

Рис 1.3

На рис 1.3 спочатку йде 0,який перед 10, який перед 11,і далі 110 та 111. Всі вони є лексикографічно послідовними. Завдяки цьому правилу можна визначити канонічний префіксний код для алфавіту лише знаючи бітові довжини кодів для кожного символу в алфавіті по порядку. Цього достатньо для дійсних кодів які використовують на практиці .

В нашому прикладі, код повністю визначений завдяки послідовності бітової довжини (2,2,3,3). Наступний алгоритм записує префіксні коди у вигляді цілих чисел, за умови, що вони будуть прочитані від найбільш до найменш важливого біту .

- порахувати кількість кодів для кожної довжини коду. Нехай bl\_count[N] – кількість кодів довжини N ,при N>=1.

- знайти числове значення найменшого коду для кожної довжини. Див. Додаток А.

- присвоїти числове значення для всіх кодів , використовуючи послідовні значення для всіх кодів однакових довжини з базовими значеннями визначеним на другому кроці. Див. Додаток А

Префіксні коди використовують для різних цілей в алгоритмі, і для кожної цілі використовують різний розмір алфавіту. Для літеральних кодів їх 256. Для кодів, що позначають вставку та копіювання їх 704. Для лічильників блоків їх 26. Для кодів відстаней та кодів типів блоків, префіксні коди є динамічними.

Перші два біти стисненого представлення кожного префіксного коду визначають чи це простий чи складний префіксний код. При значенні в 1 - це простий префіксний код. Простий префіксний код може мати лише не більше чотирьох символів з ненульовою довжиною коду. В загальному , формат простого префіксного коду:

-2 біти:значення в 1 означає що це простий код

-2 біти : NSYM(назва взята з реалізації) - 1

-1 біт :наявний лише коли NSYM = 4

Кількість бітів для позначення алфавіту – найменша кількість бітів яка може представляти символи в алфавіті. Префіксні коди однакової бітової довжини мають бути записані в тому ж порядку що й їхні символи в відсортованому за зростанням порядку. Ненульові довжин кодів символів можуть бути реконструйовані наступним чином:

- якщо NSYM=1 , то довжина коду для першого символу – 0;при кодуванні цього символу в стиснених даних ніяких додаткових бітів не додають.

- якщо NSYM = 2, то всі символи мають довжину 1.

- якщо NSYM = 3, то довжини кодів для символів наступні : 1, 2, 2 в порядку в якому вони з’являються в представленні простого префіксного коду.

- якщо NSYM = 4, то довжини кодів залежать від двох комбінацій довжин : 2, 2, 2, 2 або 1, 2, 3, 3.

Складним префіксним кодом називають канонічний префіксний код який визначений як послідовність довжин кодів . Для більшої зручності довжини послідовностей стискають використовуючи префіксні коди. Алфавіт для довжин кодів наступний :

* від 0 до 15:Для кодів довжини 0..15
* 16:Скопіювати попередню ненульову довжину коду 3..6 разів

Наступні 2 біти визначають довжину повторення (0 = 3, ... , 3 = 6) . Якщо це перший код довжини , або всі попередні мають нульову довжину, то код довжини 8 повторюють від 3 до 6 разів. Повторюваний код 16 змінює лічильник повторень наступним чином

repeat count = (4 \* (repeat count - 2)) +(3- 6,залежно від 2 наступних бітів)

* 17:Повторити довжину коду 0 від 3 до 10 разів (3 біти довжини ). Код 17 змінює лічильник повторень наступним чином:

repeat count = (8 \* (repeat count - 2)) +(3-10 з наступних 3 бітів)

Варто зазначити, що код 16 змінює попередній лічильник повторень , який стає новим лічильником повторень. Аналогічно і для 17. Послідовність з кількох кодів 16 або ж для 17 також можлива, якщо постійно будуть відбуватися зміни лічильника. Відповідно, лише останній лічильник буде остаточно використаний ..

Довжина коду в 0 означає що відповідний символ в алфавіті не з’явиться в стиснених даних , і він не бере участі в обчисленнях префіксного коду в алгоритмі описаного вище.

Складний префіксний код мусить мати щонайменше дві ненульові довжин коду. Бітові довжини префіксного коду стиснені використовуючи наступне кодування змінної довжини(variable length coding).Приклад Рис 1.4

Символ Код

------ ----

0 00

1 0111

2 011

3 10

4 01

5 1111

Рис 1.4

Тепер можна визначити формат складних префіксних кодів наступним чином:

-2 біти: HSKIP, його значення 0,2,3 представляють відповідне число пропущених довжин кодів. Пропущені довжини всі за замовчуванням нульові.(HSKIP -1 .означає, що наявний прости префіксний код).

Довжини кодів для символів в алфавіті заданому вище, наступні: 1, 2, 3, 4, 0, 5, 17, 6, 16, 7,8, 9, 10, 11, 12, 13, 14, 15.

За умови що, HSKIP дорівнює 2 , тоді довжини кодів для символів 1 та 2 дорівнюють 0 , та перша довжина коду встановлена для символу 3 позиції. За умови що HSKIP дорівнює 3 , тоді довжина коду для символу 3 також нульова , та перша довжина коду встановлена вже відповідно для символу 4. Кодові довжини для символів між 0 та 5 представлені від 2 до 4 бітів залежно від коду змінної довжини. Довжина коду 0 означає що відповідний кодовий символ не використаний .

Якщо HSKIP 2 або 3 , відповідне число основних(перших у розташування при зчитуванні )довжин коду є нульовими та вони не присутні в послідовності довжин коду визначеній вище.

Якщо у нас наявно принаймні дві ненульові довжини коду, то будь-яка нульова довжина коду опущена. Тобто, остання довжина коду мусить бути ненульовою. В цьому випадку , сума всіх ненульових довжин коду мусить дорівнювати 32.

Якщо довжини були прочитані для всього алфавіту довжин коду і є лише одна ненульова довжина коду , тоді префіскний код має один символ з нульовою довжиною. В цьому випадку, для цьому символу не визначають додаткових бітів для кодування компресором , та жодні біти не отримані декомпресором.

Одиничний символ одразу буде повернутий коли цей код буде розшифрований. Нехай у нас є код представлений символами довжин 8.Тобто ми маємо літеральний код який представляє значення цих літералів з однаковою ймовірністю. В цьому випадку єдиним символом який повторюють з попередньою частотою є 16. Попередню частоту взяту як 8 зчитують до того як буде прочитана довжина коду.

Будь-який перший символ (0 або 17) має бути пропущений, так як остання закодована довжина символу мусить бути між 1 до 16. Сума довжин по всім ненульовим довжинам коду в алфавіті , разом з закодованим кодом повторення 16 , мають бути рівні 32768. Якщо кількість повторень попередньої довжини або повторення нульової довжини матимуть в сумі більше число, аніж кількість всіх символів в алфавіті, то потік має бути відкинутим .

**2 ОСОБЛИВОСТІ РОБОТИ АЛГОРИТМУ ТА ЙОГО РЕАЛІЗАЦІЇ**

**2.1 Кодування відстаней та команд блоку**

Одним з компонентів стисненого мета-блоку є послідовність відстаней до прочитаних раніше ідентичних послідовностей. В цьому параграфі ми ознайомимося з деталями процесу стиснення відстаней. Кожна відстань в стиснених даних мета-блоку представлена у вигляді пари <код відстані,додаткові біти>.

Коди відстані та додаткові біти закодовані разом. Код відстані кодують використовуючи той самий префіксний код по всьому алфавіту відстаней . Додаткові біти кодують використовуючи цілі значення фіксованої ширини. Кількість додаткових бітів може бути від 0 до 24 залежно від коду відстані. Щоб перевести код відстані та відповідні додаткові біти до відстані до попереднього запису, слід отримати послідовність з попередніх відстаней та два додаткові параметри. Перший - кількість постфіксних бітів. позначених NPOSTFIX (0..3). Другий – кількість прямих кодів відстані позначених NDIRECT (0..120). Обидва параметри закодовані в заголовку мета-блоку. Також ми обчислюємо наступний параметр

POSTFIX\_MASK = (1 << NPOSTFIX) – 1 (2.1)

Перші 16 символів для позначення відстані є спеціальними символами для посилання до попередніх відстаней наступним чином. Коли трапляється символ відстані 0 , відстань не записують до кільцевого буфера останніх відстаней словами.

Подібно, відстані які представляють статичні словники слів не записують до кільцевого буфера останніх відстаней. Якщо NDIRECT>0, тоді наступні символи NDIRECT від 16 до 15+NDIRECT представляють відстані від 1 до NDIRECT. Ані спеціальні символи для відстаней , ані NDIRECT прямі символи відстаней не мають після себе додаткових бітів.

Символи відстаней 16+NDIRECT та більше,мають додаткові біти для кількість додаткових бітів для символу "dcode" визначають наступною формулою.

ndistbits = 1 + ((dcode - NDIRECT - 16) >> (NPOSTFIX + 1)) (2.2)

Максимальне число додаткових бітів - 24. Розмір алфавіту відстаней дорівнює:

(16 + NDIRECT + (48 << NPOSTFIX)). (2.3)

За символом dcode та додатковими бітами dextra, відстань визначають наступними формулами:

hcode = (dcode - NDIRECT - 16) >> NPOSTFIX (2.4)

lcode = (dcode - NDIRECT - 16) & POSTFIX\_MASK (2.5)

offset = ((2 + (hcode & 1)) << ndistbits) – 4 (2.6)

distance = ((offset + dextra) << NPOSTFIX) + lcode + NDIRECT + 1 (2.7)

Як зазначено в першому параграфі, довжини вставок та копіювання закодовані простим префіксним кодом. Кожна пара <довжина вставки, довжина копіювання> - це пара стиснених даних в мета-блоці представлена наступним триплетом:

<код вставки та копіювання ,додаткові біти вставки ,додаткові біти копіювання >

Всі коди записані разом, зокрема, перший код представляють завдяки префіксному коду по алфавіту, в той час як два наступні завдяки цілим значенням фіксованої довжини. Число додаткових бітів коливається від 0 до 24 і воно залежить від коду вставки та копіювання. Опис алфавіту кодів вставки та копіювання використовуючи алфавіти довжин кодів вставки та довжин кодів копіювання. Див . Додаток Б,В.

Спочатку відбувається пошук по таблиці за даними кодами вставки та копіювання , а потім отримують код довжини вставки та код довжини копіювання.

Код довжини коду визначають за двома бітами(рахуючи з найменш важливого) від коду довжини вставки та копіювання. Код довжини вставки визначають в діапазоні визначеному бітами на позиціях 3-5 коду довжини вставки та копіювання. За даними кодами можна отримати дійсну довжину вставки та копіювання, якщо прочитати додаткові біти дані в таблиці вище.

Для коду довжини вставки та копіювання між 0 та 127, код відстані в команді встановлений як 0, тобто використовують останню відстань. Як зазначено раніше, команди блоку є парою <тип блоку, лічильник блоку>. Їх кодують в стисненій частині мета-блоку, перед початком кожного нового блоку кожної блочної категорії. Кожен тип блоку стиснених даних представлений кодом типу блоку, закодованим префіксним кодом з алфавіту типів блоку. Символ 0 означає, що новий тип блоку однаковий до попереднього з цієї ж категорії. Символ 1 означає, що новий тип блоку дорівнює поточному блоку плюс ще один тип блоку. Символи 2-257 представляють типи блоків від 0 до 255 відповідно. Попередній та поточний типи ініціалізовані 1 та 0 наприкінці заголовку мета-блоку.

Якщо кожна блочна категорія має лише один тип блоку , то лічильник блоку першої команди пропускають, в іншому випадку його закодовують в заголовок мета-блоку.

Кількість різних типів блоку в кожній категорії блоку, означені як NBLTYPESL, NBLTYPESI, and NBLTYPESD для літералів, довжин та відстаней, закодовані відповідно в заголовок мета-блоку. Кількість також має дорівнювати номеру найбільшого типу блоку плюс один в даній категорії блоку. Іншими словами, множина типів має бути впорядкована так : [0..NBLTYPESL-1], [0..NBLTYPESI-1], та [0..NBLTYPESD-1]. З цього випливає, що розмір алфавіту коді типів блоку має дорівнювати NBLTYPESL + 2, NBLTYPESI + 2, NBLTYPESD + 2 відповідно. Для кожного лічильника блоку в стиснених даних є представлення у вигляді пари <код лічильника, додаткові біти>.

Код лічильника та додаткові біти закодовані разом. При цьому використовують алфавіт кодів лічильників блоку та цілі значення для додаткових бітів.

Кількість додаткових бітів змінюється від 0 до 24 залежно від лічильника блоку. Таблиця алфавіту кодів лічильників блоку з кількістю додаткових бітів та діапазоном лічильників блоку. Див. Додаток Г

**2.2 Моделювання контексту**

Як зазначено раніше, префіксне дерево використовують для кодування літеральних байтів та кодів відстаней залежно від типу блоку та контекстного ID.

Контекст для кодування кожного наступного літералу визначають за двома останніми байтами р1 та р2 які є в потоці, незалежно від того, чи ці байти створені завдяки посиланнями на нестиснені мета-блоки , посиланнями на статичні словники чи посиланнями на літеральні вставки. На початку потоку ці значення очевидно нульові .Є чотири види контексту за допомогою яких рахують контекстний ID :

* LSB6 - беруть значення шести найменш важливих бітів байту р1.
* MSB6 - беруть значення шести найбільш важливих бітів р1.
* UTF8 - обраховують за допомогою функції від р1 та р2,де р1 та р2 – останні байти .
* Знаковий метод - використовують функцію від р1 та р2 для стиснених послідовностей цілих знакових чисел.

Контекст для кодування кодів відстані визначений завдяки довжині копіювання яка відповідає даній відстані.

Загалом наявні два контекстних відображення. Один для літералів та інший для відстаней . Кожне значення у відображенні є цілим числом від 0 до 255,яке визначає індекс префіксного коду який використовували при кодуванні наступного літералу та відстані.

Контекстні відображення є насправді двовимірним матрицями ,закодованими як одновимірні масиви.

CMAPL[0..(64 \* NBLTYPESL - 1)] (2.8)

CMAPD[0..(4 \* NBLTYPESD - 1)] (2.9)

Значення контекстного відображення кодують комбінацією кодування довжин серій для нульових значень та префіксному кодуванню. Нехай RLEMAX –число кодів довжин серій та NTREES – максимальне значення в контекстному відображення плюс 1. Звідси NTREES дорівнює кількості попарно різних значень в контекстному відображенні . Алфавіт префіксного коду має RLEMAX + NTREES символів

Якщо RLEMAX=0 ,то кодування довжин серій не використовують , та символи алфавіту є значеннями контекстного відображення .

**2.3 Статичний словник**

Під час декодування стиснених даних , посилання на повторену стрічку в нестиснених даних має максимальне значення відстані , яке є мінімальним розміром розсувного вікна та числа нестиснених даних які вже декодовані. Проте, декодування відстані з потоку стиснених даних може створити відстані які є більшими за максимальне допустиме значення. Статичний словник складається з 3 частин :

- DICT[0..DICTSIZE], масив байтів

- DOFFSET[0..24], масив значень відстаней для кожної довжини

- NDBITS[0..24], масив бітових значень для кожної довжини.

DOFFSET and DICTSIZE визначають наступним чином :

DOFFSET[0] = 0

DOFFSET[length + 1] = DOFFSET[length] + length \* NWORDS[length] (2.10)

DICTSIZE = DOFFSET[24] + 24 \* NWORDS[24] (2.11)

Кожне слово статичного словника має 121 різну форму. Слово зі статичного словника для пари<довжина,відстань> знаходять наступним чином:

word\_id = distance - (MAXDIST+ 1) (2.12)

index = word\_id % NWORDS[length] (2.13)

base\_word = DICT[offset(length, index)..offset(length, index+1)-1] (2.14)

transform\_id = word\_id >> NDBITS[length] (2.15)

Стрічку отриману з нестисненого потоку обчислюють завдяки перетворенню до базового словникового слова. Якщо transform\_id більше за 120, то потік визначають як некоректний.

Максимальне число додаткових байтів які можна додати до слова під час Отже ,38 байтів достатньо для зберігання будь-яких перетворених слів.

**2.4 Формат потоку та мета-блоків**

В цьому параграфі ми опишемо формат стиснених даних через окремі елементи даних. По-перше, заголовок потоку має лише одне поле WBITS розміром від 1 до 7 бітів. Діапазон значень поля – від 10 до 24.Вони закодовані в код змінної довжини. Див. Додаток Ґ

Розмір розсувного вікна є максимальним значенням будь-якого посилання назад не на словник,визначають наступним чином :

window size = (1 << WBITS) – 16 (2.16)

Стиснений потік має мати хоча б один мета-блок. Кожен мета-блок містить заголовку з інформацією стосовно нестисненої довжини мета-блоку , та сигнальний біт який вказує на кінець мета блоку. Формат заголовку мета-блоку наступний:

* 1 біт: ISLAST, встановлений як 1 якщо це останній блок
* 1 біт: ISLASTEMPTY, встановлений 1 якщо блок порожній ;

Останнє поле існує лише якщо біт ISLAST встановлений як 1.Тоді мета-блок та потік Brotli закінчується на цьому біті,і всі решта бітів цього байту мають нульові значення .

* 2 біти : MNIBBLES, який визначає кількість напівбайт для представлення нестисненої довжини. Якщо MNIBBLES встановлений як 0,то мета-блок порожній. В цьому випадку решта мета-блоку має наступний формат:
* 1 біт: зарезервований під 0.
* 2 біти: : MSKIPBYTES - кількість байт для представлення метаданих.
* MSKIPBYTES x8 біт: MSKIPLEN – кількість байт метаданих. Це поле присутнє лише коли значення MSKIPBYTES більше від нуля.
* MNIBBLES x4 біти : MLEN – довжина мета-блоку нестиснених даних. Якщо MNIBBLES більше за 4 і останній напівбайт нульовий , тоді потік відкидають як некоректний.
* 1 біт:ISUNCOMPRESSED. Якщо його значення 1 ,тоді всі біти до наступного байту відкидають. Це поле присутнє лише коли біт ISLAST не виставлений .
* 1-11 біт : NBLTYPESL - кількість типів блоків ,закодованих наступним кодом змінної довжини. Див. Додаток Д
* 1-11 біт:NBLTYPESL – кількість типів блоку для вставки та копіювання , де використано той же код змінної довжини.
* 1-11 біт:NBLTYPESD – кількість блоків для відстаней ,для яких використано наведений вище код змінної довжини.

Префіксний код для типів блоку для відстаней існує лише коли NBLTYPESD >=2.

* 2 біти :NPOSTFIX, параметр який використовують в кодуванні відстаней.
* 4 біти :4 найбільш важливих біти NDIRECT, які використовуються для отримання значення NDIRECT перед тим зсунувши на NPOSTFIX бітів.
* 2 біти :NBLTYPESL - вид контексту для типу блоку для літералів.
* NTREEESL(1-11біт) - кількість префіксних дерев для літералів , закодованих тим же кодом змінної довжини як і NBLTYPESL.

NTREESD(1-11 біт) - кількість префіксних дерев для відстаней , закодованих тим же кодом змінної довжини що й NBLTYPESD.

Стиснена частина даних мета-блоку складається з послідовності команд. Кожна з команд має наступний формат. Код типу блоку для наступного типу блоку для вставки та копіювання з’являється лише коли NBLTYPESL>=2 та коли попередній лічильник блоку встановлений як 0.Код лічильника блоку з додатковими бітами для наступного лічильника блоку для вставки та копіювання. Він присутній лише коли NBLTYPESL>=2 та коли попередній лічильник блоку встановлений як 0.

Довжина вставки та копіювання , як зазначено вище,закодована за використання префіксних кодів та індексом типу блоку для копіювання та вставки. Число довжини для вставки літералів представлено аналогічно до попереднього прикладу.

Наступний байт нестиснених даних кодують з літеральним префіксним кодом, де індекс визначений завдяки попереднім двом байтам, та поточному типу літерального блоку і контекстному відображенню. Коди відстаней закодовані завдяки префіксному коду та поточному індексу типу блоку для відстаней .

Якщо загальна кількість нестиснених даних після частини вставки останньої команди дорівнює MLEN, тоді довжину копіювання останньої команди не беруть до уваги. В цьому випадку, довжина копіювання останньої може мати будь-яке значення .В іншому випадку, потік відкидають. Якщо потік закінчується перед завершенням останнього мета-блоку , тоді потік слід відкинути. Посилання на повторювану стрічку може посилатися на стрічку в попередньому мета-блоці .

Проте, відстань копіювання не може посилатися до місця яке визначене перед початком нестисненого потоку чи розсувного вікна .В цьому випадку , відбувається звертання до статичного словника .

Нижче подано код простої реалізації алгоритму який породжує дійсний потік для алгоритму Brotli на мові С++. Див. Додаток Е

Зауважимо що, наведений алгоритм не стискує дані. В даній примітивній версії, потік Brotli буде більшим за оригінал. Проте, завдяки даному алгоритму можна з’ясувати, що кожна послідовність з N нестиснених даних може бути представлена в потоці Brotli, який не більше ніж

N + (3 \* (N >> 16) + 5) байт. (2.17)

Як показано вище, лише мета-блоки які йдуть одразу після мета-блоку метаданих чи порожні мета-блоки гарантовано знаходяться на початку байта. В декотрих застосуваннях може бути необхідно, щоб всі мета-блоки знаходилися на початках байтів. Цього можна досягнути додаючи нульові ,а точніше,порожні мета-блоки метаданих після кожного мета-блоку зі стисненим даним .

**3.ПРАКТИЧНЕ ЗАСТОСУВАННЯ ТА ПОРІВНЯЛЬНИЙ АНАЛІЗ**

**3.1 Методи та критерії порівняльного аналізу**

Для детального розуміння роботи та особливостей використання алгоритму були проведені експерименти з порівнянням роботи не тільки різних алгоритмів на одному корпусі, але й порівняння виконання алгоритму на даних з корпусів за різних параметрів якості.

В порівнянні[1] були проведені тести на 22 бітному розсувному вікні для brotli. Алгоритми LZMA[9] та LZHAM [10] теж використовували 22 бітне вікно, та 15 бітне для Deflate та Zopfli. Більші відношення дозволяють отримувати краще відношення стискання при втраті швидкості декодування.

Тип тестових даних для порівняння – Кантерберійський корпус[5] стискання даних. Середній розмір файлу – 55 кілобайт. Було оцінено відношення стискання , швидкість стискання та відновлення. Відмінністю від інших алгоритмів є те , що Brotli використовує статичний словник. Він містить 13 504 слів та складів які мають найвищі частоти в англійській , іспанській, китайській, інді, російській та арабській. Під час масштабного експерименту згаданого в статті[1] були використанні слова які найчастіше зустрічають в веб-документах з HTML ,XML,JS,CSS файлів. Для цих слів були застосовані перетворення аби охопити більшу частину можливих стрічок в документах. Загалом наявно понад 1.6 млн послідовностей. Для оцінки використовувалося геометричне середнє значення в якому одне вимірювання ке було б надзвичайно далеко від всіх інших результатів не впливало би так сильно на загальний результат.

Також був проведений експеримент з використанням іншого відомого корпусу – корпусу Калгарі[6]. Корпус Калгарі , а саме була використана версія 1997 року. Файли з цієї колекції були обрані через «типові» показники на більшості існуючих алгоритмах стискання. Також був використаний Великий корпус(The Large Corpus ) який містить у собі файли з повним описом геному бактерії E.coli, Біблію короля Джеймса та енциклопедію ЦРУ стосовно світових фактів.

Для аналізу саме виконання алгоритму на веб-даних були відібрані середнього об’єму файли форматів HTML,CSS,JS . Зокрема, це популярна бібліотека bootstrap, та елементи Terrier - пошукового двигуну розробленого в Університеті Глазго.

**3.2 Створення автономних частини стиснених даних**

В декотрих випадках може бути необхідно створити автономну послідовність байт з потоком Brotli. Тобто таку, яка може бути відновлена незалежно від попередніх частин відновлених даних. По-перше , якщо великий файл стиснених даних пошкоджено , то можна відновити хоча б частину інформації. По-друге, можна ділити дані і передавати їх по частинам. Якщо послідовність відновлених даних не була змінена та стиснена незалежно від попередніх даних, то стиснене представлення також не буде зміненим ,а отже, передача даних буде дешевшою. По-третє, якщо послідовності нестиснених байт стиснуті незалежно , то можна паралельно стискати ці послідовності в одному файлі.

За даним послідовностями ,скажімо,U0 та U1 ми можемо створити дві послідовності С0 та С1 ,так що конкатенація С0 та С1 буде коректним потоком Brotli . До того ж, С0 та С1 можуть буди відновлені незалежно один від одного.

Для цього слід внести зміни до останнього мета-блоку С0 ,так що ISLAST останнього блоку не має значення. Тоді слід забезпечити щоб останній блок С0 точно закінчувався на границі байта. По-перше, відстані до попередніх копій в С1 не мають посилатися на слова зі статичного словника чи нетиснутих байтів U0. Навіть якщо послідовність байт в U1 співпадає зі словом в статичному словнику , чи з послідовністю байт в U0,то компресор мусить представити послідовність байт як комбінацію з вставок літералів та посилань на копії в байтах U1.Обидва компресори для С0 та С1 мають мати однакове розсувне вікно , але заголовок потоку створений лише першим компресором який створює С0.

В даній роботі не наведено приклади виконання наведеного вище алгоритму , але сама можливість цього дозволяє застосувати алгоритм Brotli до великих колекцій файлів, де завдяки даному методу можна отримати значне прискорення роботи алгоритму в порівнянні з іншими.

**3.3 Безпека**

.

Завдяки цьому алгоритму можна проводити атаки на веб-браузери користувача через експлуатацію переповнення буфера при відновленні (buffer overflow) якщо ввести некоректні стиснуті дані.

Відповідно, реалізований декомпресор має виконувати перевірку використання пам’яті, яка займає дані відновлені зі стиснутого потоку. Іншим можливим способом атаки на систему є атака через алгоритм відновлення. Тоді,за подання некоректно стиснутих даних, декомпресор використовує непропорційно багато ресурсів.

Кількість ресурсів яку використовує декомпресор для обробки одного мета-блоку на пряму залежить від розміру нестисненого мета-блоку, який закодовано на початку стиснутого мета-блоку,та розміру розсувного вікна , який закодований на початку потоку brotli. Отже, реалізації декомпресора в системах, де критичним є використання даних ресурсів , мають проводити постійну перевірку даних параметрів.

Необхідно встановити ліміт на використання пам’яті одним відновленим мета-блоком з потоку. У випадку коли весь потік (відновлений)тримають в системі, слід перевіряти загальний розмір цього потоку перед прочитанням кожного наступного стиснутого мета-блоку.

Відповідно можливі атаки на програми з використанням алгоритму компресор(наприклад, на веб-сервер) використовують можливість переповнення буфера чи спричиняють надмірне використання ресурсів . Як зазначено вище, для зберігання потоку слід мати S(n) байт .

S(N) = N + (3 \* (N >> 16) + 5)

Тобто, компресор має автоматично виділяти S(n) байт у вихідному буфері для розміщення даних . Якщо під час виконання алгоритму цю кількість перевищено, то алгоритм має повернутися до тривіального алгоритму описаного вище або ж відкинути потік.

Використання ресурсів компресором для певного конкретного типу даних залежить від алгоритму пошуку співпадінь та алгоритму для конструювання контекстних відображень та префіксних кодів і в набагато меншій мірі від самих даних.

**3.4 Тестування роботи алгоритму на корпусах даних та веб-даних**

Нижче наведені результати тестування алгоритму на 4 різних корпусах документів різних форматів та розмірів. Усі ,за винятком першого , проведені на комп’ютері з процесором Intel Dual-Core T4400 (2.4 Ггц),та ОЗУ(6 ГБ).ОС:Windows 7

|  |  |  |  |
| --- | --- | --- | --- |
|  | **СR** | **CS** | **DS** |
| **brotli 1** | **3.381** | **98.3** | **334.0** |
| **brotli 9** | **3.965** | **17** | **354.5** |
| **brotli 11** | **4.347** | **0.5** | **289.5** |
| **deflate 1** | **2.913** | **93.5** | **323.0** |
| **deflate 9** | **3.371** | **15.5** | **347.3** |
| **zopfli** | **3.580** | **0.2** | **342.1** |
| **lzma 1** | **3.847** | **10.2** | **70.0** |
| **lzma 9** | **4.240** | **3.9** | **71.7** |
| **lzham 1** | **3.836** | **3.9** | **116.0** |
| **lzham 4** | **3.952** | **0.5** | **117.7** |
| **bzip2 1** | **3.757** | **11.8** | **40.4** |
| **bzip 2 9** | **3.869** | **12.0** | **40.2** |

**Таблиця 3.1.**Порівняльна таблиця алгоритмів стискання за різних

якості(виставлені біля назви алгоритму). Швидкість –МБ\с

|  |  |  |  |
| --- | --- | --- | --- |
|  | **CR 1** | **CR 4** | **CR 9** |
| **bib** | **2.794** | **3.113** | **3.406** |
| **book1** | **2.170** | **2.313** | **2.770** |
| **book2** | **2.726** | **3** | **3.353** |
| **geo** | **1.428** | **1.470** | **1.562** |
| **news** | **2.527** | **2.774** | **2.975** |
| **obj1** | **1.75** | **1.909** | **2.1** |
| **obj2** | **2.659** | **3.025** | **3.361** |
| **paper1** | **2.476** | **2.736** | **2.888** |
| **paper2** | **2.454** | **2.7** | **2.892** |
| **paper3** | **2.3** | **2.5** | **2.556** |
| **paper4** | **2.176** | **2.176** | **2.176** |
| **paper5** | **2** | **2.4** | **2.4** |
| **paper6** | **2.533** | **2.714** | **2.923** |
| **pic** | **8.655** | **9.653** | **10.458** |
| **progc** | **2.4375** | **2.785** | **3** |
| **progl** | **3.684** | **4.117** | **4.375** |
| **progp** | **3.5** | **4.083** | **4.414** |
| **trans** | **4.181** | **4.842** | **5.111** |

**Таблиця 3.2** Порівняння відношення стискання , на основі документів з корпусу Калгарі

|  |  |  |  |
| --- | --- | --- | --- |
|  | **E.coli** | **bible.txt** | **world192.txt** |
| **CR 1** | **3.282** | **3.149** | **3.272** |
| **CR 4** | **3.775** | **3.391** | **4.628** |
| **CR 9** | **3.845** | **4.052** | **4.910** |
| **CS 1** | **15.15** | **14.78** | **14.74** |
| **CS 4** | **2.93** | **3.35** | **1.63** |
| **CS 9** | **0.37** | **0.63** | **1.04** |
| **DS 1** | **340.20** | **374.7** | **393.1** |
| **DS 4** | **421.3** | **428.0** | **477.3** |
| **DS 9** | **491.5** | **559.4** | **477.3** |

**Таблиця 3.2** Порівняння відношення стискання при різних параметрах якості на прикладі документів з колекції The Large Corpus

|  |  |  |  |
| --- | --- | --- | --- |
|  | **DS 1** | **DS 4** | **DS 9** |
| **bib** | **106.1** | **106** | **106.1** |
| **book1** | **244.3** | **366.3** | **366.5** |
| **book2** | **291.2** | **291.2** | **291.2** |
| **geo** | **97.6** | **119.6** | **97.65** |
| **news** | **359.6** | **359.6** | **359.6** |
| **obj1** | **205** | **157.1** | **157.1** |
| **obj2** | **235.3** | **147.6** | **235.3** |
| **paper1** | **50.6** | **56.1** | **50.6** |
| **paper2** | **78.3** | **78.3** | **65.3** |
| **paper3** | **44.3** | **44.3** | **44.3** |
| **paper4** | **12.6** | **12.6** | **12.6** |
| **paper5** | **11.4** | **11.4** | **11.4** |
| **paper6** | **36.5** | **36.6** | **36.3** |
| **pic** | **543.8** | **489.4** | **489.4** |
| **progc** | **37.4** | **37.7** | **37.7** |
| **progl** | **68.3** | **68.2** | **113.8** |
| **progp** | **78.4** | **47** | **47** |
| **trans** | **89.3** | **89.3** | **89.3** |

**Таблиця 3.4** Порівняння документів з корпусу Калгарі за швидкістю відновлення

|  |  |  |  |
| --- | --- | --- | --- |
|  | **CS 1** | **CS 4** | **CS 9** |
| **bib** | **5.39** | **5.90** | **1.15** |
| **book1** | **8.94** | **3.57** | **0.73** |
| **book2** | **8.32** | **4.48** | **1.20** |
| **geo** | **6.97** | **4.88** | **0.75** |
| **news** | **12.4** | **4.49** | **1.38** |
| **obj1** | **2.05** | **2.05** | **0.34** |
| **obj2** | **11.76** | **4.1** | **1.46** |
| **paper1** | **5.06** | **2.53** | **0.73** |
| **paper2** | **7.83** | **3.01** | **1.005** |
| **paper3** | **4.32** | **4.33** | **0.887** |
| **paper4** | **1.26** | **1.58** | **0.63** |
| **paper5** | **1.14** | **0.76** | **0.57** |
| **paper6** | **3.63** | **3.63** | **0.72** |
| **pic** | **24.47** | **12.23** | **2.6** |
| **progc** | **37.47** | **1.18** | **0.75** |
| **progl** | **6.83** | **6.83** | **1.13** |
| **progp** | **47.09** | **4.7** | **0.92** |
| **trans** | **8.03** | **8.03** | **1.19** |

**Таблиця 3.5** Порівняння документів з корпусу Калгарі за відношенням стискання

|  |  |  |  |
| --- | --- | --- | --- |
|  | **CR 1** | **CR 4** | **CR 9** |
| **bootstrap.css** | **5** | **5.71** | **6** |
| **bootstrap.min.css** | **4.476** | **5.22** | **6.26** |
| **bootstrap.js** | **4.809** | **5.94** | **7.21** |
| **carousel.js** | **3.2** | **4** | **4** |
| **collapse.js** | **3.25** | **4.33** | **6.5** |
| **FileSystem.html** | **7.5** | **10** | **15** |
| **HTTPFileSystem.html** | **7.8** | **9.75** | **13** |
| **HadoopPlugin.html** | **4.83** | **5.8** | **7.25** |

**Таблиця 3.6** Порівняння веб-документів за відношенням стискання з різними параметрами якості

|  |  |  |  |
| --- | --- | --- | --- |
|  | **CS 1** | **CS 4** | **CS 9** |
| **bootstrap.css** | **11.63** | **11.63** | **1.93** |
| **bootstrap,min.css** | **13.02** | **4.55** | **1.75** |
| **bootstrap.js** | **9.84** | **4.92** | **1.64** |
| **carousel.js** | **1.5** | **1.5** | **0.75** |
| **collapse.js** | **3.95** | **1.18** | **0.59** |
| **FileSystem.html** | **2.88** | **2.88** | **1.44** |
| **HTTPFileSystem.html** | **3.77** | **3.77** | **1.88** |
| **HadoopPLugin.html** | **2.74** | **2.74** | **1.37** |

**Таблиця 3.7** Порівняння веб-документів за швидкістю стискання з різними параметрами якості

|  |  |  |  |
| --- | --- | --- | --- |
|  | **DS 1** | **DS 4** | **DS 9** |
| **bootstrap.css** | **116.3** | **116.3** | **116.3** |
| **bootstrap.min.css** | **91.1** | **45.5** | **91.1** |
| **bootstrap.js** | **98.4** | **49.2** | **98.4** |
| **carousel.js** | **1.5** | **1.5** | **1.5** |
| **collapse.js** | **11.8** | **11.8** | **11.8** |
| **FileSystem.html** | **28.8** | **28.8** | **28.8** |
| **HTTPFileSystem.html** | **37.7** | **37.7** | **37.7** |
| **HadoopPlugin.html** | **27.4** | **27.4** | **27.4** |
|  |  |  |  |
|  |  |  |  |

**Таблиця 3.8** Порівняння веб-документів за швидкістю відновлення з різними параметрами якості

**3.5 Результати та висновки**

Таблиця 3.1 вказує на швидкість відновлення та відношення стискання в Кантерберійському корпусі. Ми можемо побачити, що алгоритм Brotli при виставленому параметрі якості(quality) 1 стискає та відновлює приблизно з тією ж швидкістю що й deflate. Але, він має краще відношення стискання приблизно на 12-16 %. За параметру 9 Brotli подібний до Deflate 9. Brotli 11 значно швидший в стисканні за Zopfli та має на 20 -26 % вище відношення стискання .

Brotli швидше відновлює дані аніж Deflate для тестових корпусів. В той час як алгоритми LZMA та LZHAM значно повільніші за deflate. Геометричне середнє для швидкостей відновлення дорівнює 342.МБ\с для Brotli,та 323.6 МБ\с для deflate .

При порівнянні алгоритмів слід також враховувати не тільки швидкості але й відношення стискання. При цьому , відношення стискання для brotli 1 ближче до deflate 9 і навіть перевищує його. Наприклад, при стисканні Кантерберійського корпусу при відношенні 3.3 brotli 1 стискає при швидкості 98.3 МБ\с., а deflate всього зі швидкістю - 15.5МБ\с.

Даний експеримент був проведений з 22 бітним розсувним вікном (за замовчуванням в Brotli). Алгоритми, такі як LZMA та LZHAM ,як правило, вживають з більшим розсувним вікном.

Brotli використовує статичний словник для швидшого стискання коротших файлів , а отже, тексти що містять слова на популярних мовах та слова яких мають високі частоти появи, будуть швидше стиснені та з більшим відношенням стискання. Інші алгоритми також можуть бути модифіковані таким чином, аби отримати дещо кращі відношення стискання .

Зокрема,слід взяти до уваги, що Кантерберійський корпус містить в собі в основному короткі документи англійською, і відповідно, значна частина слів були одразу закодовані через їхні відповідники в словнику .

Стосовно таблиці 3.2 ми бачимо що більшість файлів мають відношення стискання від 2.3 до 3.2(округлено). Проте, залежно від роду даних з яких був створений той чи інший файл, відношення стискання може радикально відрізнятися. Зокрема, Pic - файл, який був зроблений з відповідного зображення, мав набагато вище відношення стискання , аніж всі решта.

Той факт що при стисканні зображень вони мають на порядок вище відношення стискання, використовують в атаках на веб-сервери ,як зазначено в підпункті стосовно безпеки.

Завдяки таблиці 3.3 ми можемо наочно продемонструвати , що при зміні параметру якості на більший, ми не завжди отримуємо більшу швидкість відновлення чи вище відношення стискання . Залежно від контенту файлу, вже при стисканні за малих значень якості і вищих значень стискання, ми маємо максимально допустиму швидкість відновлення для файлу , та однакове відношення якості. Це вкрай важливо адже ідентифікація таких файлів залежно від їхнього контенту суттєво пришвидшила б роботу застосувань,в яких можна було б за менших значень якості стискати максимально ефективно веб-дані не зменшуючи швидкості стискання .

Таблиця 3.4 чудово демонструє те, що швидкість відновлення для багатьох файлів незалежно від якості їхнього стискання є однаковою ,тому доцільніше використовувати алгоритм з параметром 1 , якщо пріоритетом є швидке відновлення даних.

Стосовно швидкості стискання, то ми бачимо дуже сильний спад швидкості при переході від параметру якості 1 до 4 і до 9. При цьому, варто зауважити,що ті файли які мали набагато вищу за середню швидкість стискання скажімо понад 30 МБ\с потім втрачали до 90 % своєї швидкості .

А файли які були стиснуті вже за параметру якості 1 з низькою швидкістю , не втрачали у тих же пропорціях швидкість стискання при параметрах 4 та 9.

Стосовно веб-даних, а саме великих файлів формату CSS, HTML, JS, то стискання цих файлів має набагато вище відношення стискання , аніж файли з корпусу. Причин кілька. По–перше, ці формати містять багато стандартизованих слів англійською,а отже більшість з них зустрічаються в статичному словнику. Саме завдяки цьому, та ефективній структурі мета-блоків, яка дозволяє швидко кодувати тексти з посиланнями на вже записані до словника слова. По-друге, це форма даних в цих файлах. Вона набагато більш стандартна, аніж у файлах з корпусів, де часто трапляються великі послідовності символів та фраз які є унікальними.

Стосовно стискання веб-даних, то цікавою особливістю є те, що швидкість стискання в них часто майже не спадає при параметрах якості 1 та 4, але майже вдвічі при параметрі 9. Це можна використати таким чином, щоб кодувати такі документи параметром 4 ,з майже найбільшою швидкістю стискання та більшим відношенням стискання. Це у випадку коли прерогативою є більш збалансоване використання ресурсів часу та місця.

**ВИСНОВКИ**

В цій роботі був детально розібраний принцип роботи алгоритму Brotli для стискання вбе-даних. В теоретичній частині було пояснено спосіб кодування та представлення інформації, які використовували за основу вже відомі алгоритми але з певними впровадженнями, які сильно пришвидшують роботу алгоритму. Було також розібрано деякі особливості реалізації цього алгоритму на мові С++ та С.

З практичної точки зору, були проведені експерименти які засвідчують переваги даного алгоритму над сучасними аналогами. Також тестування реалізації цього алгоритму показали відмінності в роботі його з різними за змістовим заповненням файлами та особливості впливу налаштування параметрів алгоритму на загальні результати швидкостей стискання,відновлення та відношення стискання. На основі результатів були емпірично виведені поради використання алгоритму при різних налаштуваннях, які за різних параметрів дозволяють ощадливіше використовувати той чи інший ресурс на вимогу розробника .

**СПИСОК ПОСИЛАНЬ**

1 .Alakuijala, Jyrki; Kliuchnikov, Evgenii; Szabadka, Zoltan; Vandevenne, Lode (2015-09-22)[Електронний ресурс]. ["Comparison of Brotli, Deflate, Zopfli, LZMA, LZHAM and Bzip2 Compression Algorithms"](http://www.gstatic.com/b/brotlidocs/brotli-2015-09-22.pdf" \t "_blank) (PDF). Google. Retrieved 25 October2015.

2.Chirgwin Richard (23 September 2015). ["Google's new squeeze: Brotli compression open-sourced"](http://www.theregister.co.uk/2015/09/23/googles_brotli_compression_opensourced/). The Register.

3.Deutsch, P., "DEFLATE Compressed Data Format Specification version 1.3", RFC 1951, Aladdin Enterprises, May 1996. [Електронний ресурс].

<http://www.ietf.org/rfc/rfc1951.txt>

4.P.Deutsch, RFC 1952—GZIP file format specification version 4.3, [Електронний ресурс].

<http://www.ietf.org/rfc/rfc1952.txt>

5.Calgary corpus: <http://www.datacompression.info/Corpora/CalgaryCorpus.zip>

6.Canterbury corpus: http://corpus.canterbury.ac.nz/resources/cantrbry.zip

7.Huffman, D. A., "A Method for the Construction of Minimum Redundancy Codes", Proceedings of the Institute of Radio Engineers, September 1952, Volume 40, Number 9, pp.1098-1101.

8.Levantovsky, V. (ed.), Levien, R. (ed.), "WOFF File Format 2.0", W3C WebFonts Working Group,http://www.w3.org/TR/WOFF2/

9.Salomon David (2007). [Data Compression: The Complete Reference](http://books.google.com/books?id=ujnQogzx_2EC&pg=PA241) (4 ed.). Springer. p. 241.

10.Solomon, David (2006-12-19). Data Compression: The Complete Reference (4 ed.). [Springer Publishing](https://www.wikiwand.com/en/Springer_Publishing). p. 245.

11.Ziv J., Lempel A., "A Universal Algorithm for Sequential Data Compression", IEEE Transactions on Information Theory, Vol. 23, No. 3, pp. 337-343.

**Додаток А**

code = 0;

bl\_count[0] = 0;

for (bits = 1; bits <= MAX\_BITS; bits++) {

code = (code + bl\_count[bits-1]) << 1;

next\_code[bits] = code;

}

for (n = 0; n <= max\_code; n++) {

len = tree[n].Len;

if (len != 0) {

tree[n].Code = next\_code[len];

next\_code[len]++;

}

**Додаток Б**

|  |  |  |
| --- | --- | --- |
| **Код** | **Біти** | **Довжина** |
| **0** | **0** | **0** |
| **1** | **0** | **1** |
| **2** | **0** | **2** |
| **3** | **0** | **3** |
| **4** | **0** | **4** |
| **5** | **0** | **5** |
| **6** | **1** | **6,7** |
| **7** | **1** | **8,9** |
| **8** | **2** | **10-13** |
| **9** | **2** | **14-17** |
| **10** | **3** | **18-25** |
| **11** | **3** | **26-33** |
| **12** | **4** | **34-49** |
| **13** | **4** | **50-65** |
| **14** | **5** | **66-97** |
| **15** | **5** | **98-129** |
| **16** | **6** | **130-193** |
| **17** | **7** | **194-321** |
| **18** | **8** | **322-577** |
| **19** | **9** | **578-1089** |
| **20** | **10** | **1090-2113** |
| **21** | **12** | **2114-5209** |
| **22** | **14** | **6210-22593** |
| **23** | **24** | **22594-16799909** |

**Додаток В**

|  |  |  |
| --- | --- | --- |
| **Код** | **Біти** | **Довжина** |
| **0** | **0** | **2** |
| **1** | **0** | **3** |
| **2** | **0** | **4** |
| **3** | **0** | **5** |
| **4** | **0** | **6** |
| **5** | **0** | **7** |
| **6** | **0** | **8** |
| **7** | **0** | **9** |
| **8** | **1** | **10-11** |
| **9** | **1** | **12-13** |
| **10** | **2** | **14-17** |
| **11** | **2** | **18-21** |
| **12** | **3** | **22-29** |
| **13** | **3** | **30-37** |
| **14** | **4** | **38-53** |
| **15** | **4** | **54-69** |
| **16** | **5** | **70-101** |
| **17** | **5** | **102-133** |
| **18** | **6** | **134-197** |
| **19** | **7** | **198-325** |
| **20** | **8** | **326-581** |
| **21** | **9** | **582-1093** |
| **22** | **10** | **1094-2117** |
| **23** | **24** | **2118-16779333** |

**Додаток Г**

|  |  |  |
| --- | --- | --- |
| **Код** | **Біти** | **Довжина** |
| **0** | **2** | **1-4** |
| **1** | **2** | **5-8** |
| **2** | **2** | **9-12** |
| **3** | **2** | **13-16** |
| **4** | **3** | **17-24** |
| **5** | **3** | **25-32** |
| **6** | **3** | **33-40** |
| **7** | **3** | **41-48** |
| **8** | **4** | **49-64** |
| **9** | **4** | **65-80** |
| **10** | **4** | **81-96** |
| **11** | **4** | **97-112** |
| **12** | **5** | **113-144** |
| **13** | **5** | **145-176** |
| **14** | **5** | **177-208** |
| **15** | **5** | **209-240** |
| **16** | **6** | **241-304** |
| **17** | **6** | **305-368** |
| **18** | **7** | **369-496** |
| **19** | **8** | **497-752** |
| **20** | **9** | **753-1264** |
| **21** | **10** | **1265-2288** |
| **22** | **11** | **2289-4336** |
| **23** | **12** | **4337-8432** |
| **24** | **13** | **8433-16624** |
| **25** | **24** | **16625-16793840** |

**Додаток Ґ**

|  |  |
| --- | --- |
| **Значення** | **Паттерн** |
| **10** | **0100001** |
| **11** | **0110001** |
| **12** | **1000001** |
| **13** | **1010001** |
| **14** | **1100001** |
| **15** | **1110001** |
| **16** | **0** |
| **17** | **0000001** |
| **18** | **0011** |
| **19** | **0101** |
| **20** | **0111** |
| **21** | **1001** |
| **22** | **1011** |
| **23** | **1101** |
| **24** | **1111** |

**Додаток Д**

|  |  |
| --- | --- |
| **Значення** | **Паттерн** |
| **1** | **0** |
| **2** | **0001** |
| **3-4** | **х0101** |
| **5-8** | **хх0101** |
| **9-16** | **ххх0111** |
| **17-32** | **хххх1001** |
| **33-64** | **ххххх1011** |
| **65-128** | **хххххх1101** |
| **129-256** | **ххххххх1111** |

**Додаток Е**

string BrotliCompressTrivial(const string& u) {

if (u.empty()) {

return string(1, 6);}

int i;

string c;

c.append(1, 12); for (i = 0; i + 65535 < u.size(); i += 65536) {

c.append(1, 248);

c.append(1, 255);

c.append(1, 15);

c.append(&u[i], 65536);}

if (i < u.size()) {

int r = u.size() - i - 1;

c.append(1, (r & 31) << 3);

c.append(1, r >> 5);

c.append(1, 8 + (r >> 13));

c.append(&u[i], r + 1);}

c.append(1, 3);

return c;

}