УДК 004.627
DOI: 10.18523/2617-3808.2019.2.43-49

Глибовець А. М., Яблонський В. Р.

ЕКСПЕРИМЕНТАЛЬНЕ ПОРІВНЯННЯ АЛГОРИТМІВ КОМПРЕСІЇ ДАНИХ

Об'єми даних, які зберігаються та передаються, постійно ростуть. Коли потрібно передати великі об'єми даних, на допомогу приходить компресія. Добре підібраний алгоритм компресії здатен зменшити розмір даних у середньому на 60 %. Проблема появи нових і модифікації чи оптимізації старих алгоритмів компресії стоїть дуже гостро. У статті розглянуто деякі відомі сьогодні алгоритми компресії. Наведено короткий опис основних властивостей і варіантів реалізації таких алгоритмів. У рамках роботи над статтею було реалізовано ці алгоритми та проведено експериментальний аналіз їхньої якості та швидкості роботи. Робота може бути цікавою та корисною дослідникам галузі компресії даних.

Ключові слова: компресія, алгоритм, кодування, код, стиснення, дані.

Вступ

Алгоритми стискання даних можна поділити на два великі класи – стискання з втратами і без. Алгоритми стискання без втрат застосовують для зменшення об'єму даних. Вони працюють таким чином, що дають змогу відновити дані точно такими, якими вони були до стискання. У цій статті ми будемо розглядати лише алгоритми стискання даних без втрат.

Основний принцип алгоритмів стискання базується на припущенні, що в будь-якому файлі, який містить невипадкові дані, інформація частково повторюється. Використовуючи статистичні математичні моделі, можна визначити ймовірність повтору конкретних комбінацій символів. Після цього можна створити коди, яким відповідатимуть вибрані фрази, і надати фразам, які часто повторюються, найкоротші коди. Для цього використовують різні техніки, наприклад: ентропійне кодування, кодування повторів і стискання за допомогою словника.

У цій роботі ми проаналізуємо відомі сьогодні алгоритми компресії даних щодо швидкості та якості компресії і декомпресії цих алгоритмів на даних різних об'ємів та форматів.

1. Огляд алгоритмів компресії

1.1. Кодування Шеннона–Фано

Кодування Шеннона–Фано (англ. Shannon-Fano coding) – алгоритм префіксного неоднорідного кодування. Належить до ймовірнісних методів стиснення (точніше, методів контекстного моделювання нульового порядку). Як і алгоритм Хаффмана, алгоритм Шеннона–Фано використовує надмірність повідомлення укладених у неоднорідному розподілі частот його символів (первинного) алфавіту, тобто замінює коди символів, які частіше використовують, короткими двійковими послідовностями, а коди більш рідкісних символів – довшими двійковими послідовностями.

Символи первинного алфавіту m1 виписують у порядку зменшення ймовірностей. Символи отриманого алфавіту ділять на дві частини, сумарні ймовірності символів яких максимально близькі один одному.

У префіксному коді для першої частини алфавіту присвоюють двійкову цифру «0», другої частини – «1». Отримані частини рекурсивно діляться, і їхнім частинам призначають відповідні двійкові цифри у префіксному коді.

Коли розмір підалфавіту стає рівним нулю або одиниці, то наступне подовження префіксного коду для відповідних йому символів первинного алфавіту не відбувається. Таким чином, алгоритм привласнює різним символам префіксні коди різної довжини. На кроці ділення алфавіту існує неоднозначність, оскільки різниця сумарних ймовірностей p_0 - p_1 може бути однакова для двох варіантів поділу (враховуючи, що всі символи первинного алфавіту мають імовірність більше нуля).

Код Шеннона–Фано будується за допомогою дерева. Побудова цього дерева починається від кореня. Вся множина кодованих елементів відповідає кореню дерева (вершині першого рівня).

Воно розбивається на дві підмножини з приблизно однаковими сумарними ймовірностями. Ці підмножини відповідають двом вершинам другого рівня, які з'єднуються з коренем. Далі кожна з цих підмножин розбивається на дві підмножини з приблизно однаковими сумарними ймовірностями. Їм відповідають вершини третього рівня. Якщо підмножина містить єдиний елемент, то йому відповідає кінцева вершина кодового дерева; така підмножина розбиттю не підлягає. Подібним чином чинимо до тих пір, поки не отримаємо всі кінцеві вершини. Гілки кодового дерева розмічаємо символами 1 і 0.

При побудові коду Шеннона–Фано розбиття множини елементів може бути обрано, взагалі кажучи, декількома способами. Вибір розбиття на рівні n може погіршити варіанти розбиття на наступному рівні (n + 1) і призвести до неоптимальності коду в цілому. Іншими словами, оптимальна поведінка на кожному кроці шляху ще не гарантує оптимальності всієї сукупності дій. Тому код Шеннона–Фано не є оптимальним у загальному сенсі, хоча і дає оптимальні результати при деяких розподілах ймовірностей.

Розглянемо приклад. Нехай маємо текст із такими характеристиками зустрічальності: A (50), B (39), C (18), D (49), E (35), F (24). Тоді матимемо дерево, зображене на рисунку.

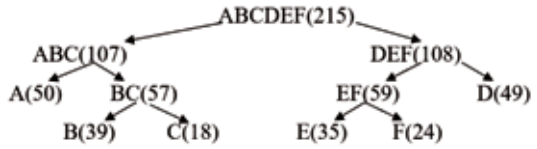

Рисунок. Приклад дерева розбиття

Отриманий код: A – 11, B – 101, C – 100, D – 00, E – 011, F – 010.

Кодування Шеннона–Фано є досить старим методом стиснення і на сьогодні не становить особливого практичного інтересу. У більшості випадків довжина послідовності, стиснутої за цим методом, дорівнює довжині стиснутої послідовності з використанням кодування Хаффмана. Але на деяких послідовностях можуть сформуватися неоптимальні коди Шеннона–Фано, тому більш ефективним вважають стиснення методом Хаффмана [1; 4].

1.2. Алгоритм Хаффмана

Алгоритм Хаффмана – адаптивний жадібний алгоритм оптимального префіксного кодування алфавіту з мінімальною надмірністю. У наш час його використовують у багатьох програмах стиснення даних.

Цей метод кодування складається з двох основних етапів: побудова оптимального кодового дерева і побудова відображення коду-символів на основі цього дерева.

Ідея алгоритму полягає у такому: знаючи ймовірності символів у повідомленні, можна описати процедуру побудови кодів змінної довжини, що складаються з цілої кількості бітів. Символам із більшою ймовірністю ставляться у відповідність коротші коди. Коди Хаффмана мають властивість префіксності (тобто жодне кодове слово не є префіксом іншого), що дає змогу однозначно їх декодувати.

Класичний алгоритм Хаффмана має низку істотних недоліків. По-перше, для відновлення вмісту стиснутого повідомлення декодер повинен знати таблицю частот, якою користувався кодувальник. Отже, довжина стиснутого повідомлення збільшується на довжину таблиці частот, яка повинна надсилатися попереду даних, що може звести нанівець усі зусилля зі стиснення повідомлення. Крім того, необхідність наявності повної частотної статистики перед початком власне кодування вимагає двох проходів повідомленням: одного для побудови моделі повідомлення (таблиці частот і Н-дерева), іншого для власне кодування. По-друге, надмірність кодування звертається 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.

Адаптивне стиснення дає змогу не передавати модель повідомлення разом із ним самим і обмежитися одним проходом за повідомленням як за кодування, так і за декодування.

У створенні алгоритму адаптивного кодування Хаффмана найбільші складності виникають при розробці процедури поновлення моделі чергових символів. Теоретично можна було би просто вставити всередину цієї процедури повну побудову дерева кодування Хаффмана, однак такий алгоритм стиснення мав би неприйнятно низьку швидкодію, оскільки побудова Н-дерева – це занадто велика робота і робити її при обробці кожного символу нерозумно.

Оновлення дерева при зчитуванні чергового символу повідомлення складається з двох операцій.

Перша – збільшення ваги вузлів дерева. Спочатку збільшуємо вагу листа, відповідного вважають символом, за одиницю. Потім збільшуємо вагу батька, щоб привести його у відповідність із новими значеннями ваги нащадків. Цей процес триває до тих пір, поки ми не доберемося до

кореня дерева. Середнє число операцій збільшення ваги дорівнює середній кількості бітів, необхідних для того, щоб закодувати символ.

Друга операція – перестановка вузлів дерева – потрібна тоді, коли збільшення ваги вузла призводить до порушення властивості впорядкованості, тобто тоді, коли збільшена вага вузла стає більшою, ніж вага наступного порядку вузла. Якщо і далі продовжувати обробляти збільшення ваги, рухаючись до кореня дерева, то дерево перестане бути деревом Хаффмана.

Щоб зберегти упорядкованість дерева кодування, алгоритм працює в такий спосіб. Нехай нова збільшена вага вузла дорівнює W+1. Тоді починаємо рухатися по списку у бік збільшення ваги, поки не знайдемо останній вузол із вагою W. Переставимо поточний і знайдений вузли між собою в списку, відновлюючи таким чином порядок у дереві (при цьому батьки кожного з вузлів теж зміняться). На цьому операція перестановки закінчується.

Після перестановки операція збільшення ваги вузлів продовжується далі. Наступний вузол, вага якого буде збільшена алгоритмом, – це новий батько вузла, збільшення ваги якого викликало перестановку [1; 3].

1.3. Алгоритм LZSS

Алгоритм Лемпеля–Зіва–Сторера–Сжиманскі (LZSS) був представлений у 1982 р. як покращена версія LZ77. LZ77 використовує уже проглянуту частину повідомлення як словник. Щоб добитися стиснення, він намагається замінити наступний фрагмент повідомлення на вказівник у словник.

Як моделі даних LZ77 використовує вікно, що «плаває» по повідомленню, розділене на дві нерівні частини. Перша, велика за розміром, містить уже переглянуту частину повідомлення. Друга, набагато менша, є буфером, який містить ще не закодовані символи вхідного потоку. Зазвичай розмір вікна дорівнює декільком кілобайтам. Буфер набагато менший, зазвичай не більше ніж 100 байт. Алгоритм намагається знайти в словнику фрагмент, який збігається зі вмістом буфера.

Алгоритм LZ77 видає коди, які складаються з трьох елементів: зміщення в словнику щодо початку до підстрічки, що збігається зі вмістом буфера; довжина підстрічки; перший символ у буфері, що йде після підстрічки.

Алгоритм кодування такий: поки буфер не пустий, знайти найдовший збіг (позиція початку, довжина), якщо збіг знайдено, то вивести в кодовані дані пару (позиція початку, довжина) і наступний символ буфера. В іншому випадку вивести в кодовані дані пару (0, 0) і перший символ буфера.

Декодування в LZ77 ще простіше, оскільки не потрібно здійснювати пошук у словнику.

Очевидно, що швидкість кодувальника LZ77 сильно залежить від того, яким чином здійснюватиметься пошук збігу підстрічки в словнику. Якщо шукати збіг повним перебиранням усіх можливих варіантів, то очевидно, що стиснення буде дуже повільним. До того ж при збільшенні розмірів вікна для підвищення степеня стиснення швидкість роботи буде пропорційно зменшуватися. Для декодера це неважливо, бо при декодуванні не здійснюється пошук збігу.

Крім проблем зі швидкодією, у алгоритму LZ77 виникають проблеми із самим стисненням. Вони з'являються, коли кодувальник не може знайти підстрічку, що збігається у словнику, і видає стандартний 3-компонентний код, намагаючись закодувати один символ. Якщо словник має довжину 4Кб, а буфер – 16 байтів, то код <0, 0, символ> займатиме 3 байти.

На відміну від LZ77, LZSS прораховує, чи не збільшився розмір результату заміни висхідних даних закодованими [5; 2; 4].

1.4. Алгоритм LZW

Алгоритм Лемпеля–Зіва–Велча (LZW) був представлений у 1984 р. як покращена версія LZ78. LZ78 був першим у сім'ї схем, що розвивалися паралельно з LZ77. Незалежно від можливості вказівників звертатись до будь-якої уже переглянутої стрічки, проглянутий текст розбирається на фрази, де кожна нова фраза є найдовшою з уже проглянутих плюс один символ. Вона кодується як індекс її префіксу плюс додатковий символ. Після чого нова фраза додається до списку фраз, на які можна посилатися.

Наприклад, стрічка «aaabbabaabaaabab», ділиться на 7 фраз. Кожна з них кодується як фраза, яка траплялась раніше плюс поточний символ. Наприклад, останні три символи кодуються як фраза номер 4 («ba»), за якою іде символ «b». Фраза номер 0 – пуста стрічка.

Таблиця 1. **Приклад кодування LZ78**

Вхід	a	aa	b	ba	Baa	Baaa	Bab
Номер фрази	1	2	3	4	5	6	7
Вихід	(0,a)	(1,a)	(0,b)	(3,a)	(4,a)	(5,a)	(4,b)

Дальність просування вперед указника необмежена (тобто немає вікна), тому у процесі виконання кодування накопичується все більше фраз. Допуск великої їх кількості потребує під час розбору збільшення розміру вказівника. Коли розібрано *p* фраз, указник представляється [log p] бітами. На практиці, словник не може продовжувати рости безкінечно. При вичерпуванні доступної пам'яті, вона очищається і кодування продовжується немовби з початку нового тексту.

Перехід від LZ78 до LZW паралельний переходу від LZ77 до LZSS. Включення явного символу у вивід після кожної фрази часто є затратним. LZW управляє вилученням цих символів, тому вивід містить лише вказівник. Це досягається ініціалізацією списку фраз, який містить усі символи вхідного алфавіту. Останній символ кожної нової фрази кодується як перший символ наступної фрази. Особливої уваги потребує ситуація, що виникає при розкодовуванні, якщо фраза кодувалась з допомогою іншої фрази, що передує їй, але це не є проблемою, яку не можна визначити [5; 2; 4].

2. Реалізація і експериментальне порівняння алгоритмів компресії

2.1. Технології і середовища реалізації

Усі алгоритми були розроблені мовою C# в Visual Studio 2013. Для роботи з бітами використовували такі структури даних: BitArray, List<bool> і в деяких випадках string.

Усі експерименти проводили на комп'ютері з процесором Intel® Core™ i5-6200U CPU @ 2,30-2,40 GHz, об'ємом оперативної пам'яті 8 Гб та 64-розрядній операційній системі Windows 10.

Для проведення експериментів використовували два види файлів різного розміру. Перший – файл, заповнений байтами за допомогою рандомайзера, далі – ЗФ (згенерований файл). Другий – файл, заповнений текстами із реальних книжок, далі – РФ (реальний файл).

2.2. Реалізація алгоритму Шеннона–Фано

Алгоритм працює з байтами. Спочатку алгоритм проходить по вхідному файлу, підраховуючи кількість зустрічей кожного байта. Далі кожному байту приписується код за результатами застосування коду Шеннона–Фано. Алгоритм заново проходить по файлу, замінюючи кожен байт на відповідний код.

Закодоване повідомлення складається з двох частин. Перша – коди, кількість яких рівна довжині словника, а саме перші 8 біт – довжина словника, далі для кожного коду записується: 8 біт – значення байта, 8 біт – довжина коду, 2-255 біт код. Друга – власне закодоване повідомлення, а саме, кожен байт замінюється на такий код: 1 біт – індикатор останнього коду, 2-255 біт – код.

2.3. Реалізація алгоритму Хаффмана

Алгоритм працює з байтами. Спочатку алгоритм проходить по вхідному файлу, підраховуючи кількість зустрічей кожного байта. Далі за методом Хаффмана будується дерево ймовірностей. Після цього кожному байту приписується код згідно з побудованим деревом. Алгоритм заново проходить по файлу, замінюючи кожен байт на відповідний код.

Закодоване повідомлення складається з двох частин. Перша – коди, кількість яких рівна довжині словника, а саме перші 8 біт – довжина словника, далі для кожного коду записується: 8 біт – значення байта, 8 біт – довжина коду, 2-255 біт – код. Друга – власне закодоване повідомлення, а саме, кожен байт замінюється на такий код: 1 біт – індикатор останнього коду, 2-255 біт – код.

2.4. Реалізація алгоритму LZSS

Як фразу і словник використовують List<byte>. Пошук входження відбувається повним перебором, через що час кодування доволі великий.

Алгоритм пробігається по файлу в пошуку фраз, які можна замінити. Мінімальна довжина фрази – 6 байт. Якщо фразу знайдено в словнику, то зчитується наступний символ, якщо ні, то фраза додається в словник. Якщо довжина фрази менша шести, то код залишається таким самим, як і вхідна фраза. Інакше фраза кодується входженням фрази без останнього символу в словник, плюс останній символ.

Закодоване повідомлення для кожного коду має таку структуру. 1 біт – індикатор того, код закодований чи ні. Якщо код не закодований, то далі йде 8-48 біт – незакодоване повідомлення. Якщо закодований, то код матиме таку структуру: 1 біт – індикатор того, 8 чи 16 біт займає індекс, 8-16 біт – значення індексу, 1 біт – індикатор того, 8 чи 16 біт займає довжина, 8-16 біт – значення довжини, 8 біт – значення останнього символу.

2.5. Реалізація алгоритму LZW

Як словник використовують Dictionary<string,int>, як фразу – string. Початковий словник складається з 256 символів. Розмір коду починається з 8 біт.

Алгоритм пробігається по файлу в пошуку фраз, які можна замінити кодом зі словника. Мінімальна довжина фрази – 1 байт. Якщо фраза знайдена в словнику, то зчитується наступний символ, якщо ні, то фраза додається в словник під наступним порядковим індексом. Довжина коду збільшується щоразу, коли кількість елементів в словнику стає рівною степені двійки. Фраза, без останнього символу, заміняється кодом зі словника.

За декодування створюється такий самий початковий словник, як і за кодування.

2.6. Порівняння часу виконання

Згідно з даними, наведеними в табл. 2, найкращі результати за часом компресії показав алгоритм Шеннона–Фано, Хаффмана і LZW. Найгірший результат – алгоритм LZSS зі словником розміром 65534.

Таблиця 2. **Порівняння часу кодування**

		Шеннона–Фано	**Хаффмана**	**LZW**	**LZSS 8192**	**LZSS 16384**	**LZSS 32768**	**LZSS 65534**
ЗФ	**1459**	00.054	00.122	00.019	00.022	00:00.009	00:00.009	0:00:00.031
РФ	**1028**	00.007	00.029	00.009	00.007	00:00.316	00:00.006	0:00:00.005
ЗФ	**14931**	00.022	00.058	00.035	00.537	00:00.574	00:00.562	0:00:00.615
РФ	**14931**	00.015	00.048	00.022	00.454	00:00.569	00:00.504	0:00:00.551
ЗФ	**149617**	00.093	00.176	00.281	07.398	00:11.959	00:21.842	0:00:42.708
РФ	**150255**	00.349	00.111	00.202	06.372	00:09.845	00:17.776	0:00:34.538
ЗФ	**1497303**	00.665	00.938	03.076	01:13.463	02:11.002	04:13.796	0:08:57.072
РФ	**1497891**	00.659	00.868	01.452	01:02.488	01:48.840	03:23.150	0:06:39.022
ЗФ	**14981041**	06.876	08.176	25.784	13:31.746	21:27.130	45:14.619	1:38:00.897
РФ	**14987013**	06.850	08.017	11.802	09:27.691	18:36.507	37:02.770	1:08:23.215
Середній на мегабайт		00.468	00.557	01.282	45.958	01:20.094	02:43.236	05:30.335

Згідно з даними, наведеними в табл. 3, найкращі результати за часом декомпресії показали алгоритми LZSS зі словником 8192, Шеннона–Фано, LZSS зі словником 16384 і Хаффмана. Насправді всі алгоритми показали непогані результати за часом декомпресії, окрім алгоритмів LZW.

Таблиця 3. **Порівняння часу декодування**

		LZSS 8192	**Шеннона-Фано**	**LZSS 16384**	**Хаффман**	**LZSS 32768**	**LZSS 65534**	**LZW**
ЗФ	**1459**	00.009	00.006	00.002	00.009	00.362	00.007	00:00.033
РФ	**1028**	00.003	00.004	00.002	00.003	00.002	00.002	00:00.033
ЗФ	**14931**	00.013	00.022	00.013	00.017	00.011	00.013	00:01.534
РФ	**14931**	00.010	00.015	00.010	00.012	00.009	00.013	00:00.279
ЗФ	**149617**	00.160	00.108	00.125	00.145	00.149	00.212	01:10.251
РФ	**150255**	00.107	00.080	00.091	00.136	00.129	00.156	00:22.255
ЗФ	**1497303**	01.428	01.280	01.799	01.622	02.013	02.979	52:46.227
РФ	**1497891**	01.079	01.003	01.343	01.213	01.739	02.182	09:43.593
ЗФ	**14981041**	11.203	12.996	15.175	16.664	23.384	36.049	
РФ	**14987013**	09.283	09.433	10.600	11.110	15.474	24.512	
Середній на мегабайт		00.700	00.749	00.876	00.929	01.300	01.986	19:15.313

2.7. Порівняння ефективності компресії

Згідно з даними, наведеними в табл. 4, найкращий результат компресії показав алгоритм LZW, середній рівень стиснення якого становить 32,12 %. Цей алгоритм дає змогу досягти стиснення до 65,52 %. Одразу за LZW йдуть алгоритми Шеннона–Фано і Хаффмана. Найгірший результат показав алгоритм LZSS із розміром словника 8192. Середні відсоток стиснення становить 8,58 %, а максимальний – 22,05 %.

Таблиця 4. **Порівняння ефективності компресії**

		LZSS 8192	LZSS 16384	LZSS 32768	LZSS 65534	Шеннона–Фано	Хаффман	LZW
ЗФ	**1459**	-0.75 %	-0.75 %	-0.75 %	-0.75 %	-31.32 %	-30.29 %	5.41 %
РФ	**1028**	11.58 %	11.58 %	11.58 %	11.58 %	26.36 %	27.53 %	36.87 %
ЗФ	**14931**	-2.09 %	-1.98 %	-1.98 %	-1.98 %	4.26 %	4.64 %	1.71 %
РФ	**14931**	20.03 %	20.03 %	20.03 %	20.03 %	30.86 %	32.35 %	50.46 %
ЗФ	**149617**	-2.08 %	-2.07 %	-2.07 %	-2.06 %	7.85 %	7.97 %	7.43 %
РФ	**150255**	22.05 %	26.87 %	29.92 %	31.44 %	30.57 %	30.96 %	56.31 %
ЗФ	**1497303**	-2.08 %	-2.08 %	-2.07 %	-2.06 %	8.22 %	8.26 %	18.82 %
РФ	**1497891**	20.51 %	25.88 %	31.35 %	36.66 %	28.74 %	29.10 %	62.29 %
ЗФ	**14981041**	-2.08 %	-2.08 %	-2.07 %	-2.07 %	8.28 %	8.29 %	16.45 %
РФ	**14987013**	20.72 %	26.15 %	31.82 %	37.39 %	29.06 %	29.34 %	65.52 %
Середній		8.58 %	10.15 %	11.57 %	12.82 %	14.29 %	14.81 %	32.13 %

Висновки

У цій статті було розглянуто питання компресії даних різними методами. Докладно описано та реалізовано алгоритми стиснення даних Шеннона–Фано, Хаффмана, LZSS та LZW. Проведено тестування аналіз і порівняння цих алгоритмів.

Найкращий рівень компресії показав алгоритм LZW, але час декодування виявився доволі великим. Оптимальні результати показали алгоритми Шеннона–Фано і Хаффмана, вони ввійшли в трійку найкращих за рівнем компресії, часом кодування і декодування. Також експериментально було підтверджено, що більшість алгоритмів компресії без втрат працюють лише зі структурованими даними, а на файлах, згенерованих на основі випадкових даних, показують скоріше негативні результати.

Список літератури

1. Ватолин Д. Методы сжатия данных. Устройство архиваторов, сжатие изображений и видео / Д. Ватолин, А. Ратушняк, М. Смирнов, В. Юкин. – Москва : ДИАЛОГ-МИФИ, 2002. – 384 с.
2. Bell Timothy. Modeling for text compression / Timothy Bell, Ian H. Witten, John G. Cleary // Journal ACM Computing Surveys (CSUR). – 1989. – Vol. 21, Issue 4. – 1989. – P. 557–591.
3. Habr [Электронный ресурс] : Алгоритмы сжатия данных без потерь – 2014. – Режим доступа: https://habr.com/en/post/231177. – Заглавие с экрана.
4. Habr [Электронный ресурс] : Алгоритмы сжатия данных без потерь, часть 2 – 2014. – Режим доступа: https://habr.com/en/post/235553. – Заглавие с экрана.
5. Habr [Электронный ресурс] : Алгоритмы LZW, LZ77 и LZ78. – 2014. – Режим доступа: https://habr.com/en/post/132683. – Заглавие с экрана.

References

Bell, Timothy, Witten, Ian, & Cleary, John. G. (1989). Modeling for Text Compression. *ACM Computing Sureys, 21*, 557–591. Retrieved from http://doi.org/ 10.1145/76894.76896.

Habr: Alhoritmy szhatiia dannykh bez poter. Retrieved from https://habr.com/ru/post/231177.

Habr: Alhoritmy szhatiia dannykh bez poter, chast 2. Retrieved from https://habr.com/ru/post/235553.

Habr: Alhoritmy LZW, LZ77 I LZ78. Retrieved from https://habr.com/en/post/132683.

Vatolin, D., Ratushniak, A., Smirnov, M., & Yukin, V. (2002). *Metody szhatiia dannykh. Ustroistvo arkhivatorov, szhatie izobrazhenii i video*. Moskva: DIALOH-MIFI.

A. Hlybovets, V. Yablonskyi

EXPERIMENTAL COMPARISON OF DATA COMPRASSION ALGORITHMS

The amount of data that is stored and transferred grows regularly and rapidly. When it comes to transferring large data volumes, a data compression algorithm can be useful. A well-chosen data compression algorithm can reduce the size of the data to up to 60%. The problem of creating new and modifying or optimizing old algorithms is up to date. This article discloses some of the most widespread algorithms of data comparison; more exactly, four of them. Shannon–Fano coding, named after Claude Shannon and Robert Fano, is a technique for constructing a prefix code based on a set of symbols and their probabilities. It is suboptimal in the sense that it does not achieve the lowest possible expected code word length like Huffman coding. Huffman code is a particular type of the optimal prefix code that is commonly used for lossless data compression. The process of finding or using such a code proceeds by means of Huffman coding; an algorithm developed by David A. Huffman. Lempel–Ziv–Storer–Szymanski (LZSS) is a lossless data compression algorithm, a derivative of LZ77 that was created in 1982 by James Storer and Thomas Szymanski. The main difference between LZ77 and LZSS is that in LZ77 the dictionary reference could actually be longer than the string it was replacing. In LZSS, such references are omitted if the length is less than the "break even" point. Furthermore, LZSS uses one-bit flags to indicate whether the next chunk of data is a literal (byte) or a reference to an offset/length pair. Lempel–Ziv–Welch (LZW) is a lossless data compression algorithm created by Abraham Lempel, Jacob Ziv, and Terry Welch. It was published as an improved implementation of the LZ78 algorithm. As part of the work on the article, these algorithms were implemented and an experimental analysis of their quality and speed was carried out. Those experiments gave the conclusion that the best compression speed results were shown by LZSS and the best compression ratio was reached by LZW. The work can be useful for researchers in the field of data compression.

Keywords: compression, algorithm, coding, code, data.

Матеріал надійшов 06.05.2019