MCIT
Modeling, control & information technologies

Прискорення виконання алгоритму зменшення розміру сформованих DEFLATE-блоків за допомогою мемоізації в процесі прогресуючого ієрархічного стиснення зображень без втрат

https://doi.org/10.31713/MCIT.2025.042

Олександр Шпортько
Національний університет водного господарства та природокористуванн
м. Рівне, Україна
ITShportko@gmail.com

Андрій Бомба
Національний університет водного господарства та природокористування
м. Рівне, Україна
abomba@ukr.net

Abstract—**Обґрунтована доцільність та описаний механізм використання мемоізації для прискорення виконання алгоритму зменшення розміру сформованих DEFLATE-блоків під час прогресуючого ієрархічного стиснення зображень без втрат. На прикладі зображень тестового набору ACT показано, що застосування мемоізації для аналітичного способу реалізації цього алгоритму дає змогу прискорити її виконання максимум на 58.9 %.**

Keywords—мемоізація, прогресуюче стиснення зображень, стиснення без втрат

I. Вступ

Будь-яка обчислювальна техніка та мобільні пристрої мають обмежені ресурси (швидкість процесора, обсяг оперативної та зовнішньої пам'ять, місткість зарядного пристрою). Швидші алгоритми дають змогу виконувати більше завдань за той самий час та економлять чи перерозподіляють витрати окремих ресурсів, тому проблема розробки методів для прискорення виконання алгоритмів є актуальною на цей час і залишатиметься такою у найближчому майбутньому. Одним з дієвих підходів для прискорення виконання алгоритмів на сьогодні є мемоізація.

II. Аналіз останніх досліджень і публікацій

Як відомо, мемоізація в програмуванні – це техніка оптимізації, яка полягає у зберіганні результатів виконання функції при заданих значеннях параметрів. Тоді в процесі повторних викликів такої функції з тими ж значеннями параметрів відразу повертається збережене значення, а не виконуються ті ж самі розрахунки ще раз [1; 2]. Також на сьогодні мемоізація використовується не лише для збільшення швидкості виконання програм, а й, наприклад, під час рекурсивного синтаксичного розкладу в узагальненому алгоритмі низхідного синтаксичного аналізу [3] чи при табулюванні значень предикатів в мовах логічного програмування. З іншого боку, для запам'ятовування результатів обчислень найчастіше використовуються масиви, які займають додаткові обсяги оперативної пам'яті [4; 5]. Тому на практиці при використанні мемоізації потрібно віднаходити баланс між додатковими витратами ресурсів обчислювального середовища і прискоренням виконання програми від їх використання. В цій праці ми покажемо, як можна застосувати мемоізацію для прискорення обчислення двійкових логарифмів, як однієї з найуживаніших функцій в процесі реалізації алгоритму зменшення розміру сформованих DEFLATE-блоків [6-9].

III. Використання формату словникової компресії DEFLATE для стиснення зображень без втрат

Загальновідомо, стиснення зображень без втрат найчастіше відбувається максимум в три етапи [9]: на першому яскравості пікселів перетворюються за допомогою предикторів; на другому контекстно-залежне кодування зменшує надлишковості між окремими фрагментами; на третьому контекстно-незалежне кодування усуває надлишковості між переважаючими значеннями яскравостей компонентів пікселів.

Серед контекстно-залежних алгоритмів у форматах графічних файлів найчастіше використовується словниковий алгоритм LZ77 [10], оскільки він забезпечує найшвидше декодування. Описуючи словникові алгоритми, фіксовану кількість попередніх закодованих неподільних елементів (літералів) вхідного потоку називають *словником*, а наступних незакодованих – *буфером*. Алгоритм LZ77 базується на заміні у вихідному потоці послідовності чергових літералів буфера посиланням на аналогічну послідовність літералів словника у вигляді пари чисел <*довжина; зміщення від закінчення словника*>. У випадку відсутності аналогічної послідовності літералів у словнику,

перший літерал буфера переноситься у вихідний потік без змін. Після цього закодовані літерали переносяться з початку буфера в кінець словника і кодування продовжується аналогічно аж до закінчення літералів вхідного потоку. Під час декодування кодів алгоритму LZ77 окремі літерали копіюються у вихідний потік без змін. Пари ж <*довжина; зміщення*> декодуються шляхом послідовного копіювання з кінця вихідного потоку за вказаним зміщенням в кінець вихідного потоку необхідної кількості літералів.

Контекстно-залежне кодування може зменшувати коефіцієнт стиснення у декілька разів в основному за рахунок однакових фрагментів. Але такі фрагменти рідко трапляються у фотореалістичних зображеннях, тому єдиним універсальним етапом стиснення зображень без втрат на сьогодні є контекстно-незалежне кодування.

Основний принцип контекстно-незалежного кодування, яке в процесі стиснення зображень без втрат застосовується для компресії значень окремих елементів, – *довжина коду довільного елемента з більшою ймовірністю не повинна перевищувати довжину коду будь-якого елемента з меншою ймовірністю*. Цей принцип базується на фундаментальному положенні теорії інформації, згідно з яким для мінімізації довжини коду послідовності елемент s_i з ймовірністю появи $p(s_i)$ доцільно кодувати $l_i = -\log_2 p(s_i)$ бітами. Тому середня довжина коду елемента блоку після застосування будь-якого контекстно-незалежного алгоритму згідно з формулою of Shannon [11], не може бути меншою *ентропії джерела*

$$H = -\sum_i p(s_i) \times \log_2 p(s_i). \qquad (1)$$

Ентропія джерела зменшується зі збільшенням нерівномірності розподілу ймовірностей (частот) між елементами [11]. Якщо частоту кожного з елементів s_i позначити через N_i , а загальну довжину послідовності – через N (очевидно, що $N = \sum_i N_i$) то, згідно статистичного означення ймовірності, $p(s_i) = N_i / N$, тому

$$l_i = -\log_2 p_i = \log_2 \frac{N}{n_i}. \qquad (2)$$

Після застосування контекстно-незалежного алгоритму середня довжина коду наближається до ентропії (1), а загальна довжина блоку наближається до *ентропійної* довжини

$$L_H = N \times H = N \log_2(N) - \sum_i N_i \log_2(N_i). \qquad (3)$$

Підвищити ефективність контекстно-незалежного кодування в процесі стиснення зображень без втрат найчастіше намагаються за допомогою *предикторів* [7-9], які під час обходу прогнозують значення яскравості кожної компоненти чергового піксела, використовуючи значення яскравостей тих самих компонентів опрацьованих раніше суміжних пікселів, оскільки дані яскравості мають між собою найбільшу степінь кореляції. В процесі використання цього підходу обчислюють і надалі кодують відхилення Δ_{uv} значення яскравості чергової компоненти піксела F_{uv} від прогнозованого обраним предиктором значення $predict_{uv}$, тобто

$$\Delta_{uv} = F_{uv} - predict_{uv} \qquad (4)$$

(*u* та *v* пробігають відповідно по всіх рядках та стовпцях компонентів пікселів зображення). Суміжні пікселі зображення найчастіше мають подібні кольори, а значить і близькі значення яскравостей відповідних компонентів, тому значення прогнозу часто збігається зі значенням яскравості чергової компоненти, найчастіше – є близьким до цього значення і рідко – значно відрізняється від нього. Тобто більшість значень Δ_{uv} виявляються близькими до нуля, що збільшує нерівномірність розподілу і, як наслідок, зменшує ентропію (1).

Природно, що алгоритм декодування кодів LZ77 має розрізняти окремі літерали та групи <*довжина; зміщення*>. У форматі DEFLATE [6] з цією метою довжини замін та окремі літерали алгоритму LZ77 кодуються разом числами в межах [0; 285]. При цьому числа з діапазону [0; 255] відповідають кодам окремих літералів, 256 позначає закінчення блоку, а числа з діапазону [257; 285] вказують на базові значення довжин. Після базових значень довжин міститься визначена форматом кількість бітів, що разом з базовим значенням однозначно визначає довжину заміни. Зміщення зберігається після відповідної довжини заміни аналогічно – у вигляді базового значення та додаткових бітів. Базове значення зміщення знаходиться в межах [0; 29]. В DEFLATE максимальне значення довжини закодованої послідовності може сягати 258, зміщення – 32768, а для кодування базових значень літералів/довжин та зміщень застосовуються різні коди HUFF [12]. Ми модифікували формат DEFLATE, застосувавши замість кодування HUFF арифметичне кодування [13], оскільки його середня довжина коду ближча до ентропії (1).

Зменшити розмір DEFLATE-блоків, розрахованих з використання (3), можливо шляхом відкидання неефективних замін алгоритму LZ77, довжина коду яких перевищує суми довжин кодів окремих літералів, які вони заміняють. Ми розробили два способи такого зменшення – *синтетичний*, який аналізує довжину коду заміни LZ77 відносно сформованого розподілу [7], та *аналітичний*, який розраховує довжину DEFLATE-блоку після можливого врахування чи повернення такої заміни.

IV. Застосування мемоізації для прогнозування довжини DEFLATE-блоків

Зрозуміло, що мемоізацію доцільно застосовувати до функцій з дискретними

значеннями аргументів, адже тоді її результати можливо зберігати в масиві з індексами-значеннями цих аргументів. Тому для обчислення довжини ентропійного коду (2) ми виконали перехід від двійкового логарифму дійсного значення його ймовірності до різниці логарифмів цілих кількостей всіх елементів та частоти чергового елемента:

$$l_i = -\log p_i = \log\frac{N}{n_i} = \log N - \log n_i . \qquad (5)$$

Відповідна функція для обчислення двійкового логарифму від цілого числа з використанням мемоізації мовою C++ використовується нами у такому вигляді:

```
double log2(UBYTE4 x)
 {if (x<MaxMemoizationLog2)
   {if (arrayLog2[x]<0)
     arrayLog2[x]=log(x)*1.4426950409;
    return arrayLog2[x]; }
  return log(x)*1.4426950409; }   ,
```

де *MaxMemoizationLog2* – кількість елементів в масиві *arrayLog2* для зберігання результатів мемоізації двійкового логарифму. Вплив цієї функції на час кодування ми дослідимо далі. Значення двійкового логарифму від додатних цілих чисел невід'ємні, тому на початку роботи кодера ми ініціалізуємо елементи масиву *arrayLog2* значенням -1. Якщо цілий додатний аргумент цієї функції не входить в діапазон індексів масиву мемоізації (надалі – діапазон мемоізації), то її значення обчислюється за допомогою стандартної математичної функції log(*x*). Інакше виконується перевірка наявності попереднього розрахованого значення двійкового логарифму від цього аргумента і у випадку, коли таке значення ще не розраховувалося (тобто *arrayLog2*[*x*]<0), двійковий логарифм від аргументу обчислюється і зберігається в масиві. Після цієї перевірки функція повертає обчислене раніше значення двійкового логарифму з масиву *arrayLog2*. Тобто для значень, які входять в діапазон мемоізації, двійковий логарифм обчислюється перший раз і зберігається в масиві, після чого надалі при наступних викликах функції береться з цього масиву, а не обчислюється ще раз.

При такому підході обчислюються не всі значення з діапазону мемоізації, а лише ті, які хоча б раз використовуються при розрахунках. Після застосування предикторів [4] частоти більшості елементів стають близькими до нуля, тому ми виконували мемоізацію саме для найменших додатних значень аргументу.

Проаналізуємо тепер зміни часу виконання синтетичного і аналітичного способів алгоритму зменшення розміру сформованих DEFLATE-блоків [6] для стиснення зображень набору ACT [14] в залежності від діапазонів мемоізації (TABLE I-II та Figure 1). Цей набір містить як дискретно-тонові (№№ 1, 2, 7), так і фотореалістичні (всі інші) зображення.

TABLE I. ЧАС ВИКОНАННЯ СИНТЕТИЧНОГО СПОСОБУ АЛГОРИТМУ ЗМЕНШЕННЯ РОЗМІРУ СФОРМОВАНИХ DEFLATE-БЛОКІВ ПРИ СТИСНЕННІ ЗОБРАЖЕНЬ НАБОРУ ACT ДЛЯ РІЗНИХ ДІАПАЗОНІВ МЕМОІЗАЦІЇ, С

Діапазон мемоізації	**№ файла**								**Середній час**
	1	***2***	***3***	***4***	***5***	***6***	***7***	***8***	
∅	0.07	0.12	0.03	0.25	0.13	0.58	0.05	0.15	0.17
[1; 99]	0.05	0.12	0.03	0.24	0.13	0.55	0.05	0.14	0.16
[1; 999]	0.05	0.11	0.03	0.24	0.13	0.55	0.05	0.14	0.16
[1; 4999]	0.05	0.11	0.03	0.24	0.13	0.55	0.05	0.14	0.16
[1; 9999]	0.05	0.11	0.03	0.24	0.13	0.55	0.05	0.14	0.16

TABLE II. ЧАС ВИКОНАННЯ АНАЛІТИЧНОГО СПОСОБУ АЛГОРИТМУ ЗМЕНШЕННЯ РОЗМІРУ СФОРМОВАНИХ DEFLATE-БЛОКІВ ПРИ СТИСНЕННІ ЗОБРАЖЕНЬ НАБОРУ ACT ДЛЯ РІЗНИХ ДІАПАЗОНІВ МЕМОІЗАЦІЇ, С

Діапазон мемоізації	**№ файла**								**Середній час**
	1	***2***	***3***	***4***	***5***	***6***	***7***	***8***	
∅	0.32	0.61	0.36	1.42	0.97	3.62	0.24	1.02	1.07
[1; 99]	0.21	0.44	0.21	1.12	0.78	2.92	0.18	0.82	0.84
[1; 999]	0.19	0.38	0.21	1.05	0.64	2.38	0.14	0.72	0.71
[1; 4999]	0.16	0.34	0.17	0.82	0.56	1.74	0.12	0.50	0.55
[1; 9999]	0.14	0.34	0.15	0.54	0.44	1.39	0.12	0.39	0.44

Бачимо, що мемоізація суттєвіше прискорює стиснення фотореалістичних зображень відносно дискретно-тонових. Це пояснюється більшою часткою неефективних замін і, відповідно, більшою кількістю ітерацій алгоритму, що призводить до частіших обчислень двійкових логарифмів. Застосування мемоізації суттєво прискорює (в середньому по набору ACT – максимум на 58.9 %) виконання реалізації аналітичного способу зменшення розміру сформованих DEFLATE-блоків і майже не впливає на час виконання реалізації синтетичного способу (прискорює максимум на 5.9 %). І це закономірно, адже синтетичний спосіб зменшення розміру DEFLATE-блоків використовує функцію обчислення двійкових логарифмів, перераховуючи довжини кодів елементів розподілів на кожній ітерації після аналізу **всіх** замін, а аналітичний спосіб для аналізу ефективності **кожної** заміни викликає цю функцію в середньому не менше десяти разів.

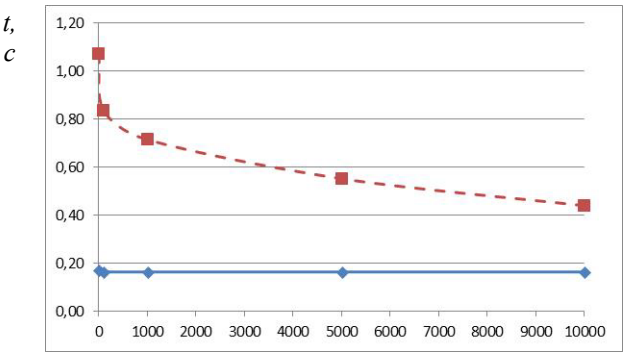

Figure 1. Залежність середнього часу виконання від ширини діапазону індексів масиву мемоізації для синтетичного (суцільна лінія) та аналітичного (пунктирна лінія) способів алгоритму зменшення розміру DEFLATE-блоків при стисненні зображень набору ACT

На реалізацію синтетичного способу в середньому по набору ACT припадає 9.3 % загального часу кодування, на виконання аналітичного способу без мемоізації – 38.2 %, а на роботу аналітичного способу з масивом мемоізації розміром біля 78 Кб – 20.3 % (останній рядок

TABLE II). Саме такий масив ми рекомендуємо використовувати на практиці для аналітичного способу зменшення розміру DEFLATE-блоків. Оскільки синтетичний та аналітичний способи алгоритму забезпечують приблизно однакові коефіцієнти стиснення, а перший з них – суттєво швидший, то в кодерах ми рекомендуємо реалізовувати саме синтетичний спосіб алгоритму зменшення розміру сформованого DEFLATE-блоку.

V. Висновки

Наведені результати підтверджують відому закономірність: чим більше значень запам'ятовується в процесі мемоізації – тим швидше виконання програми. Але самого лише збільшення масиву мемоізації для суттєвого прискорення програми недостатньо (як для синтетичного способу зменшення розміру DEFLATE-блоку на Figure 1). Потрібно, щоб ще й значення, які зберігаються в масиві мемоізації, використовувалися багаторазово. Зазначимо також, що кожен стиснутий блок в нашій реалізації (і, відповідно, частота окремого елемента) та змішення заміни у форматі DEFLATE не перевищують 32768. Тому використання більших масивів мемоізації не призведе до прискорення кодування.

Крім використання додаткових обсягів оперативної пам'яті, реалізація мемоізації ще й збільшує розмір коду, тобто розмір самого кодера. Тому для реалізації синтетичного способу зменшення розміру DEFLATE-блоку використовувати мемоізацію недоцільно, а для аналітичного способу вона дає суттєвий (до 58.9 %) виграш у часі кодування.

Література

[1] О. В. Шпортько, К. М. Малаш, "Застосування мемоізації в задачах мінімізації кількості знаків арифметичних операцій," Вісник Національного університету водного господарства та природокористування (Серія: Технічні науки). Рівне: НУВГП, 2020, № 2 (90), С. 127-143. URL: https://ep3.nuwm.edu.ua/19764/.

[2] O. V. Shportko, A. Ya. Bomba, K. M Malash, "Using of memoization in arithmetic operations sign placement problems," Modern problems of mathematical modeling, automates control and information technologies : materials of IIIth International scientific and practical conference (Rivne, 14-16 november, 2019). Rivne: National university of water and environmental engineering, 2019, pp. 202-207.

[3] J. Mayfield et all, "Using Automatic Memoization as a Software Engineering Tool in Real-World AI Systems," Proceedings of the Eleventh IEEE Conference on Artificial Intelligence for Applications (CAIA '95), 1995, pp. 87-93.

[4] A. Umut et all, "Selective Memoization," Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisiana, 2003, pp. 14–25.

[5] P. Norvig, "Techniques for Automatic Memoization with Applications to Context-Free Parsing," Computational Linguistics, vol. 17, 1991, № 1, pp. 91–98.

[6] P. Deutsch, "DEFLATE Compressed Data Format Specification version 1.3," RFC 1951, 1996, Alladin enterprises,- May 1996, 15 p.

[7] О. В. Шпортько, "Оптимізація застосування модифікованого формату словникової компресії Deflate у процесі прогресуючого ієрархічного стиснення зображень без втрат," Науковий вісник Чернівецького національного університету імені Юрія Федьковича (Серія: Комп'ютерні системи та компоненти), 2013, Т. 4, Вип. 4, С. 40-52.

[8] A. Ya. Bomba, A. V. Shportko, V. A. Postolatii, "Redistribution of the Compressed Data Between Modified DEFLATE-Blocks in the Image Compression Process Without Lossless," Computational Linguistics and Intelligent Systems (COLINS 2024) : Proceedings of the 8th International Conference (Lviv, 12-13 Apr 2024). Volume II: Modeling, Optimization, and Controlling in Information and Technology Systems Workshop (MOCITSW), pp. 145-156. URL: https://ceur-ws.org/Vol-3668/paper11.pdf.

[9] О. В. Шпортько, "Пристосування формату словникової компресії DEFLATE до ієрархічного стиснення зображень без втрат," Оброблення сигналів і зображень та розпізнання образів : Праці дванадцятої Всеукраїнської міжнародної конференції УкрОБРАЗ'2014 (3-7 листопада 2014 року). Київ: МННЦ ІТтС, 2014, С. 151-154.

[10] J. Ziv, A. Lempel, "A universal algorithm for sequential data compression," IEEE Transactions on Information Theory, vol. 23(3), 1977, pp 337-343, DOI: 10.1109/TIT.1977.1055714.

[11] C. E Shannon, "A Mathematical Theory of Communication," Bell System Technical Journal, 1948, vol. 27, pp. 379-423, 623-656, DOI: 10.1002/j.1538-7305.1948.tb00917.x.

[12] D. Huffman, "A Method for the Construction of Minimum Redundancy Codes," Proceedings of the IRE, 19526 № 40 (9), pp. 1098-1101, DOI: 10.1109/JRPROC.1952.273898.

[13] J. Rissanen, G. G. Langdon, "Arithmetic coding," IBM Journal of Research and Development, 1979, № 23 (2), pp. 149-162. DOI: 10.1147/rd.232.0149.

[14] *ACT – Test Files*. [Online]. URL: http://compression.great-site.net/act/act-files.html.