Міністерство освіти і науки України

Рівненський державний гуманітарний університет

На правах рукопису

Шпортько Олександр Володимирович

УДК 004.04

ПІДВИЩЕННЯ ЕФЕКТИВНОСТІ СТИСНЕННЯ КОЛЬОРОВИХ ЗОБРАЖЕНЬ У ФОРМАТІ PNG

Спеціальність 01.05.03 – математичне та програмне забезпечення обчислювальних машин і систем

Дисертація на здобуття наукового ступеня кандидата технічних наук

Науковий керівник:
доктор технічних наук, професор
Бомба Андрій Ярославович

Рівне - 2010

З м і с т

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

КС	Коефіцієнт стиснення
РДГУ	Рівненський державний гуманітарний університет
ACT	Archive Comparison Test
ARIC	Arithmetic Coding
HUFF	Huffman
JPEG	Joint Photographic Experts Group
KTCI	Kodak True Color Images
$\log\langle x \rangle$	$\log_2 x$
LPC	Linear Prediction Coding
LS	Loss less
LZ	Ziv, Lempel
PNG	Portable Network Graphics

ВСТУП

Актуальність теми. У сучасному світі зображення є невід'ємною складовою мультимедійної інформації, що найчастіше створюється, накопичується і зберігається на цифрових носіях та передається каналами зв'язку. Компресія відповідних файлів дає змогу пропорційно підвищити швидкість обміну інформацією по мережі та зменшити обсяги використання дискового простору. Тому проблема підвищення ефективності стиснення зображень не втрачає своєї актуальності протягом останніх десятиліть і, ймовірно, не втратить у найближчому майбутньому.

На сьогодні для збереження без втрат космічних знімків, фотореалістичних зображень з невисокою роздільною здатністю чи синтезованих ілюстрацій та емблем, у галузях діяльності людини, де спотворення неприпустимі (наприклад, у медицині [8, с. 641] чи картографії), як один з основних використовується формат PNG [62]. В мережі Інтернет, наприклад, нараховується більше 16 млн. сторінок, що містять зображення у цьому форматі, щороку кількість таких сторінок збільшується на понад 1 млн. Популярності формату PNG сприяють, насамперед, прийнятні показники стиснення та висока швидкість декодування, а саме ці критерії ефективності є визначальними для форматів графічних файлів. І якщо для переважної більшості форматів компресії зображень з втратами (наприклад, для JPEG та ін. [28; 22; 38; 60; 65; 77]) можна забезпечити потрібний КС (в роботі коефіцієнт стиснення – це відношення розмірів стиснутого до нестиснутого файлів зображення [37, с. 21], виражене в процентах) за рахунок погіршення якості, то КС у форматах компресії зображень без втрат, до яких належить і PNG, залежить, власне, лише від перепадів яскравостей кольорів їх пікселів та самого алгоритму стиснення, не регулюється програмно і становить в середньому лише 30 – 70 %. Тому підвищення ефективності стиснення зображень у форматі PNG, враховуючи поширеність та стрімке збільшення їх кількості, є на сьогодні актуальною задачею.

Теоретичною та методологічною основою роботи стали праці таких вчених, як C. Shannon, D. Knuth, D. Huffman, J. Ziv, A. Lempel, W. Pratt, R. Gonzalez, R. Woods,

D. Salomon, J. Miano, T. Boutell, P. Deutsch, Д. Ватолін, О. Ратушняк, М. Смірнов, В. Юкін, а також результати досліджень текстів програм та функціонування існуючого програмного забезпечення для стиснення зображень.

Зв'язок роботи з науковими програмами, планами, темами. Дисертаційна робота виконувалася, керуючись метою та завданнями Національної програми інформатизації (Закон України № 74/98-ВР від 4 лютого 1998 року), у напрямку тематики науково-дослідної роботи "Підвищення ефективності стиснення зображень без втрат у сучасних графічних форматах" (№ державної реєстрації 0110U004001) кафедри інформатики і прикладної математики РДГУ. Автором у цій НДР проведено дослідження можливостей підвищення ефективності стиснення зображень у форматі PNG.

Мета і завдання дисертаційної роботи. *Метою дослідження* є зменшення КС та прискорення найтриваліших етапів компресії зображень як у межах діючого, так і в модифікованому стандарті формату PNG за допомогою вдосконалення і врахування взаємодії використаних та застосування альтернативних чи нових методів і алгоритмів кодування, які кардинально не впливають на час декодування.

Для досягнення поставленої мети у роботі розв'язано такі *завдання*:

1. Дослідження та аналіз ефективності алгоритмів кодування, що використовуються у форматі PNG і програмного забезпечення, яке застосовується для збереження зображень у цьому форматі.

2. Розробка алгоритмів для прискорення виконання найтриваліших етапів стиснення зображень у форматі PNG.

3. Виявлення можливостей та розробка алгоритмів і програмного забезпечення для зменшення КС зображень у діючому стандарті формату PNG.

4. Дослідження можливостей та розробка алгоритму використання декількох ковзаючих вікон для результатів застосування різних предикторів у процесі формування модифікованого розкладу LZ77.

5. Розробка методів для генерування різницевих кольорових моделей, які дають змогу зменшити КС зображень у форматах, що використовують предиктори та контекстно-незалежне кодування.

6. Вивчення можливостей та розробка алгоритму для застосування палітри в процесі стиснення зображень без втрат.

Об'єкт дослідження – процеси стиснення даних без втрат.

Предмет дослідження – методи компресії 24-бітних RGB-зображень [29] у форматі PNG. Означена підмножина зображень обрана невипадково, оскільки такі зображення зустрічаються найчастіше. І це не дивно, адже трикомпонентні зображення найкраще відповідають фізіологічним основам кольорового зору людини [31, с. 37], кольорова модель RGB відображає роботу комп'ютерних дисплеїв [25, с. 16], а 8-ми бітна частота дискретизації найчастіше використовується у найпоширеніших ОС сімейства Windows [25, с. 17]. Колір довільного піксела таких зображень формується злиттям яскравостей червоної, зеленої та синьої компоненти, кожна з яких є рівнозначною і подається цілим числом з діапазону [0; 255].

Методи дослідження – алгоритми стиснення зображень розроблено з застосуванням методів теорій кодування та програмування, прогнозовані оцінки довжин кодів блоків обчислено з використанням положень теорії інформації та теорії ймовірності, вибір варіанту компресії для кожного з мінімальних блоків рядків здійснено методом динамічного програмування, оцінювання параметрів нерівномірностей розподілів частот елементів виконано методами математичної статистики, оцінку складності алгоритмів прискорення стиснення проведено методами теорії алгоритмів.

Наукова новизна одержаних результатів. В процесі виконання дисертаційного дослідження створено нові та вдосконалено існуючі методи і алгоритми, *які дають змогу зменшити КС зображень без втрат*, зокрема:

1. Вперше одержано метод зменшення розміру стиснутого блоку у форматі DEFLATE за допомогою: розрахунку розподілів частот для визначення розмірів альтернативних стиснутих блоків без їх попередньої генерації; вибору найкоротшого блоку з альтернативних; ітеративного відкидання неефективних замін у найкоротшому стиснутому блоці з подальшим його формуванням. В процесі реалізації цього методу вдосконалено (з метою прискорення) метод точного

розрахунку розмірів блоків динамічних кодів HUFF.

2. Вперше розроблено метод попереднього аналізу зображень, в якому реалізовано розбиття на мінімальні та однорідні блоки рядків з метою визначення для кожного з них оптимального способу кодування методом динамічного програмування.

3. Вперше розроблені методи генерування альтернативних різницевих кольорових моделей для кожного зображення як з цілими, так і з дійсними коефіцієнтами, які орієнтовані на формати стиснення зображень без втрат, що використовують предиктори та контекстно-незалежне кодування.

4. Вдосконалено метод словникового стиснення для забезпечення можливості одночасного використання результатів застосування декількох альтернативних предикторів та механізм формування "лінивого" і майже оптимального розкладів цього методу на прикладі алгоритму LZ77 з використанням результатів попереднього аналізу зображень.

5. Вдосконалено метод пошуку однакових послідовностей потоку, що використовує хешування, шляхом вибору найкоротших хеш-ланцюгів, реалізація якого дозволила не лише зменшити КС зображень для швидких розкладів словникового алгоритму LZ77, а й суттєво прискорити формування розкладів цього алгоритму загалом.

6. Отримав подальший розвиток метод групового статистичного кодування за допомогою використання палітри в процесі стиснення трикомпонентних зображень без втрат.

Практична цінність дисертації. Розроблені методи, алгоритми і способи стиснення зображень без втрат використовуються для підвищення ефективності стиснення у форматі PNG та можуть бути використані з цією ж метою у інших графічних форматах, зокрема, у середньому по набору зображень ACT:

1. Застосування алгоритму вибору найкоротших хеш-ланцюгів у процесі пошуку однакових послідовностей як в одному, так і у декількох словниках дає змогу прискорити формування розкладу LZ77 у випадку аналізу всіх елементів цих ланцюгів більше, ніж у 13 разів, а у разі перегляду до 128 їх елементів – на 9 %.

2. Використання розроблених алгоритмів наближених і точних розрахунків довжини блоків динамічних кодів HUFF без їх генерації прискорюють виконання таких обчислень відносно варіанту з формуванням цих кодів на понад 85 %.

3. Реалізація алгоритму мінімізації розміру стиснутих блоків у форматі PNG дозволяє зменшити КС переважної більшості синтезованих з шумами та фотореалістичних зображень на 2 – 6 %, хоча й сповільнює кодування на 10 %.

4. Розроблений метод попереднього аналізу зображень з розбиттям на мінімальні та однорідні блоки рядків з метою визначення для кожного з них оптимального способу кодування за допомогою методу динамічного програмування дозволяє зменшити КС відносно швидких варіантів стиснення на понад 3.9 %, сповільнюючи кодування більш ніж у 4.5 рази.

5. Використання алгоритму LZPR у випадку застосування трьох альтернативних ковзаючих вікон LZ77 та формування попіксельного розкладу дає змогу зменшити КС зображень на понад 2.1 % у порівнянні з найкращим на сьогодні способом стиснення у стандарті формату PNG, прискорюючи при цьому кодування у 4.6 рази, а декодування – на 4.3 %.

6. Застосування описаного методу генерування та переходу до альтернативних різницевих кольорових моделей з цілими коефіцієнтами для кожного зображення у форматі PNG дає змогу зменшити КС на 4.6 % (максимально – на понад 12 %) в основному за рахунок фотореалістичних знімків, хоча й сповільнює кодування більше, ніж на 10 %.

7. Розроблений алгоритм використання палітри для групового статистичного кодування трикомпонентних зображень без втрат дозволяє додатково зменшити їх КС на 1.4 % та прискорити декодування на понад 7 %, хоча й сповільнює кодування на 38 %.

Реалізація результатів дослідження. Розроблені з використанням результатів дисертації утиліта MinPNG для створення зображень у форматі PNG з низькими КС і комплекс програм ModifyPNG для компресії зображень з застосуванням досліджених модифікацій цього формату та їх вихідні тексти доступні для завантаження з Web-сторінки http:\\apserver.org.ua/peregl.php?=5 і

розповсюджуються як безкоштовне сервісне програмне забезпечення. Основні фрагменти коду цих програм захищені двома авторськими свідоцтвами (додаток А). Утиліта MinPNG застосовується як на домашніх ПК, так і в організаціях, зокрема в Рівненській обласній клінічній лікарні, інженерному центрі "Імпульс" та Рівненській міській бізнес-довідці (ТзОВ-фірма "ВІЗА"). Алгоритм вибору найкоротших хеш-ланцюгів використовується в процесі розробки прикладного програмного забезпечення у ВАТ "Алмаз ССС". Відповідні акти впровадження містяться у додатку Б.

Основні результати дисертаційної роботи використовуються у спецкурсі "Проблеми оптимізації і керування процесами", що вивчається студентами спеціальностей "Інформатика" та "Прикладна математика" факультету математики та інформатики, у процесі викладання інших дисциплін в РДГУ (додаток В).

Особистий внесок здобувача. Всі наукові результати, подані в дисертації, отримані здобувачем особисто. У працях, опублікованих у співавторстві, здобувачеві, окрім отримання та аналізу результатів числових експериментів, належать: [4; 5] – ідеї та реалізації запропонованих алгоритмів; [6] – вибір і адаптація модифікацій формату PNG та їх комбінацій.

Апробація результатів дисертації. Результати дослідження та основні положення роботи доповідалися і обговорювалися на: секції математичного моделювання та обчислювальних методів XIX, XX, XXI наукових сесій наукового товариства ім. Шевченка (Рівне, березень 2008-2010 рр.); звітних наукових конференціях РДГУ (Рівне, березень 2008 р., лютий 2009 та 2010 р.); науково-технічній конференції "Обчислювальні методи і системи перетворення інформації" (Львів, жовтень 2010 р.) [6]; X, XI, XII Міжнародних науково-технічних конференціях "Системний аналіз та інформаційні технології" (Київ, травень 2008-2010 рр.) [48; 54; 57]; III, IV Міжнародних конференціях "Комп'ютерні науки та інформаційні технології" (Львів, вересень 2008 р., жовтень 2009 р.) [49; 56]; IX Всеукраїнській міжнародній конференції з оброблення сигналів і зображень та розпізнання образів (Київ, листопад 2008 р.). У повному обсязі результати дослідження доповідалися і обговорювалися на: засіданні наукового семінару секції

інформатики при Західному науковому центрі НАН та МОН України (Львів, травень 2010 р.); засіданні наукового семінару "Образний комп'ютер" при Міжнародному науково-навчальному центрі інформаційних технологій і систем НАН та МОН України (Київ, червень 2010 р.); засіданні наукового семінару кафедри інформаційних систем КНУ ім. Т. Г. Шевченка (Київ, вересень 2010 р.); розширеному засіданні наукового семінару кафедри інформатики та прикладної математики РДГУ (Рівне, жовтень 2010 р.).

Публікації. Основні результати дисертаційного дослідження опубліковані у 20 наукових працях, з них 2 – авторські свідоцтва, 12 статей (в т. ч. 11 – у фахових виданнях з технічних наук, включених до переліку ВАК України), 3 – матеріали доповідей та 3 – тези доповідей на міжнародних конференціях.

РОЗДІЛ 1
ШЛЯХИ ПІДВИЩЕННЯ ЕФЕКТИВНОСТІ СТИСНЕННЯ ЗОБРАЖЕНЬ У ФОРМАТІ PNG. АНАЛІЗ ЛІТЕРАТУРНИХ ДЖЕРЕЛ. ОБГРУНТУВАННЯ ЗАВДАНЬ ДОСЛІДЖЕННЯ

1.1. Послідовність кодування зображень у форматі PNG

Загальновідомо, що будь-яке стиснення даних можливе в основному за рахунок зменшення надлишковості [37, с. 15; 8, с. 598; 13, с. 1]. Чим більше видів надлишковостей виявляє і опрацьовує компресор і чим краще він ці надлишковості усуває – тим ефективніше він зможе стиснути орієнтовані на таку обробку дані. Стиснення зображень у форматах без втрат, до яких належить і формат PNG [62; 25, с. 249-318], найчастіше складається з двох послідовних достатньо незалежних операцій [8, с. 642]: переходу до альтернативного подання зображення (відображення), під час якого зменшується *міжелементна* надлишковість та поелементного кодування отриманих даних для ліквідації *кодової* надлишковості.

Формат графічних файлів PNG був створений 1 жовтня 1996 року для ефективного збереження растрових зображень без втрат після того, як компанія Unisys почала вимагати плату за використання формату GIF [25, с. 250]. У форматі PNG з метою забезпечення мінімальних КС передбачено три способи кодування: для зменшення міжелементної надлишковості послідовно застосовуються предиктори [7, с. 2-7; 24, с. 54-66] та контекстно-залежний словниковий алгоритм LZ77 [78; 24, с. 78-82; 37, с. 84-88; 25, с. 284-285; 14, с. 98-100; 19; 21], а після цього, для ліквідації кодової надлишковості, використовується контекстно-незалежний канонічний алгоритм HUFF [66; 24, с. 31-34; 25, с. 91-103; 40, с. 52-54; 17]. Для прискорення компресії зображень чи їх фрагментів перші два з цих способів кодування (або один з них) можуть і не застосовуватися, хоча це й збільшить КС, відмова ж від кодування HUFF унеможливлює використання алгоритму LZ77 та робить недоцільним кодування предикторами [25, с. 294-295; 5], тому для забезпечення стиснення цей алгоритм застосовується обов'язково. Компоненти пікселів RGB-зображень, отримані в результаті прямокутної дискретизації [41, с. 6],

записуються у PNG-файли (після опрацювання предикторами та можливих перестановок по рядках) зверху вниз, а у кожному рядку – послідовно зліва направо (як символи у текстах), формуючи тим самим вхідний потік. Байтам яскравостей компонентів пікселів кожного рядка у вхідному потоці передує байт, що визначає номер предиктора, який до них застосовувався [25, с. 300-306, 315-317]. У PNG-файлах, що використовуються на сьогодні, байти вхідного потоку кодуються у відокремлених стиснутих блоках відповідно до формату словникового стиснення DEFLATE [24, с. 94-105; 25, с. 286-292; 27] з універсальної бібліотеки стиснення ZLIB [64], який регламентує застосування алгоритмів LZ77 та HUFF.

Для дослідження можливостей підвищення ефективності стиснення RGB-зображень у форматі PNG розглянемо детальніше принципи трьох його способів кодування та визначимо напрямки можливих модифікацій цього формату.

1.1.1. Словниковий алгоритм LZ77 та основні варіанти формування його розкладу

1.1.1.1. "Жадібний" розклад алгоритму LZ77. Розглянемо спочатку детальніше механізм стиснення словникового алгоритму LZ77 на основі найуживанішого "жадібного" розкладу вхідного потоку (детальний опис цього та інших альтернативних розкладів наведено в [24, с. 75-119]). Описуючи словникові алгоритми, фіксовану кількість попередніх закодованих елементів вхідного потоку називають *словником*, а наступних незакодованих – *буфером*. Нехай послідовність елементів потоку $s_1, s_2, \ldots, s_{j-1}$ вже закодована і занесена в словник. Тоді під час чергової ітерації алгоритму для послідовності незакодованих елементів буфера $s_j, s_{j+1}, \ldots$ шукають однакову послідовність $s_i, s_{i+1}, \ldots$ *максимальної* довжини $len(j)$ (саме тому такий розклад називають "жадібним"), яка бере початок у словнику послідовності ($i<j$). У разі виявлення однакової послідовності $s_j, s_{j+1}, \ldots, s_{j+len(j)-1}$ її кодують парою чисел $<$*довжина*; *зміщення від закінчення словника*$>$, тобто $<len(j); j-i>$, послідовність $s_j, s_{j+1}, \ldots, s_{j+len(j)-1}$ заносять в словник та виконують перехід до кодування потоку, починаючи з елемента $s_{j+len(j)}$. Якщо ж однакова послідовність у словнику відсутня, то елемент s_j заносять у закодовані дані (де його називають *літералом*) та словник без змін і виконують перехід до кодування потоку,

починаючи з наступного елемента s_{j+1} (саме тому відмову від кодування алгоритмом LZ77 надалі ототожнимо з застосуванням розкладу LZ77 без використання замін парами чисел <*довжина*; *зміщення*>). Кодування припиняють після опрацювання останнього елемента потоку. Максимальні розміри словника та буфера встановлюються конкретними реалізаціями алгоритму (наприклад, у нашому випадку формат DEFLATE обмежує словник 32768 елементами, а буфер – 258 елементами). Оскільки однакові послідовності кодуються парою чисел <*довжина*; *зміщення*>, то для досягнення стиснення їх довжина має перевищувати 2. Однакові послідовності максимальної довжини шукають у словнику з кінця справа наліво і запам'ятовують найменші зміщення до них [25, с. 308-311], оскільки однакові фрагменти зображень чи фрагменти однакової структури найчастіше трапляються недалеко. Якщо позначити через *offset*(*j*; *k*) найменше зміщення від позиції *j* до однакової послідовності у словнику, довжина якої *k*, то, згідно з викладеним вище, $i = j - offset(j; len(j))$. Коротші однакові послідовності від послідовності максимальної довжини можуть зустрічатися у словнику і правіше [50], тому для $3 \le k < len(j)$

$$offset(j;k) \le offset(j;len(j)). \qquad (1.1)$$

Сукупність словника з закодованими елементами та буфера з незакодованими ще називають *ковзаючим вікном* [24, с. 78], оскільки вони весь час синхронно переміщуються по елементах потоку.

Приклад 1.1. Потік значень байтів "2, 4, 1, 2, 2, 4, 1, 3, 2, 4, 1, 2, 4, 1, 3" (за умови, що далі в потоці йде значення, яке до цього не зустрічалося) в закодованому вигляді згідно "жадібного" розкладу LZ77 записується як "2, 4, 1, 2, <3; 4>, 3, <4; 8>, <3; 7>".■

Під час декодування окремі елементи копіюють у вихідний потік та буфер без змін. Пари ж <*довжина*; *зміщення*> декодують шляхом послідовного копіювання з кінця словника за вказаним зміщенням в початок буфера та вихідний потік вказаної кількості елементів. Природно, що алгоритм декодування кодів LZ77 має розрізняти окремі елементи та групи <*довжина; зміщення*>. У форматі словникового

стиснення DEFLATE, на якому базується формат графічних файлів PNG, з цією метою довжини замін та окремі елементи кодуються разом [25, с. 283-290; 24, с. 94-99] (ця модифікація алгоритму LZ77 має назву LZH [24, с. 90-91]) числами в межах [0; 285]. При цьому числа з діапазону [0; 255] відповідають кодам окремих елементів, 256 позначає закінчення стиснутого 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 аналогічно – у вигляді базового значення та додаткових бітів. Базове значення зміщення знаходиться в межах [0; 29]. Зрозуміло, що коротші однакові послідовності зустрічаються в зображеннях частіше від довших, оскільки у різних їх частинах найчастіше повторюються невеличкі фрагменти яскравостей чи структури, а коротші зміщення трапляються частіше від довших у відповідності з алгоритмом їх генерації. Саме тому для зберігання довжин та зміщень у форматі DEFLATE використовують модифіковані групові коди [14, с. 36-37], які більшим значенням ставлять у відповідність неменшу кількість додаткових бітів.

Розглянутий "жадібний" розклад алгоритму LZ77 забезпечує максимальну швидкість кодування, але не мінімальні КС [24, с. 106]. Ми використаємо цей розклад під час попереднього аналізу та для найшвидшого стиснення зображень.

Оскільки однакових послідовностей різної довжини у словнику може виявитися багато, то й альтернативних розкладів LZ77 теж, як привило, існує чимало (наприклад, для потоку з прикладу 1.1 можливий також розклад "2, 4, 1, 2, <3; 4>, 3, <3; 4>, <4; 7>"). Природно, що хотілося б мати стратегію розкладу, яка забезпечує кращі КС і незначно поступається по часу кодування "жадібному" розкладу та *оптимальну* стратегію розкладу, яка б *завжди* забезпечувала мінімальну довжину закодованої послідовності [24, с. 108]. Саме тому на сьогодні використовується декілька альтернативних варіантів формування розкладу LZ77 [24, с. 90-94; 14, с. 97-116]. Але результати кодування LZ77 не використовуються самостійно а, як правило, кодуються контекстно-незалежним алгоритмом, тому вартість кодування окремих елементів та замін змінюється для різних вхідних

послідовностей. Враховуючи цей факт, в [70] дану задачу було визнано *NP*-повною, тобто такою, що вимагає перебору всіх можливих варіантів для знаходження оптимального розв'язку. Тому у наступних підпунктах охарактеризуємо інші розклади LZ77, що найчастіше використовуються у роботі. Насамперед розглянемо той розклад серед відомих на сьогодні варіантів розкладів [24, с. 108], який забезпечує мінімальні КС. Власний алгоритм для зменшення КС розкладів LZ77, що не виконують пошук однакових послідовностей зі всіх позицій потоку, запропонований нами у пункті 3.4.1.

Звичайно, замість алгоритму LZH, що належить до сімейства LZ77, у форматі PNG можна було б спробувати застосувати будь-який інший алгоритм, з альтернативного сімейства словникових алгоритмів LZ78 [79; 20; 74] (ці алгоритми базуються на заміні послідовності чергових елементів потоку індексом аналогічної послідовності у словнику, яка формується, як правило, з попередніх елементів потоку). Але алгоритми сімейства LZ78 в середньому забезпечують значно повільніше декодування та неменший КС від алгоритмів сімейства LZ77 [24, с. 89-90], тому вони і не застосовувалися під час дослідження.

1.1.1.2. Майже оптимальний розклад алгоритму LZ77. Алгоритм цього розкладу був розроблений для зменшення довжини закодованої послідовності [24, с. 108]. Ідея цього алгоритму полягає в обчисленні мінімальної вартості (кількості бітів) $V(j)$ кодування потоку від початку до кожного чергового елемента j включно. Згідно даного алгоритму,

$$V(j) = \min_{l<j} (V(l) + price(l+1, j)), \tag{1.2}$$

де $price(l+1, j)$ – мінімальна вартість зберігання пари <*довжина*; *зміщення*> для кодування елементів з $l+1$ по j-й, якщо $l<j-1$ або вартість кодування елемента s_j, якщо $l=j-1$ [44]. Після визначення мінімальної вартості кодування до останнього елемента потоку включно відбирають позиції та зміщення, що дають змогу досягнути цієї мінімальної вартості, і саме з них формують закодований потік.

Для зображень з високим рівнем міжелементної надлишковості майже оптимальний розклад LZ77 дає змогу зменшити КС відносно "жадібного" розкладу

на понад 10 % [24, с. 106] і досягає по цьому показнику найкращих результатів серед відомих варіантів розкладів LZ77. Основним недоліком цього розкладу є низькі швидкості формування вихідного потоку його реалізаціями, що в декілька разів поступаються швидкостям реалізацій жадібного розкладу [50], адже майже оптимальний розклад виконує пошук однакових послідовностей з *кожної*, а не з окремих позицій потоку. Крім цього, реалізації даного розкладу найчастіше виконують ще й додатковий прохід по даних вхідного потоку для визначення прогнозованих вартостей кодувань окремих елементів та замін [58] (звичайно, обчислювати вартості кодувань можна для наступних з попередніх блоків, але такий підхід збільшує КС [44]). Ось чому на практиці найчастіше використовуються варіанти, які забезпечують краще стиснення і незначно поступаються по часу кодування "жадібному" розкладу, тобто виконують пошук однакової послідовності не для всіх позицій потоку [24, с. 110]. Ми використаємо майже оптимальний розклад для стиснення зображень з мінімальними КС у випадках, коли обмеження по часу не накладаються.

1.1.1.3. Розклад алгоритму LZ77 з "лінивими" порівняннями. Відійти від "жадібного" розкладу потоку найчастіше намагаються за рахунок використання елементів замість фрагментів замін там, де це доцільно [50, 58] (як, наприклад, на рис. 1.1), адже окремі елементи кодуються меншою кількістю бітів, ніж пари <*довжина*; *зміщення*>.

Рис. 1.1. Приклад підвищення ефективності "жадібного" розкладу

Для такого вдосконалення, як правило, використовують розклад послідовності LZ77 з "лінивими" порівняннями [24, с. 104-105]. Згідно алгоритму цього розкладу, у

випадку виявлення однакової послідовності з позиції *j* виконують пошук однакової послідовності також і з позиції *j*+1. Якщо *len(j+1)>len(j)* (див. рис. 1.1), то в позиції *j* кодують елемент і переходять до кодування з позиції *j*+1, інакше кодують віднайдену з цієї позиції заміну і переходять до позиції *j+len(j)*. Основними недоліками такого розкладу є можливість генерування послідовності декількох елементів замість однієї заміни [24, с. 105] та ймовірне виконання пошуку з позиції *j*+2, яке рідко покращує результати розкладу [14, с. 108].

Класичний розклад з "лінивими" порівняннями не враховує того факту, що різні елементи та заміни можуть кодуватися різною кількістю бітів. Тому ми у підрозділі 3.4 опишемо і надалі використаємо модифікований розклад з "лінивими" порівняннями, який відмовляється від кодування заміни з позиції *j* лише тоді, коли кодування літералу з цієї та заміни з наступної позиції забезпечують менший прогнозований КС.

Класичний та модифікований "ліниві" розклади генерують коротші кодові послідовності від "жадібного" розкладу згідно алгоритму їх формування, хоча й вимагають при цьому виконання додаткових пошуків однакових послідовностей, що призводить до збільшення загального часу кодування [14, с. 108].

1.1.1.4. Організація пошуку однакових послідовностей за допомогою хешування. Довжина коду вихідної послідовності, отриманого згідно алгоритму LZ77, зменшується у випадку виявлення довших однакових послідовностей. Тому, як правило, чим довшу однакову послідовність елементів для послідовності чергових елементів буфера вдається віднайти під час кодування серед всіх можливих послідовностей, що починаються у словнику, тим ефективнішим виявляється використання цього алгоритму. На пошук таких послідовностей припадає основна частина часу кодування як вхідного потоку алгоритмом LZ77 [47, 56], так і зображень у форматі PNG загалом [25, с. 306].

Відшукання таких послідовностей для елементів буфера шляхом повного перебору, починаючи з кожної позиції словника, призводить максимум до *N*M* порівнянь послідовностей, де *N* – кількість елементів потоку, а *M* – кількість елементів словника. Оскільки розміри сучасних файлів даних можуть сягати

десятків Гб ($N=O(10^{10})$), а розміри словника – десятків Кб ($M=O(10^4)$), то на практиці пошук однакових послідовностей шляхом повного перебору не використовується.

Досягнути значно вищої швидкості пошуку найдовшої однакової послідовності дозволяє використання ключів. Під *ключем* для словникових алгоритмів найчастіше розуміють послідовність елементів буфера найменшої довжини, яку доцільно замінювати посиланням на закодовані раніше дані [25, с. 307]. Оскільки замінювати чергову послідовність елементів буфера парою чисел <*довжина; зміщення*> доцільно лише тоді, коли її довжина перевищує 2, то ключем для алгоритму LZ77 є послідовність з трьох елементів (у нашому випадку – з трьох байтів). Очевидно, що необхідною умовою результативного пошуку послідовності є наявність ключа початку буфера серед ключів, що починаються в словнику. Тому, на перший погляд, для прискорення пошуку доцільно створити ланцюги вказівників на однакові ключі, що починаються в словнику та таблицю вказівників на початки відповідних ланцюгів для кожного з можливих значень ключів. У цьому випадку, виконуючи пошук найдовшої однакової послідовності, достатньо переглянути лише послідовності, що починаються у словнику з ключа початку буфера. Наприклад, у процесі стиснення вже згаданого потоку "2, 4, 1, 2, 2, 4, 1, 3, 2, 4, 1, 2, 4, 1, 3", шукаючи найдовшу однакову послідовність для буфера "2, 4, 1, 3", у словнику "2, 4, 1, 2, 2, 4, 1, 3, 2, 4, 1" достатньо переглянути лише послідовності, які починаються з "2, 4, 1" (рис. 1.2), що значно прискорить виконання цієї операції відносно варіанту повного перебору.

Але якщо під час реалізації такого варіанту пошуку найдовшої однакової послідовності кількість елементів усіх ланцюгів дорівнює кількості елементів словника, то таблиця з вказівниками на початки ланцюгів має містити елементи для всіх можливих значень ключів. Наприклад, якщо елементами потоку є байти, то для алгоритму LZ77, враховуючи триелементність ключів, кількість елементів такої таблиці становитиме $(2^8)^3$, тобто понад 16 млн.

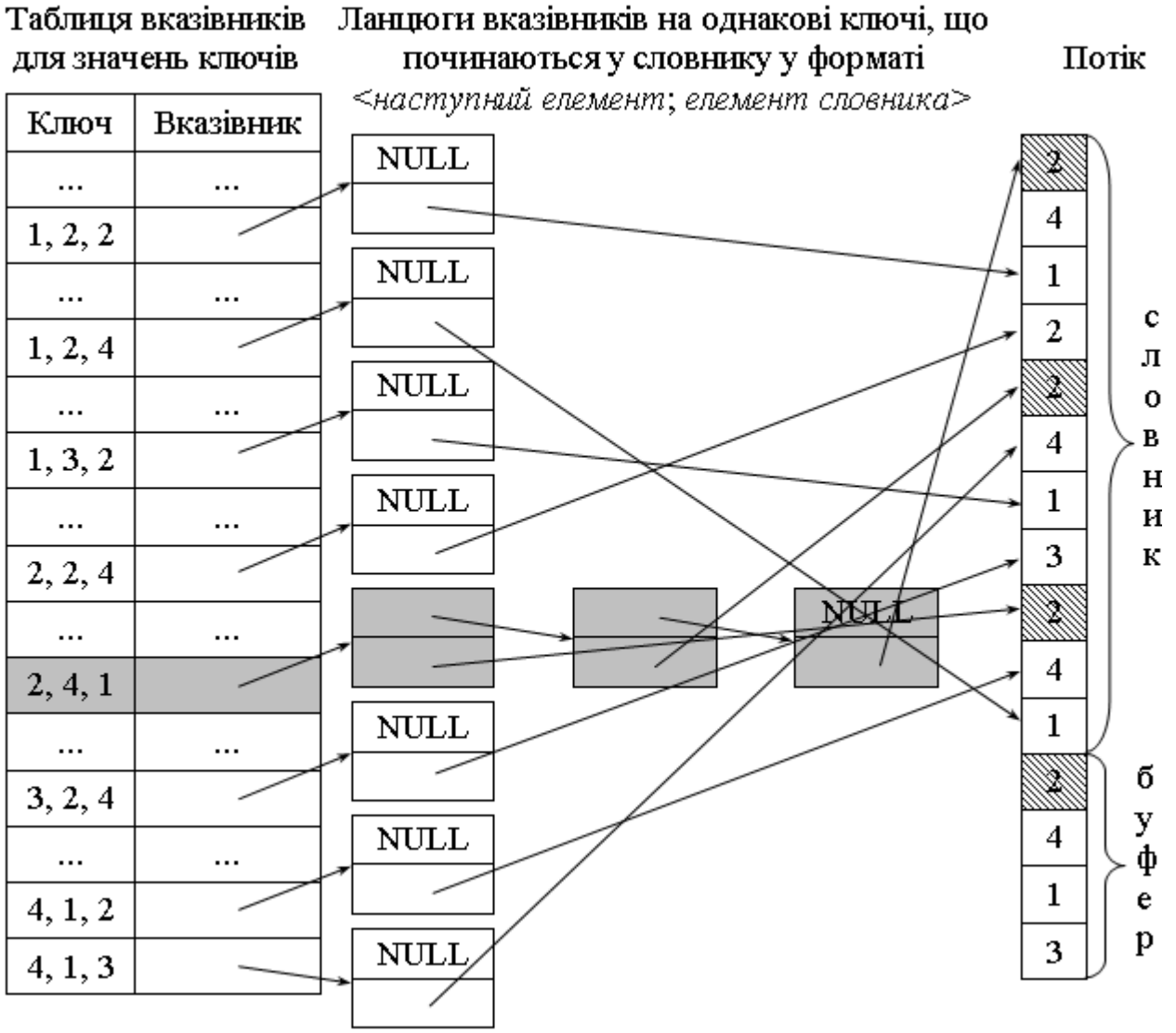


Рис. 1.2. Елемент таблиці вказівників та ланцюг вказівників (виділені сірим кольором) на ключі "2, 4, 1" (початки заштриховані) словника "2, 4, 1, 2, 2, 4, 1, 3, 2, 4, 1" перед кодуванням буфера "2, 4, 1, 3"

Очевидно, що створювати та опрацьовувати таку величезну таблицю недоцільно, а подекуди і неможливо через значні витрати пам'яті і сповільнене звертання до елементів. Тому на практиці замість таблиць з індексами всеможливих значень ключів використовують хеш-таблиці з індексами зі значно менших фіксованих діапазонів [15, с. 549-596]. Хеш-значення індексів такої таблиці генеруються для значень ключів за допомогою хеш-функції, яка, власне, і виконує їх змішування [24, с. 103; 25, с. 307]. Природно, що хеш-таблиця містить вказівки на початки відповідних хеш-ланцюгів, які, в свою чергу, містять вказівники на ключі

словника з однаковими хеш-значеннями.

Хеш-функції, що використовуються на практиці, як правило, намагаються розподілити можливі значення ключів рівномірно по діапазону індексів хеш-таблиці [25; с. 308]. Наприклад, в програмі з CD до [25], яку ми модифікували в процесі дослідження, хеш-функція генерує хеш-значення шляхом зчеплення п'яти молодших бітів кожного байта ключа, формуючи тим самим діапазон індексів хеш-таблиці [0; 2^{15}-1]. Але очевидно, що хеш-функція, виконуючи відображення (змішування) декількох можливих значень ключів в один індекс хеш-таблиці, тим самим поєднує елементи відповідних ланцюгів в один хеш-ланцюг. Тобто формування коротших хеш-таблиць призводить до виникнення довших хеш-ланцюгів. Крім того, однорідні дані потоку можуть породжувати як завгодно довгі хеш-ланцюги [25, с. 308]. Тому на сьогодні проблема прискорення пошуку однакової послідовності у словнику повністю **не вирішена**. На практиці для досягнення прийнятної швидкості виконання нові елементи дописують на початок хеш-ланцюгів і під час пошуку найдовшої однакової послідовності переглядають обмежену кількість їх перших елементів (відсікають хеш-ланцюги) [24, с. 104; 25, с. 310]. Це, звичайно, значно прискорює пошук однакової послідовності, але погіршує КС, оскільки у словнику не розглядаються послідовності, що починаються з ключів відкинутих елементів хеш-ланцюгів. Ми опишемо альтернативний підхід до розв'язання цієї проблеми у підрозділі 2.1.

1.1.2. Кодування HUFF та його альтернативи. У форматі PNG потік дискретних значень яскравостей компонентів пікселів зображення (після застосування предикторів та можливого використання алгоритму LZ77) розбивається на суміжні вхідні блоки, кожен з яких відображається у відповідний стиснутий блок одного з трьох типів: без опрацювання алгоритмом LZ77 та кодами HUFF (аналогічний вхідному); з використанням фіксованих кодів HUFF; з застосуванням динамічних кодів HUFF [62]. Мінімальні КС вхідних блоків у форматі PNG забезпечують стиснуті блоки третього типу [25, с. 295], використанням яких ми й обмежимося надалі. Тому для подальшого виявлення можливостей підвищення ефективності стиснення зображень у форматі PNG

розглянемо принципи і алгоритм формування динамічних кодів HUFF.

Контекстно-незалежне статистичне кодування HUFF [66] використовується для зменшення кодової надлишковості, яка, як правило, присутня при довільному рівномірному двійковому кодуванні елементів зображень [8, с. 642]. Ідея використання цих кодів полягає у заміні елементів з більшою частотою кодами не більшої довжини (як правило, меншої), ніж для елементів з меншою частотою [8, с. 645-646]. Кодування HUFF ставить у відповідність кожному елементу залежно від його частоти появи (ймовірності) фіксований префіксний код [24, с. 33-34; 37, с. 30-32; 42, с. 157-159; 23, с. 392-394; 32, с. 777-778]. Генерують ці коди за наступним алгоритмом:

1. Підраховують частоти окремих елементів.

2. Впорядковують елементи за спаданням частот.

3. Якщо наявна лише одна ненульова частота, то присвоюють відповідному елементу код "0" і завершують виконання алгоритму. Інакше ітеративно поєднують та вилучають до отримання одного елемента два елементи з найменшими ненульовими частотами (останні в утвореному списку), сумують їх частоти для обчислення частоти утвореного елемента та вставляють його у відсортований список частот. При цьому вхідним елементам, що утворили перший поєднуваний елемент дописують спереду код "0", а тим, що утворили другий – код "1".

У форматі DEFLATE літерали/базові значення довжин (надалі скорочено – літерали/довжини) та базові значення зміщень (надалі скорочено – зміщення), що утворюються після виконання алгоритму LZ77, кодуються різними кодами HUFF [63]. Це пов'язано як з неоднаковими діапазонами значень елементів так і з різними характерами нерівномірностей розподілів їх частот [44]. Приклад кодів HUFF, сформованих згідно описаного вище алгоритму, для елементів послідовності літералів/довжин "2, 4, 1, 2, <3>" (у кутових дужках вказуються довжини замін), отриманої у результаті виконання алгоритму LZ77 (див. прикл. 1.1) наведено в табл. 1.1 . Етапи поєднання елементів цієї послідовності наведено на рис. 1.3 (поєднувані елементи виведені на сірому фоні).

Таблиця 1.1

Характеристики елементів послідовності літералів/довжин "2, 4, 1, 2, <3>"

Елемент, s_i	**Значення**	**Частота**	**Код HUFF**	***p*(s_i)**	**-log *p*(s_i)**
s_1	1	1	01	0.2	2.322
s_2	2	2	1	0.4	1.322
s_3	4	1	000	0.2	2.322
s_4	<3>	1	001	0.2	2.322

Етап 1

Елемент	**Частота**	**Дописаний код**
s_2	2	
s_1	1	
s_3	1	0
s_4	1	1

Етап 2

Елемент	**Частота**	**Дописаний код**
s_2	2	
$s_{3,4}$	2	0
s_1	1	1

Етап 3

Елемент	**Частота**	**Дописаний код**
$s_{3,4,1}$	3	0
s_2	2	1

Рис. 1.3. Етапи поєднання елементів послідовності літералів/довжин "2, 4, 1, 2, <3>" в процесі формування кодів HUFF

Згідно з фундаментальним положенням теорії інформації, для мінімізації довжини коду послідовності, елемент s_i з ймовірністю появи $p(s_i)$ доцільно кодувати $-\log p(s_i)$ бітами [24, с. 17; 8, с. 619-620]. Тому середня довжина коду елемента блоку після застосування будь-якого контекстно-незалежного алгоритму (у тому числі, і кодування HUFF), згідно з формулою of Shannon [69; 8, с. 611; 25, с. 25-26], не може бути меншою *ентропії джерела*

$$H = -\sum_i p(s_i) \times \log p(s_i). \qquad (1.3)$$

Ентропія джерела зменшується зі збільшенням нерівномірності розподілу ймовірностей між елементами [7] та для багатьох зображень тісно корелюється з контрастом [30; 9-11]. Середня довжина коду HUFF для різних блоків незначно (в межах біта) перевищує ентропію і досягає цього значення лише у випадку, коли всі $-\log p(\xi_i)$ – цілі числа [24, с. 35; 8, с. 642-644]. Наприклад, для розглянутої послідовності "2, 4, 1, 2, <3>" середня довжина коду HUFF, наведеного у табл. 1.1, становить 2 біти, а ентропія – 1.922 біта. Зрозуміло, що ймовірності кожного елемента у різних блоках можуть відрізнятися між собою, що призводить до формування для них різних кодів HUFF. З метою генерації цих кодів для елементів послідовностей у форматі PNG на початку кожного стиснутого блоку третього типу міститься заголовок з описом їх довжин [25, с. 295-297], який може займати, за нашими підрахунками, від 37 до 1324 бітів (в середньому – 1000 бітів). Саме кодування HUFF не збільшує розміри вхідних блоків у найгіршому випадку, тому навіть для зображень з випадковими значеннями яскравостей компонентів пікселів розмір кожного стиснутого блоку може перевищити розмір відповідного вхідного блоку лише на розмір його заголовка. Разом з тим, середній КС кодування HUFF становить 67 % [24, с. 34], що й вказує на його ефективність.

Вдосконалити алгоритм генерування кодів HUFF не видається можливим, оскільки у випадку незалежного кодування елементів джерела даних ці коди забезпечують найменшу середню довжину кодів елементів блоку серед відомих нерівномірних кодів [8, с. 642-647]. Але під час стиснення зображень у форматі PNG доцільно: розбивати зображення на блоки рядків (див. пункт 3.3.1) чи результати розкладу алгоритму LZ77 – на окремі фрагменти (див. пункт 3.3.2) з метою підвищення нерівномірності розподілу ймовірностей між елементами; враховувати чи відкидати окремі заміни алгоритму LZ77, впливаючи тим самим на розподіли частот літералів/довжин та зміщень і, відповідно, на довжини кодів HUFF різних елементів. Цю ідею застосовано у підрозділі 3.1 під час опису алгоритму мінімізації розміру стиснутого блоку, який враховує не лише точні розміри блоків кодів HUFF (див. алгоритм підрозділу 2.2), а й додаткові біти.

Крім цього, згідно з першою теоремою of Shannon підвищити ефективність

кодування HUFF можна, використовуючи *n*-кратне розширення джерела (тобто, кодуючи не окремі елементи, а їх суміжні пари, трійки чи групи іншої розмірності), хоча й таке кодування складно реалізувати [8, с. 629, 638-639]. Ми використаємо неявне трикратне розширення джерела, виконуючи групове кодування яскравостей компонентів пікселів під час формування палітри (див. підрозділ 4.3).

Основною альтернативою кодуванню HUFF серед контекстно-незалежних алгоритмів на сьогодні є арифметичне кодування (ARIC) [68; 75; 18; 39, с. 36-40; 16, с. 25-28, 33-34; 59, с. 50-54]. Використання цього кодування не передбачається стандартом DEFLATE, а його застосування зменшує КС для зображень набору ACT в середньому лише на 0.07 % (додаток Д), тому аналіз ефективності альтернативного застосування ARIC винесено за межі основної частини роботи.

1.1.3. Предиктори формату PNG та їх альтернативи. Зменшити ентропію джерела під час обробки зображень найчастіше намагаються за допомогою предикторів [7; 24, с. 54-66]. *Предиктор* – це функція, що прогнозує (моделює) значення чергового елемента, використовуючи значення відомих суміжних елементів. У процесі використання даної технології обчислюють і надалі кодують *відхилення* чергового елемента від прогнозованого предиктором значення. У загальному випадку процес застосування предиктора для кожного елемента у вузлі *(i,j)* записують формулою

$$\Delta_{ij} = F_{ij} - predict_{ij}, \tag{1.4}$$

де F_{ij} – значення елемента до застосування предиктора, Δ_{ij} – значення елемента після застосування предиктора, $predict_{ij}$ – значення предиктора для обраного елемента. Для RGB-зображень з точністю дискретизації 8 бітів предиктори, згідно з (1.4), застосовують до значень кожної компоненти окремо. Тому надалі під значенням елемента для таких зображень будемо розуміти значення окремої компоненти.

Як правило, предиктори використовують значення найближчих суміжних пікселів [24, с. 63], оскільки вони мають найбільшу степінь кореляції з черговим елементом [32, с. 675; 43, с. 318]. Суміжні ж пікселі зображень найчастіше мають

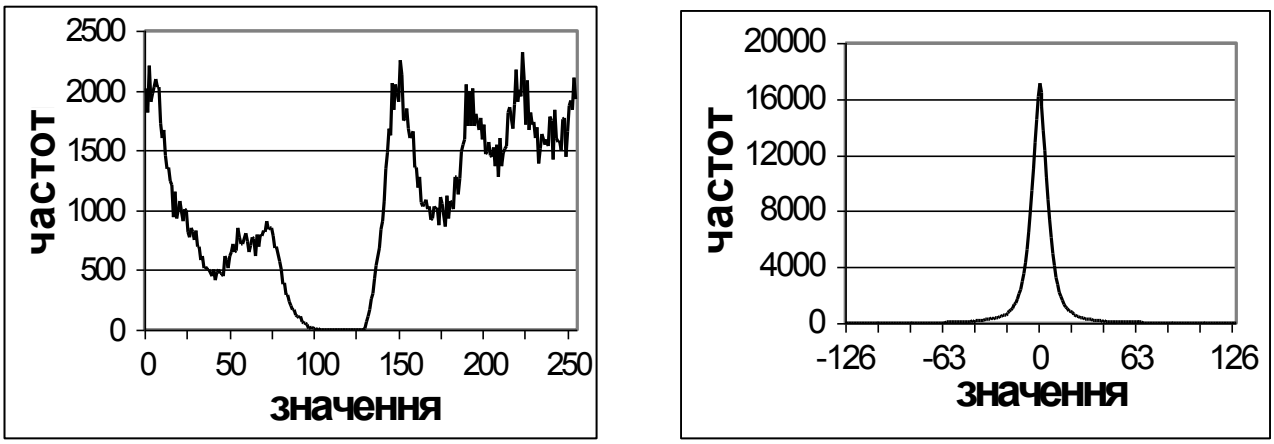
близькі кольори, а значить і близькі значення відповідних елементів, тому значення прогнозу часто буде збігатися зі значенням чергового елемента, найчастіше – буде близьким до цього значення і рідко – значно відрізнятиметься від нього [7]. Тобто більшість значень Δ_{ij} будуть близькими до 0 (як на рис. 1.4б).

а) б)

Рис. 1.4. Відхилення яскравостей суміжних пікселів зеленої компоненти зображення *Lena.bmp* до застосування предиктора (а) та після застосування *Left*-предиктора (б)

Такий перерозподіл частот значень значно підвищує нерівномірність розподілу їх ймовірностей і тому зменшує ентропію джерела (згідно з (1.3)), а отже, і довжину закодованої послідовності. Фактично, предиктори, виконуючи відображення, зменшують міжелементну надлишковість, збільшуючи при цьому кодову надлишковість, що підвищує ефективність контекстно-незалежного алгоритму кодування. Приклад перерозподілу частот значень компонентів після застосування предиктора наведено на рис. 1.5.

Як зазначено в [37, с. 122-123], "експерименти з великими числами показують, що значення Δ прагне мати розподіл, близький до розподілу Лапласа". Цей розподіл, який має нульове математичне сподівання, часто використовують для моделювання густини похибок передбачень [8, с. 667]. Ми використаємо розподіл Лапласа з цією метою у підрозділі 4.2.



а) б)

Рис. 1.5. Розподіл частот значень зеленої компоненти зображення *Lena.bmp* до застосування предиктора (а) та після застосування *Left*-предиктора (б)

Розробляючи предиктори, враховують, що, "як правило, кожен елемент сигналу відхиляється від свого передбачуваного значення не тільки через "сильні" обумовлені зміни – еволюції, але й через "слабкі" фонові коливання, тобто шуми. Тому можливі два протилежні типи моделей: внесок шуму незначний у порівнянні з внеском еволюції; внесок еволюції незначний у порівнянні з внеском шуму. В першому випадку ... будемо передбачувати значення ..., виходячи з лінійної тенденції, що склалася, у другому – як рівне середньому арифметичному ... попередніх елементів" [24, с. 59].

Для однозначності декодування предиктори не можуть використовувати значення, що обходяться після чергового елемента. Оскільки елементи зображення найчастіше обходять по рядках зверху вниз, а в кожному рядку – зліва направо, то предиктори, як правило, не використовують значення правіше та нижче від чергового елемента. Крім цього, застосування для прогнозування більше трьох значень найближчих суміжних елементів не дає відчутного зменшення відхилень прогнозу [32, с. 675]. Тому на практиці, у більшості випадків, предиктори розраховуються за допомогою не більше трьох значень найближчих суміжних елементів.

Позначимо ці значення суміжних елементів (тобто значення аналогічної компоненти суміжних пікселів) для елемента *X* згідно зі схемою рис. 1.6.

LeftAbove	*Above*
Left	*X*

Рис. 1.6. Схема розміщення суміжних елементів для елемента *X*

Тоді предиктори (для $predict_{ij}$ з (1.4)) формату PNG мовою C записують так:

```
char NonePredict(char Left, char Above, char LeftAbove) {return 0; }
char LeftPredict(char Left, char Above, char LeftAbove) {return Left; }
char AbovePredict(char Left, char Above, char LeftAbove) {return Above; }
char AveragePredict(char Left, char Above, char LeftAbove)
 {return (Left+Above)/2; }
char PaethPredict(char Left, char Above, char LeftAbove)
 {int pa, pb, pc, pp=Left+Above-LeftAbove;
  pa=abs(pp-Left); pb=abs(pp-Above); pc=abs(pp-LeftAbove);
  if (pa<=pb && pa<=pc) return Left;
  else if (pb<=pc) return Above;
       else return LeftAbove; }
```

Формати графічних файлів насамперед мають забезпечувати високу швидкість декодування, тому в них найчастіше застосовуються статичні предиктори, коефіцієнти яких не змінюються в процесі декодування (див., наприклад, [7; 37, с. 200]). У форматі PNG, предиктори якого теж є статичними, використовується однаковий предиктор для компонентів всіх пікселів кожного рядка. Номер цього предиктора (від 0 до 4 для кожної з наведених функцій за порядком) зберігається у вхідному потоці алгоритму LZ77 в окремому байті перед байтами відповідного рядка [25, с. 300-303, 317-317].

Ми пропонуємо, по-перше, дослідити можливості підвищення ефективності стиснення за допомогою застосування різних предикторів до різних фрагментів рядків (див. підрозділ 4.1) і, по-друге, розширити стандарт PNG (див. пункт 3.2.1) за допомогою введення в нього додаткового предиктора *MedPredict*, який використовується у стандарті JPEG-LS [71; 72; 35] та вперше був запропонований у [73]. Цей предиктор мовою C записують так:

```
char MedPredict(char Left, char Above, char LeftAbove)
 {if (LeftAbove>=max(Left, Above)) return min(Left, Above);
  else if (LeftAbove<=min(Left, Above)) return max(Left, Above);
       else return Left+Above-LeftAbove; }
```

Розглянемо механізм дії наведених статичних предикторів. Предиктор *NonePredict* використовується в ситуаціях, коли передбачення фактично не

застосовуються і доводиться відмовлятися від кодування предикторами (наприклад, для забезпечення максимальної швидкості кодування або коли застосування предикторів погіршує КС). Предиктор *LeftPredict* прогнозує значення чергового елемента рівним значенню зліва, *AbovePredict* – рівним значенню зверху, *AveragePredict* – середньому арифметичному цих значень. Ці три предиктори описують шумову модель і належать до лінійних предикторів [24, с. 65]. Предиктор Піфа *PaethPredict* розраховує значення у точці *X*, виходячи з площини, що проходить через точки *Left*, *Above* та *LeftAbove* у тривимірному просторі і прогнозує одне з цих трьох значень у напрямку найменшого приросту стосовно розрахованого значення. Цей предиктор використовується при стисненні найчастіше, зокрема в архіваторі RAR [4]. Предиктор *MedPredict* намагається адаптуватися до локальних горизонтальних та вертикальних ребер. Значення *Left* найчастіше повертається при виявленні горизонтального, а *Above* – при виявленні вертикального ребра. Якщо ребро не виявлено, то повертається значення з площини над точкою *X*, що проходить у тривимірному просторі через точки *Left*, *Above* та *LeftAbove*. Два останні предиктори описують змішану еволюційно-шумову модель і належать до нелінійних предикторів [45]. Опис інших предикторів та функції для їх реалізації можна віднайти в [7].

Проблема вибору статичних предикторів для стиснення рядків пікселів різних зображень у форматі PNG залишалася *невирішеною* до останнього часу [25, с. 317]. Ми опишемо один із способів її вирішення у підрозділі 3.2.2 (вперше ідея цього способу наведена в [4]).

Сучасні методи стиснення зображень (наприклад, JPEG-LS [71; 72; 35]) для додаткового зменшення міжелементної надлишковості коригують значення функції предиктора, компенсуючи її систематичні відхилення від реальних значень попередніх елементів для кожного з наборів дискретизованих значень приростів найближчих суміжних елементів [57]. Ми дослідимо ефективність даного алгоритму коригування значень предикторів у пункті 4.2.4.

Отже, формат PNG орієнтований на зменшення надлишковостей чотирьох різновидів, що опрацьовуються способами його кодування згідно табл. 1.2.

Таблиця 1.2

Використання способів кодування формату PNG для зменшення різновидів надлишковостей

№ з/п	Різновид надлишковості	Способи кодування формату PNG		
		Предиктори	Алгоритм LZ77	Кодування HUFF
1	Міжелементна надлишковість між однаковими приростами яскравостей компонентів пікселів фрагментів зображення	Так	Так	Так
2	Міжелементна надлишковість між суміжними пікселами, що мають близькі яскравості компонентів	Так	Ні	Так
3	Міжелементна надлишковість між однаковими фрагментами яскравостей компонентів пікселів зображення	Ні	Так	Так
4	Кодова надлишковість між переважаючими яскравостями компонентів пікселів зображення	Ні	Ні	Так

1.2. Додаткові методи зменшення надлишковостей, що можуть застосовуватися у форматі PNG

У процесі дисертаційного дослідження нами розроблено дві групи методів для зменшення надлишковостей, що не використовуються у форматі PNG та дають змогу у випадку свого застосування, не впливаючи кардинально на час декодування, суттєво зменшити КС трикомпонентних зображень у цьому форматі:

➢ **перехід до альтернативних кольорових моделей з нижчою кореляцією компонентів**. У форматі PNG компоненти RGB-зображень кодуються послідовно у межах кожного піксела. При цьому кореляційні залежності між окремими компонентами таких зображень не враховуються і не усуваються, хоча коефіцієнти їх попарної взаємної кореляції близькі до одиниці [26]. Для зменшення міжелементної надлишковості між різними компонентами RGB-зображень у сучасних форматах графічних файлів найчастіше використовуються фіксовані альтернативні кольорові моделі (YCrCb, YPrPb, RCT та інші) [25, с. 16-21; 37, с. 187-

191], хоча на сьогодні вже розроблені методи формування кольорових моделей для кожного зображення (наприклад, у роботі [12] в альтернативній кольоровій моделі до RCT замість компоненти яскравості запропоновано використовувати вхідну компоненту (R, G, або B) з найменшою насиченістю). Але всі ці кольорові моделі не враховують рівень кореляції між компонентами окремих зображень та особливості обробки даних у форматах стиснення без втрат. Тому ми у підрозділі 4.2 розглянемо власні методи формування різницевих кольорових моделей (вперше запропоновані у [55]), що орієнтуються на застосування предикторів та контекстно-незалежного кодування, оскільки ці два способи кодування найчастіше використовуються в процесі стиснення зображень без втрат [7];

- **палітрування результатів застосування предикторів.** Якщо зобразити кольори пікселів, не опрацьованих контекстно-залежним алгоритмом (для формату PNG – алгоритмом LZ77) після застосування предикторів, точками в тривимірному просторі RGB, то більшість з них, враховуючи перерозподіл частот значень елементів (див. рис. 1.5б) виявляться близькими до початку координат. Колір кожного з таких пікселів доцільно подати індексом найближчого кольору в палітрі та трьома зміщеннями відносно нього [51; 46; 48], після чого закодувати індекси пікселів в палітрі контекстно-незалежним алгоритмом, а зміщення – рівномірними кодами [14, с. 36]. Фактично, цей метод (на відміну від методу компресії палітрових зображень [3]) застосовує для збереження результатів векторного квантування [24, с. 52-54; 34, с. 64-66] групові коди [14, с. 36-37]. З опису ідеї методу зрозуміло, що він дає змогу досягнути менших КС у випадку збільшення нерівномірності розподілу значень індексів палітри та зменшення кількості бітів для запису зміщень. Прискорення ж декодування відбувається за рахунок застосування для таких пікселів контекстно-незалежного алгоритму лише до індекса в палітрі, а не до значень всіх трьох компонентів. Відповідний алгоритм формування та застосування палітри для стиснення RGB-зображень без втрат розглядається у підрозділі 4.3.

Наголосимо, що в роботі не розглядалися методи компресії, які кардинально сповільнюють як кодування, так і декодування (контекстного моделювання [76], адаптованої орієнтації [67, 76], перетворення BWT [61; 24, с. 183-228]).

1.3. Тестові набори файлів зображень та програмне забезпечення, яке використовувалося для апробації розроблених алгоритмів

Всі розроблені та описані в роботі алгоритми апробувалися на восьми файлах 24-бітних зображень стандартного набору ACT, характеристики яких наведено у табл. 1.3 (% унікальних кольорів – це відношення кількості унікальних (різних) кольорів пікселів до загальної кількості пікселів зображення). Завантажити TIFF-версії цих зображень можна, наприклад, з http://www.compression.ca/act/act-files.html чи з http://www.compression.ru/arctest/act/act-tif.htm. Даний набір містить як синтезовані (№№ 1, 2, 7) так і фотореалістичні (решта) зображення. Вибір цього тестового набору зумовлений різноплановістю його зображень.

Остаточні версії програм для стиснення зображень з мінімальними КС як у стандартному, так і у модифікованому форматі PNG додатково апробувалися (додаток Е) на 24 файлах 24-бітних фотореалістичних зображень стандартного набору KTCI (http://r0k.us/graphics/kodak/index.html), кожне з яких у форматі BMP має розмір 1153 Кб (768 x 512 або 512 x 768 пікселів).

Тестування розроблених алгоритмів проводилися на комп'ютері з процесором AMD-K6 300 MHz, 128 Mb RAM в операційній системі Windows 98 за допомогою власних програм, розроблених мовою програмування C на базі програми з CD до [25] для запису зображень у форматі PNG, у якій попередньо були внесені такі модифікації: забезпечена можливість виходу зі словника в буфер під час кодування повторів алгоритмом LZ77; максимальна кількість елементів в стиснутих блоках та блоках IDAT [62] збільшена до 65536; запроваджений аналіз додаткових бітів зміщень у форматі DEFLATE (приріст цих додаткових бітів не повинен перевищувати 75 % приросту закодованих бітів); застосований предиктор *PaethPredict* до всіх рядків зображень, вдосконалений алгоритм запису заголовка стиснутого блоку та відкинуті допоміжні текстові блоки.

Час декодування модифікацій програми, що реалізує стиснення в існуючому стандарті PNG в роботі, як правило, не наводиться, оскільки швидкість декодування майже не залежить від застосованих предикторів чи їх комбінацій та параметрів стиснутих блоків і складає в середньому на зазначеному комп'ютері для

синтезованих зображень без шумів 910 Кб/с, для синтезованих з шумами – 545 Кб/с, а для фотореалістичних – 350 Кб/с. Тому середній час декодування зображень наборів ACT та KTCI складає лише 3 с.

Таблиця 1.3

Характеристики зображень набору ACT

№ файла	Назва файла	Розмір, Кб	Розміри, пікселів	К-ть унікальних кольорів	% унікальних кольорів	Використання спектру, %
1	Clegg.bmp	2101	814 x 880	127696	17.83	0.76
2	Frymire.bmp	3622	1118 x 1105	3622	0.29	0.02
3	Lena.bmp	769	512 x 512	148279	56.56	0.88
4	Monarch.bmp	1153	768 x 512	78617	19.99	0.47
5	Peppers.bmp	769	512 x 512	111344	42.47	0.66
6	Sail.bmp	1153	768 x 512	75748	19.26	0.45
7	Serrano.bmp	1464	629 x 794	1313	0.26	0.01
8	Tulips.bmp	1153	768 x 512	118233	30.07	0.70

Продовж. табл. 1.3

№ файла	Особливості
1	Синтезоване, з шумами, декілька великих об'єктів
2	Синтезоване, один великий об'єкт
3	Фотореалістичне, декілька великих об'єктів
4	Фотореалістичне, один великий і багато малих об'єктів
5	Фотореалістичне, декілька великих об'єктів
6	Фотореалістичне, багато середніх об'єктів
7	Синтезоване, один великий фрагментований об'єкт
8	Фотореалістичне, декілька великих об'єктів

Результати виконання програми, що реалізує стиснення за допомогою запропонованих алгоритмів в існуючому стандарті PNG, в основному порівнювалися з результатами програми OptiPNG (завантажити її можна, наприклад, з http://www.optipng.sourceforge.net), яка генерує короткі PNG-файли за результатами перебору предикторів, розмірів стиснутих блоків та стратегій стиснення. На сьогодні ця програма найкраще стискає PNG-файли (не враховуючи програми, розробленої за результатами дослідження) та найчастіше рекомендується з цією метою в Інтернеті. КС та час стиснення програми з модифікаціями формату PNG додатково порівнювалися з аналогічними показниками поширеного архіватора RAR v. 3.0 та програми для стиснення зображень ERI v. 5.1, яка забезпечує найкращі на сьогодні КС зображень набору ACT (за даними http://www.compression.ru/arctest/act/act-tif.htm).

Під час розробки програм не робилося жодних спроб підвищення ефективності кодування, орієнтованих на зображення згаданих тестових наборів, тому аналогічні результати тестування до наведених в роботі отримуються і на інших зображеннях тих самих типів.

1.4. Постановка завдань дослідження

Враховуючи зазначене вище, для досягнення мети дослідження у наступних розділах роботи потрібно розв'язати такі **завдання.**

- **У напрямку прискорення стиснення зображень** (розділ 2): розробити алгоритм для ефективнішого використання хеш-ланцюгів в процесі формування розкладів алгоритму LZ77; дослідити можливості пришвидшення розрахунків розмірів стиснутих блоків за відомими розподілами частот їх елементів в процесі вибору найкоротшого блоку з декількох альтернативних.
- **Для зменшення КС зображень у форматі PNG** (розділ 3): розробити алгоритм генерування, зменшення розмірів та вибору найкоротшого серед альтернативних стиснутих блоків для різних розкладів LZ77; вдосконалити механізм вибору предикторів як для окремого рядка, так і для блоку однорідних рядків пікселів; запропонувати власні варіанти зменшення КС розкладів LZ77; дослідити

можливості попереднього аналізу зображень перед стисненням у цьому форматі.

➢ **У напрямку внесення модифікацій у формат PNG** (розділ 4): дослідити можливості організації вибору різних предикторів для фрагментів рядків пікселів зображень, розробити та реалізувати методи генерування для кожного зображення різницевих кольорових моделей, орієнтованих на використання предикторів та контекстно-незалежного кодування; вивчити можливості використання палітри для групового статистичного кодування.

1.5. Висновки до першого розділу

Проведений аналіз ефективності способів кодування формату PNG дає змогу зробити такі основні **висновки**:

1. Предиктори та алгоритм LZ77, усуваючи міжелементну надлишковість, впливають на рівень кодової надлишковості, яка зменшується в цьому та в інших випадках кодуванням HUFF. Саме тому динамічні коди HUFF доцільно використовувати для кожного стиснутого блоку даних. Алгоритм LZ77 теж доцільно використовувати для всіх стиснутих блоків за умов відкидання замін, що кодуються більшою кількістю бітів, ніж їх окремі літерали, та відсутності жорстких обмежень по часу. Отже, в процесі попереднього аналізу зображень потрібно насамперед визначити номери предикторів окремих рядків, що забезпечують мінімальний КС.

2. Найтривалішим етапом в процесі стиснення зображень у форматі PNG є формування розкладу алгоритму LZ77, найшвидшим – застосування предикторів, яке, однак, неефективне без кодування HUFF. Тому у випадку найшвидшого стиснення доцільно обмежитися використанням предиктора, що найкраще в середньому мінімізує ентропію, і закодувати його результати кодами HUFF.

3. Суттєво підвищити ефективність стиснення зображень у форматі PNG можна лише за умови сукупної реалізації найефективніших алгоритмів кожного з трьох напрямків: прискорення кодування, зменшення КС у затвердженому стандарті формату та внесення модифікацій у цей формат.

РОЗДІЛ 2
ПРИСКОРЕННЯ СТИСНЕННЯ ЗОБРАЖЕНЬ У ФОРМАТІ PNG

Проаналізуємо алгоритми, що реалізують способи кодування формату PNG, з метою виявлення можливостей підвищення швидкості кодування у цьому форматі. Застосування предикторів полягає у відніманні значення функції прогнозу від дискретизованого значення яскравості кожної компоненти чергового піксела, тому алгоритмічно прискорити його неможливо. Генерування довжин кодів HUFF для кожного елемента розподілів літералів/довжин та зміщень виконується лише один раз для кожного стиснутого блоку, і тому прискорення відповідних алгоритмів (див., наприклад, [25, с. 91-108]) несуттєво впливає на швидкість стиснення. На формування ж розкладу алгоритму LZ77 припадає, за нашими спостереженнями, понад 50 % часу кодування зображень. Тому у підрозділі 2.1 дослідимо можливості прискорення формування розкладу алгоритму LZ77 за рахунок ефективнішого використання хеш-ланцюгів [47; 56].

Крім цього, у підрозділі 3.1 буде наведено алгоритм генерування частот розподілів літералів/довжин та зміщень альтернативних стиснутих блоків. Зрозуміло, що серед цих блоків для збереження даних зображення необхідно обирати той, що має найменшу довжину. Тому у підрозділі 2.2 опишемо ефективні алгоритми для точного і наближеного розрахунку довжин блоків динамічних кодів HUFF за частотами їх елементів без генерації цих кодів [53].

2.1. Вибір найкоротших хеш-ланцюгів у процесі пошуку однакових послідовностей для формування розкладу алгоритму LZ77

Традиційно, виконуючи пошук найдовшої однакової послідовності за допомогою хешування в процесі формування розкладу LZ77, переглядають послідовності, що починаються у словнику з ключа початку буфера. Наприклад, шукаючи найдовшу однакову послідовність для буфера "2, 4, 1, 3" у словнику "2, 4, 1, 2, 2, 4, 1, 3, 2, 4, 1", що вже згадувалися раніше, з використанням хеш-функції сумування кодів трьохелементних ключів, аналізують лише послідовності, котрі починаються з ключів, яким відповідає хеш-значення 2+4+1=7 (рис. 2.1).

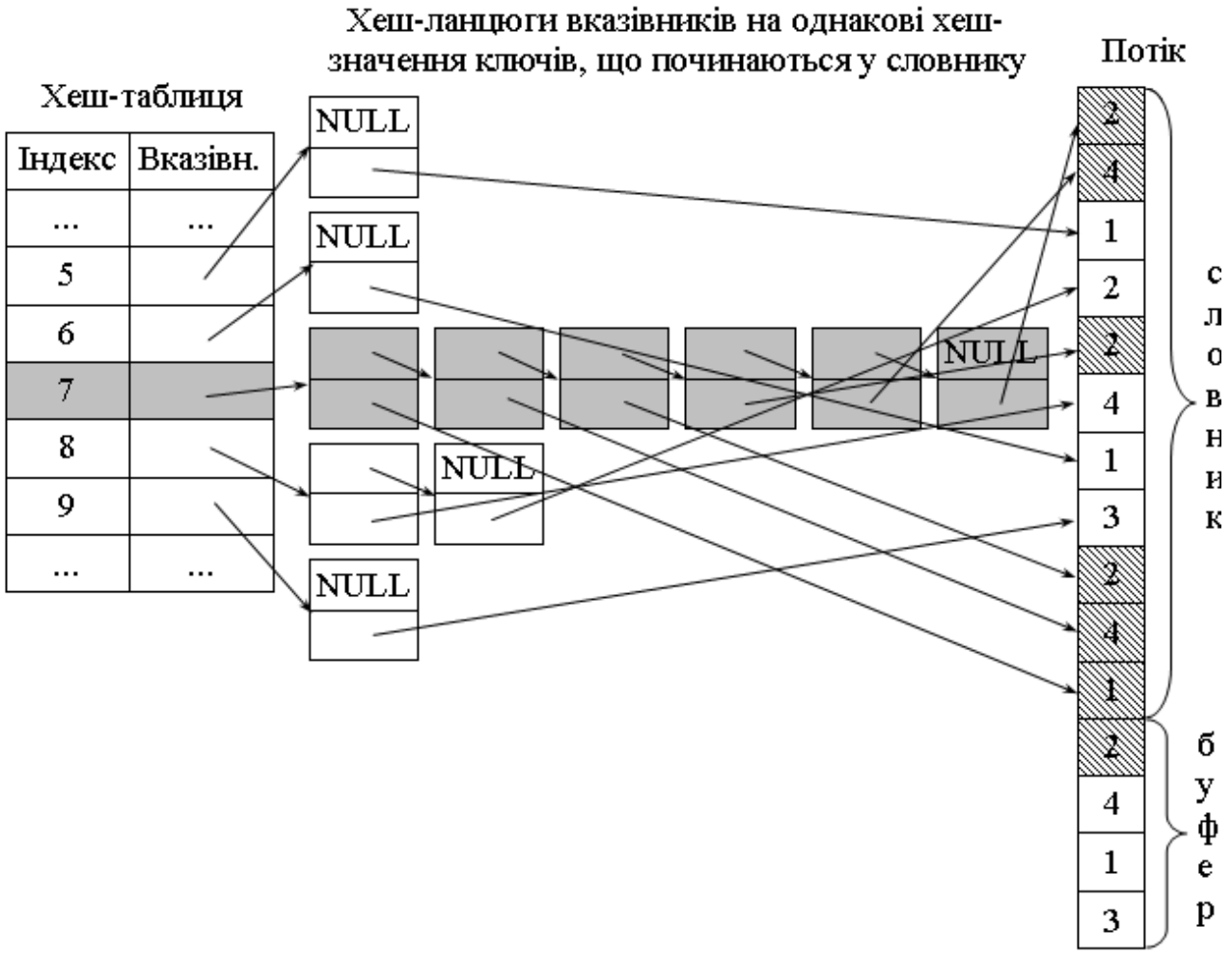


Рис. 2.1. Елемент хеш-таблиці та хеш-ланцюг вказівників (виділені сірим кольором) на трьохелементні ключі з хеш-значенням "7" (початки заштриховані) хеш-функції сумування значень елементів

Але, як зазначалося у підпункті 1.1.1.4, однорідні дані потоку можуть породжувати як завгодно довгі хеш-ланцюги і на практиці, для прискорення пошуку однакових послідовностей, виконують їх відсікання, аналізуючи обмежену кількість перших елементів цих ланцюгів (наприклад, 128), і цим погіршують КС. Тому розглянемо алгоритм для прискорення пошуку найдовшої однакової послідовності у словнику шляхом ефективнішого використання результатів хешування.

Для опису алгоритму вибору найкоротших хеш-ланцюгів у процесі пошуку однакових послідовностей у загальному випадку позначимо кількість елементів ключа через *lenKey*. Нехай для послідовності елементів буфера $s_1 \dots s_{len}$ довжини *len* (*len>=lenKey*) в результаті перегляду хеш-ланцюга вже знайдена однакова послідовність у словнику. Очевидно, що метою подальшого пошуку є відшукання у

словнику однакової послідовності довжини хоча б $len+1$ для послідовності елементів буфера $s_1 \dots s_{len+1}$, яку назвемо *розширеною*. Основна **ідея алгоритму** полягає в тому, що пошук розширеної послідовності в словнику можна продовжити не лише серед послідовностей, в яких ключі початку співпадають з ключем початку буфера $s_1 \dots s_{lenKey}$, а й серед послідовностей, в яких ключі з другого елемента співпадають з ключем з другого елемента розширеної послідовності $s_2 \dots s_{lenKey+1}$, або ключі з третього елемента співпадають з ключем з третього елемента розширеної послідовності $s_3 \dots s_{lenKey+2}$ і т.д. аж до ключа $s_{len-lenKey+2} \dots s_{len+1}$. Тобто **подальший пошук можна виконувати по хеш-ланцюгах довільного ключа, що повністю належить розширеній послідовності буфера**. Певна річ, що переходити до іншого хеш-ланцюга доцільно лише тоді, коли в ньому міститься менше елементів, ніж залишилося переглянути в активному хеш-ланцюгу.

Для реалізації запропонованого алгоритму в хеш-таблиці поряд з вказівками на вершини відповідних хеш-ланцюгів збережемо кількості їх елементів. Перераховувати ці кількості щоразу не потрібно: під час внесення елемента в словник хеш-ланцюг відповідного ключа доповнюється та очолюється новим елементом і кількість його елементів збільшується на одиницю; при вилученні ж елемента з початку словника анулюється останній елемент хеш-ланцюга відповідного ключа і кількість його елементів зменшується на одиницю. Зберігання кількостей елементів хеш-ланцюгів для кожного індекса хеш-таблиці, як правило, не призводить до зайвих витрат пам'яті (звичайно, якщо кількість елементів хеш-таблиці співпадає з кількістю елементів хеш-ланцюгів), оскільки при цьому відпадає необхідність зберігання вказівок на попередні елементи хеш-ланцюгів, що використовуються для забезпечення коректного вилучення останніх елементів. Дуже важливим є включення в процес аналізу наступного елемента s_{len+1} стосовно вже віднайдених у словнику, адже ключ з цим елементом дає змогу реально оцінити перспективність подальшого пошуку. Очевидно, що оцінювати на доцільність переходу слід лише хеш-ланцюги, що не оцінювалися раніше. Тому виконувати таке оцінювання будемо лише у випадку виявлення довшої однакової послідовності у словнику і лише для хеш-ланцюгів, які відповідають новим ключам розширеної

послідовності буфера. Аналізувати нові ключі в розширеній послідовності слід підряд зліва направо для уникнення пропусків однакових послідовностей, що починаються в словнику, але завершуються в буфері.

Покроково алгоритм вибору найкоротших хеш-ланцюгів у процесі пошуку однакових послідовностей записується так:

1. Присвоїти *len* значення нуль та перейти на початок хеш-ланцюга, який відповідає ключу початку буфера.

2. Послідовно перебираючи елементи поточного хеш-ланцюга, віднайти у словнику однакову послідовність довжини *newLen* (*newLen*>*len*).

3. У випадку відсутності довшої однакової послідовності у словнику після перегляду всіх елементів ланцюга або при відшуканні збігу максимально можливої довжини чи закінченні елементів буфера завершити виконання алгоритму.

4. Інакше, якщо кількість неопрацьованих ключів розширеної послідовності буфера менша від кількості нерозглянутих елементів поточного ланцюга, то віднайти найкоротший хеш-ланцюг серед тих, які відповідають ключам, що починаються з елементів буфера $s_{len-lenKey+3}, ..., s_{newLen-lenKey+2}$. Якщо при цьому кількість елементів віднайденого найкоротшого хеш-ланцюга менша від кількості нерозглянутих елементів поточного ланцюга, то перейти на початок віднайденого хеш-ланцюга.

5. Присвоїти *len* значення *newLen* та перейти до кроку 2.

Наведемо приклад застосування розглянутого алгоритму під час пошуку найдовшої однакової послідовності для буфера та словника з рис. 2.1 за умови, що далі в потоці йде значення, яке до цього не зустрічалося (рис. 2.2). У відповідності з першим кроком алгоритму, на початку пошуку вважаємо найдовшу однакову послідовність порожньою. Ключу початку буфера, згідно хеш-функції сумування кодів трьохелементних ключів, відповідає хеш-значення 7 (початки ключів з цим хеш-значенням заштриховані на рис. 2.2 по діагоналі), тому розпочнемо пошук з відповідного йому хеш-ланцюга (на рис. 2.2 виділено світло-сірим кольором).

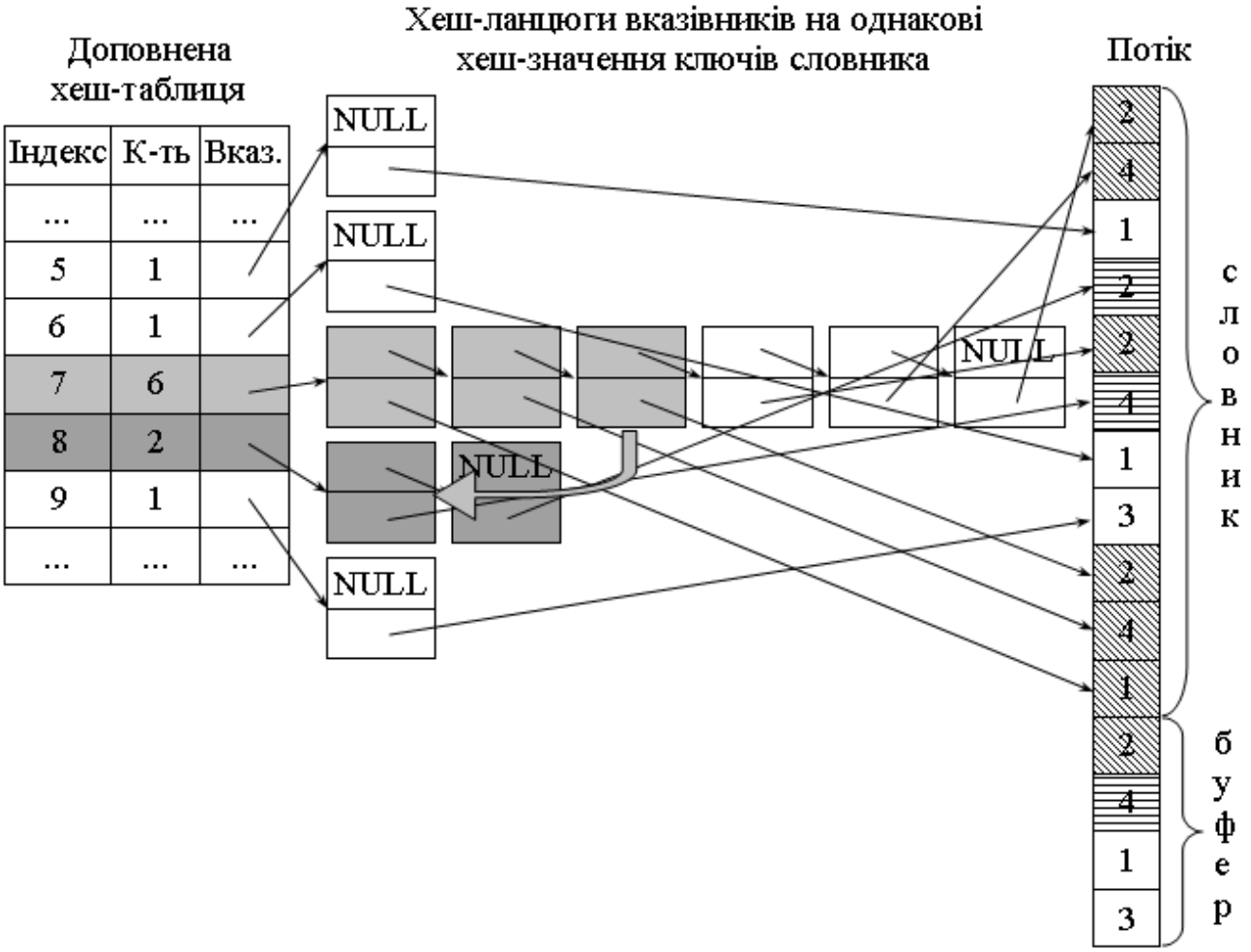


Рис. 2.2. Перехід до коротшого хеш-ланцюга (позначений товстою стрілкою) в процесі пошуку найдовшої однакової послідовності для буфера "2, 4, 1, 3" у словнику "2, 4, 1, 2, 2, 4, 1, 3, 2, 4, 1" з використанням хеш-функції сумування кодів трьохелементних ключів

Перший елемент цього хеш-ланцюга вказує на ключ "1, 2, 4", що не співпадає з ключем початку буфера, тому переходимо до наступного елемента хеш-ланцюга. Аналогічно опрацьовуємо другий елемент активного хеш-ланцюга. Третій елемент поточного хеш-ланцюга вказує на ключ "2, 4, 1", який співпадає з початковим ключем буфера, тому довша однакова послідовність у словнику знайдена (другий крок алгоритму). Оскільки кількість неопрацьованих ключів розширеної послідовності буфера рівна 1, а кількість нерозглянутих елементів поточного хеш-ланцюга дорівнює 3, то, згідно четвертого кроку алгоритму, виконаємо пошук коротших хеш-ланцюгів. В розширеній послідовності буфера "2, 4, 1, 3" міститься новий ключ "4, 1, 3", якому відповідає хеш-значення 8 (4+1+3). Хеш-ланцюг для

цього хеш-значення (початки відповідних ключів виділені горизонтальними лініями) містить два елементи, що менше від кількості елементів, які залишилися в поточному хеш-ланцюгу (тобто, 3), тому виконаємо перехід до віднайденого хеш-ланцюга (на рис. 2.2 позначено товстою стрілкою). Згідно п'ятого кроку алгоритму, повертаємося до другого кроку та шукаємо у словнику по поточному хеш-ланцюгу серед послідовностей, що з другого елемента містять "4, 1, 3" (заштриховано горизонтальними лініями) ті, що починаються з "2". Цей хеш-ланцюг містить два елементи (на рис. 2.2 виділені темно-сірим кольором), які аналізуються до завершення виконання алгоритму. Таким чином, використовуючи алгоритм вибору найкоротших хеш-ланцюгів, нам вдалося віднайти у словнику найдовшу однакову послідовність "2, 4, 1, 3" після аналізу п'яти елементів хеш-ланцюгів замість аналізу шести елементів, які містяться в ланцюгу ключа початку буфера.

Мовою С алгоритм вибору найкоротших хеш-ланцюгів можна реалізувати, наприклад, такою функцією:

```
void PngEncoder::LongestMatch (unsigned int &bestLength,
                               unsigned int &bestOffset)
 {bestLength = 0;
  unsigned long bestPoz; // позиція найдовшої віднайденої послідовності
  // обчислення значення хеш-функції стосовно активної позиції
  unsigned int hashvalue = HashValue(currentPozImage);
◆unsigned int lenAnalizHash=lenKeyHash; // розглядається ключ початку
  if (countHashValue[hashvalue] == 0) return; // якщо хеш-ланцюг порожній
  HashEntry *current;
  unsigned int countRivni, countAnalizValue=0; // к-ть розглянутих значень
  // кількість елементів, що залишилися в поточному хеш-ланцюгу
  unsigned long countZaluchokPoz=countHashValue[hashvalue];;
  current=hash_table[hashvalue]; // перехід до першої вершини ланцюга
◆unsigned int delta=0; // зміщення початку буфера від ключа ланцюга
◆startAnalizPoz:
  // search_limit вказує на максимальну кількість елементів
  // хеш-ланцюгів для перебору
  while (countZaluchokPoz > 0 && countAnalizValue++ < search_limit)
   {if (currentPozImage-(current->index-delta) > PngWindowSize)
     break; // вже вийшли за межі словника
    // визначаємо кількість однакових елементів з початку буфера
    // та відповідні позиції словника
    comparePosl(currentPozImage, current->index-delta, countRivni);
    countZaluchokPoz--;
    if (countRivni > bestLength) // знайшли довшу однакову послідовність
     {bestLength = countRivni;
      bestPoz = current->index-delta; // запам'ятали її положення
```

```
    if (bestLength == PngLongestLength)
     break; // знайдено однакову послідовність максимально можливої довжини
    // якщо доцільний пошук коротших хеш-ланцюгів
    if ((int)countZaluchokPoz>(int)bestlength-lenAnalizHash)
     {while (lenAnalizHash <= bestLength) // цикл по нових ключах
       {lenAnalizHash++;
       // обчислюємо хеш-значення нового ключа
       unsigned int newHashValue =
        HashValue(currentPozImage+lenAnalizHash-lenKeyHash);
       if (countZaluchokPoz > countHashValue[newHashValue])
        {// перехід до хеш-ланцюга з меншою кількістю елементів
         countZaluchokPoz = countHashValue[newHashValue];
         current = hash_table[newHashValue]; delta = lenAnalizHash-lenKeyHash;
         goto startAnalizPoz; }}}}
   current=current->next; }
  if (bestLength < lenKeyHash)
   {bestLength=0; return; } // однакові послідовності не можуть бути коротшими ключа
  bestOffset=currentPozImage - bestPoz; return; }
```

Зауважимо, що без позначених вище рядків та описаних у них змінних наведена функція реалізує алгоритм пошуку найдовшої однакової послідовності згідно ключа початку буфера.

Розглянемо **результати застосування** різних варіантів аналізу хеш-ланцюгів під час стиснення зображень набору ACT (див. табл. 1.3) у форматі PNG. КС кожного зображення та набору в цілому наведено в табл. 2.1, а час формування "жадібного" розкладу LZ77 – у табл. 2.2. Тестування проводилося як для алгоритму аналізу хеш-ланцюгів згідно ключів початку буфера (див. у таблицях варіанти 1, 3), так і для описаного алгоритму вибору найкоротших хеш-ланцюгів (варіанти 2, 4). При цьому для кожного алгоритму аналізувалися модифікації з відсіканням хеш-ланцюгів (варіанти 1, 2) та без нього (варіанти 3, 4).

Як свідчать дані цих таблиць, використання алгоритму вибору найкоротших хеш-ланцюгів у випадку їх відсікання не лише прискорює формування розкладу алгоритму LZ77 максимум на 16 %, а й зменшує КС окремих файлів максимум на 0.5 % завдяки ефективнішому аналізу послідовностей словника.

Таблиця 2.1

КС файлів зображень набору ACT у форматі PNG після застосування різних варіантів аналізу хеш-ланцюгів "жадібного" розкладу алгоритму LZ77, %

№ з/п	Опис варіанта аналізу	№ файла								Середній
		1	2	3	4	5	6	7	8	
1	До 128 елементів хеш-ланцюгів згідно ключів початків буферів	23.94	10.27	67.75	55.85	58.65	67.65	10.38	61.23	44.47
2	До 128 елементів найкоротших хеш-ланцюгів	23.80	9.77	67.75	55.68	58.65	67.65	9.97	61.14	44.30
3	Всі елементи хеш-ланцюгів згідно ключів початків буферів	23.80	9.75	67.75	55.68	58.65	67.65	9.90	61.14	44.29
4	Всі елементи найкоротших хеш-ланцюгів	23.80	9.75	67.75	55.68	58.65	67.65	9.90	61.14	44.29

Таблиця 2.2

Час формування "жадібного" розкладу алгоритму LZ77 з різними варіантами аналізу хеш-ланцюгів в процесі стиснення файлів зображень набору ACT у форматі PNG, с

№ з/п	Опис варіанта аналізу	№ файла								Середній
		1	2	3	4	5	6	7	8	
1	До 128 елементів хеш-ланцюгів згідно ключів початків буферів	6.05	7.85	2.35	6.42	3.69	4.33	3.25	5.22	4.90
2	До 128 елементів найкоротших хеш-ланцюгів	5.60	7.64	2.26	5.39	3.19	3.74	3.25	4.46	4.44
3	Всі елементи хеш-ланцюгів згідно ключів початків буферів	92.95	371.35	2.49	10.02	4.02	4.46	151.38	6.38	80.38
4	Всі елементи найкоротших хеш-ланцюгів	9.90	12.14	2.26	5.39	3.19	3.74	5.99	4.46	5.89

Використання розробленого алгоритму для аналізу всього словника дозволяє прискорити формування цього розкладу для синтезованих файлів в десятки разів, а

також дає змогу аналізувати всі елементи хеш-ланцюгів, витрачаючи максимально лише на 85 % більше часу, ніж у випадку використання відсікання ланцюгів, довших 128-и елементів, що дозволяє для синтезованих файлів забезпечити кращий КС (див. рядки 1, 4 з табл. 2.1, 2.2).

Отже, розроблений алгоритм прискорює процес пошуку найдовших однакових послідовностей для елементів буфера в словнику завдяки вибору найкоротших хеш-ланцюгів серед допустимих. Саме тому даний алгоритм надалі застосовується у всіх наступних модифікаціях базової тестової програми. Ефективність застосування запропонованого алгоритму в основному залежить від кількостей однакових ключів потоку, тобто від рівня міжелементної надлишковості: чим більше однакових ключів зустрічається в потоці (для зображень – чим менше різних кольорів), тим довші хеш-ланцюги вони породжують і тим більша доцільність переходу до коротших хеш-ланцюгів.

2.2. Способи та алгоритми розрахунків довжин блоків динамічних кодів HUFF

Як згадувалося вище, для кожного стиснутого блоку у форматі словникової компресії DEFLATE, що використовується у форматі PNG для зберігання яскравостей компонентів пікселів після застосування предикторів, з метою забезпечення мінімальних КС генеруються два блоки кодів HUFF – для літералів/довжин та для зміщень [25, с. 295-297]. Визначення розміру таких блоків дає змогу обрати найкоротший стиснутий блок з альтернативних як в процесі попереднього аналізу, так і безпосередньо під час кодування.

Зрозуміло, що розраховувати довжину блоку кодів HUFF для чергової послідовності елементів можливо безпосередньо, генеруючи та використовуючи ці коди: підрахувати абсолютні частоти кожного елемента; сформувати за цими частотами коди HUFF, як це описано в підрозділі 1.1.2; визначити їх довжини та обчислити суму добутків частоти кожного елемента на довжину його коду HUFF. На практиці ж розраховувати довжину блоку кодів HUFF можна значно швидше, не генеруючи безпосередньо ці коди для кожного елемента. Розглянемо два способи

виконання таких розрахунків.

2.2.1. Наближене обчислення довжини блоку кодів HUFF за допомогою ентропії. Як відомо, середня довжина коду HUFF близька до ентропії [8, с. 642-644; 24, с. 35], тому загальна довжина блоку таких кодів для послідовності елементів наближено дорівнює сумі довжин їх ентропійних кодів, тобто *довжині ентропійного коду послідовності*. Виведемо формулу для обчислення цієї довжини. Нехай кожен з елементів s_i зустрічається N_i разів в послідовності довжини $N = \sum_i N_i$. Згідно статистичного означення ймовірності, $p(s_i) = N_i / N$, тому загальна довжина ентропійного коду послідовності, враховуючи (1.3), наближається до значення

$$L_H = N \times H = N \log(N) - \sum_i N_i \log(N_i). \qquad (2.1)$$

Мовою С підпрограма для наближеного обчислення довжини блоку кодів HUFF (тобто довжини ентропійного коду) за частотами його елементів згідно (2.1) з визначенням прогнозованого КС має вигляд:

```
double lenEntropiCode(long * freq, short countAllFreq=256)
 {// freq - масив частот елементів
  // countAllFreq - загальна кількість частот в масиві
  double len=0; long n=0;
  for (long i=0; i<countAllFreq; i++)
   {n+=freq[i]; // сумуємо частоти
    if (freq[i]>1) // для існування log
     len+=freq[i]*log(freq[i]); }
  if (n) {len=(n*log(n)-len)/log(2); }
  else {len=0; }
  return len; }            .
```

Застосування цієї підпрограми дає змогу пришвидшити наближене обчислення довжини блоку кодів HUFF у порівнянні з варіантом безпосередньої генерації цих кодів для блоків літералів/довжин в середньому на 96.5 %, а для блоків зміщень – на 80.8 %.

Додатково прискорити виконання цієї підпрограми можна за рахунок зменшення кількостей обчислень натуральних логарифмів. Як свідчать експерименти, після застосування предикторів більшість значень елементів, віддалених від нуля, мають невелику частоту (див., наприклад, рис. 1.5 б). В процесі

ж вибору варіанту стиснення кожного рядка зображення у форматі PNG кодування предикторами застосовуються у 80 % випадків (4 варіанти з 5). Крім цього, довші зміщення трапляються у розкладі алгоритму LZ77, як правило, рідше від коротших, оскільки однакові послідовності у словнику для цього розкладу шукаються від менших зміщень до більших. Тому добутки невеликих частот на їх натуральні логарифми доцільно не обчислювати щоразу, а брати з масиву, заповненого на початку виконання програми. Ми, наприклад, використовували з цією метою масив таких значень для частот з діапазону [2; 15], оскільки більші частоти повторюються порівняно рідко. Тоді фрагмент наведеної підпрограми для обчислення сум добутків частот елементів на їх натуральні логарифми перепишеться у вигляді:

```
if (freq[i]>1) // для існування log
 if (freq[i]>15) len+=freq[i]*log(freq[i])
 else len+=masBit(freq[i]);        .
```

Використання такого масиву значень добутків невеликих частот на їх натуральні логарифми дозволило додатково підвищити швидкість наближеного обчислення довжини блоку кодів HUFF у порівнянні з варіантом безпосередньої генерації цих кодів для блоків літералів/довжин в середньому на 0.37 %, а для блоків зміщень – на 0.57 %. Ми використаємо даний спосіб наближеного обчислення довжини блоку кодів HUFF в процесі попереднього аналізу зображень, оскільки він дає змогу реально оцінити нерівномірність розподілу частот елементів.

2.2.2. Точне обчислення довжини блоку кодів HUFF з використанням принципу їх генерації. Для точного обчислення довжини блоку кодів HUFF за частотами його елементів дослідимо детальніше принцип генерації цих кодів, розглянутий в підрозділі 1.1.2: під час кожної ітерації серед всіх частот елементів шукають дві найменші і надалі розглядають їх суму, при цьому вхідним елементам, що утворили першу поєднувану частоту, дописують спереду код "0", а тим, що утворили другу – код "1" (або навпаки). Тобто **внаслідок поєднання двох найменших частот довжина блоку кодів HUFF збільшується на суму цих частот**. Цей принцип дозволяє ітеративно обчислювати довжину таких блоків без генерації самих кодів елементів: до отримання однієї частоти серед всіх додатних частот знаходимо дві найменші, збільшуємо довжину блоку на суму цих частот і

надалі замість цих двох частот розглядаємо їх суму.

Звичайно, щоразу знаходити в масиві частот дві найменші недоцільно, адже це значно сповільнить обчислення. Тому відсортуємо додатні ненульові частоти за спаданням і будемо поєднувати дві останні частоти. Крім цього, сума двох чергових частот не може бути меншою за суму двох попередніх поєднаних частот, оскільки щоразу обираються два найменших значення. Отже, чергова сума поєднання має бути вставлена перед попередньою сумою частот, що значно зменшує діапазон пошуку позиції її вставки у відсортований масив. Приклад таких поєднань частот наведено на рис. 2.3.

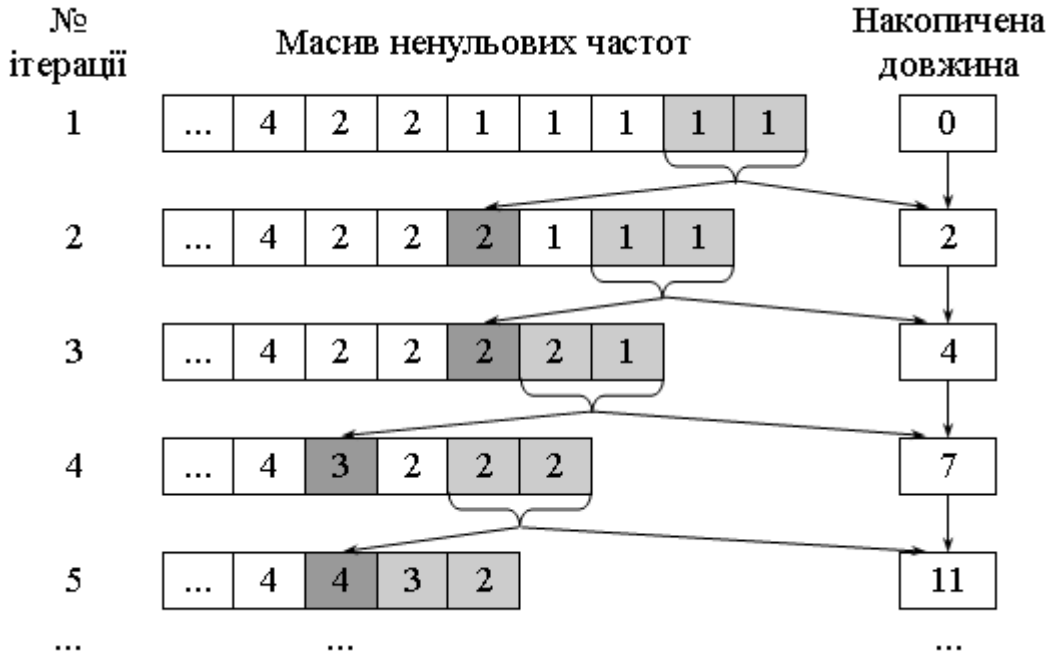


Рис. 2.3. Ітеративні поєднання двох найменших частот в процесі точного розрахунку довжини блоку кодів HUFF

Наведемо текст підпрограми мовою С, що реалізує обчислення довжини блоку кодів HUFF з використанням описаних підходів і бінарного пошуку позиції вставки ненульових частот під час їх початкового сортування за спаданням (тут ділення на 2 замінено побітовим зсувом):

```
long lenHuffmanCode(long *freq, short countAllFreq)
{// сортуємо ненульові частоти масиву за спаданням
 for (i=0; i<countAllFreq; i++)
  if (masFreq[i]>0) // частота елемента ненульова
   {element=masFreq[i];
```

```
   if (countFreq==0) j=0;
   else // бінарний пошук позиції для вставки
    {minIndex=-1; maxIndex=countFreq;
     while (maxIndex-minIndex>1)
      {j=(minIndex+maxIndex)>>1; // індекс середини
       if (masFreq[j]>=element) minIndex=j;
       else maxIndex=j; }
     j=maxIndex; }
   for (k=countFreq; k>j; k--) // зміщуємо менші частоти
    masFreq[k]=masFreq[k-1];
   masFreq[j]=element; // вставляємо знайдену частоту
   countFreq++; } // збільшуємо кількість ненульових частот
if (countFreq==0) return 0; // якщо частоти відсутні
if (countFreq==1) // один елемент кодується бітом
 return masFreq[0];
countBit=0;
// ненульових частот більше, ніж одна – використовуємо
// ітеративний принцип генерації динамічних кодів HUFF
j=countFreq-2; // максимальна позиція для вставки поєднаних частот
while (countFreq>2) // поки ненульових частот більше, ніж дві
 {// обчислюємо суму двох найменших частот
  element=masFreq[countFreq-1]+masFreq[countFreq-2];
  countBit+=element; // збільшуємо довжину на суму найменших частот
  // шукаємо позицію для прямого включення суми частот
  while (j>0 && masFreq[j-1]<element) j--;
  for (k=countFreq-2;k>j;k--) // зміщуємо менші частоти
   masFreq[k]=masFreq[k-1];
  masFreq[j]=element; // вставляємо суму частот
  countFreq--; } // зменшуємо к-ть необроблених частот
// кодуємо додатковим бітом дві найбільші частоти
return countBit+masFreq[0]+masFreq[1]; }
```

Реалізація цієї підпрограми дає змогу пришвидшити точне обчислення довжини блоку кодів HUFF у порівнянні з варіантом безпосередньої генерації цих кодів для блоків літералів/довжин в середньому на 87 %, а для блоків зміщень – на 78 %.

Додатково прискорити точний розрахунок довжини блоку кодів HUFF можна, насамперед, за рахунок врахування особливостей обробки окремих частот. Значну частину часу в процесі поєднання частот займає включення нової суми у відсортований масив, адже при цьому доводиться щоразу всі менші частоти від отриманої суми посувати вправо для формування позиції вставки. Пришвидшити виконання таких включень можна за допомогою використання однозв'язного списку, який реалізується додатковим масивом індексів більших елементів. Застосування такого списку дозволило додатково підвищити швидкість точного обчислення довжини блоку кодів HUFF у порівнянні з варіантом безпосередньої

генерації цих кодів для блоків літералів/довжин в середньому на 5 %, а для блоків зміщень – на 4 %.

Крім цього, прискорити розрахунок довжини таких блоків дає змогу врахування особливостей розподілу їх частот: як свідчать наші експерименти, після застосування предикторів біля 67 % частот елементів виявляються малими (до 15 включно) і часто повторюються (див., наприклад, рис. 1.5 б). Вставка таких частот у відсортований масив і подальше їх опрацювання займає багато часу, тому розглянемо алгоритм для їх окремої ефективнішої обробки. Проаналізуємо, наприклад, розподіл, в якому частота 1 повторюється 5 разів (рис. 2.4). Відразу можна визначити, що в процесі обробки буде поєднано 2 пари частот зі значенням 1 і внаслідок цих поєднань довжина блоку кодів збільшиться на 4 та буде створено 2 частоти зі значенням 2. П'ята частота зі значенням 1 поєднається з більшою частотою.

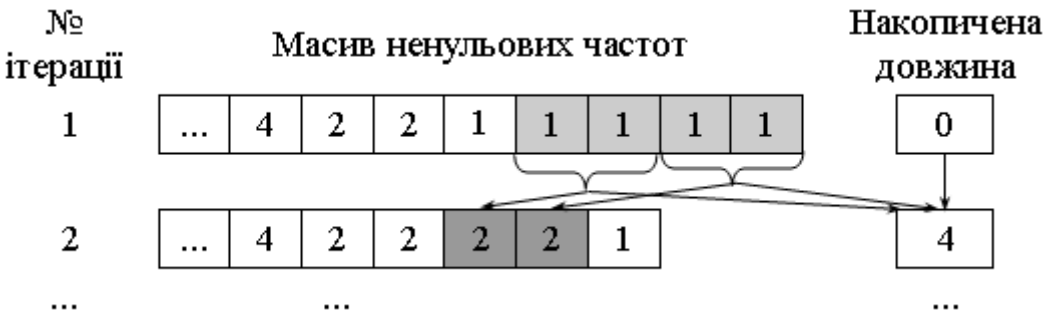


Рис. 2.4. Приклад окремого опрацювання малих частот в процесі точного розрахунку довжини блоку кодів HUFF

Тому **для частот до 30 включно підрахуємо кількості їх повторень (частоти частот) в окремому масиві, поєднаємо малі частоти (до 15 включно) з використанням цього ж масиву та запишемо у відсортований масив частот лише результати цих поєднань.** Дане вдосконалення дозволяє зменшити кількість елементів у відсортованому масиві частот на 30-40 %. Застосування такого масиву частот малих частот дозволило додатково підвищити швидкість точного обчислення довжини блоку кодів HUFF у порівнянні з варіантом безпосередньої генерації цих кодів для блоків літералів/довжин в середньому ще на 4 %, а для блоків зміщень –

ще на 3.5 %.

Фрагмент програми мовою С, що реалізує всі описані у цьому пункті модифікації, має вигляд:

```
long lenHuffmanCode(long *freq, short countAllFreq)
{// сортуємо в масиві частоти, більші 30, за спаданням;
 // для менших частот підраховуємо їх кількість
 for (i=0; i<countAllFreq; i++)
  if (masFreq[i]>0) // частота елемента ненульова
   if (masFreq[i]<31)
    {// підрахунок кількостей частот до 30 включно
     masCountMinFreq[masFreq[i]]++;
     countMinFreq++; } // к-ть мінімальних частот
   else // вставляємо частоту у відсортований масив
    {element=masFreq[i];
     if (countFreq==0) j=0;
     else // бінарний пошук позиції для вставки
      {minIndex=-1; maxIndex=countFreq;
       while (maxIndex-minIndex>1)
        {j=(minIndex+maxIndex)>>1; // індекс середини
         if (masFreq[j]>=element) minIndex=j;
         else maxIndex=j; }
       j=maxIndex; }
     for (k=countFreq;k>j;k--) // зміщуємо менші частоти
      masFreq[k]=masFreq[k-1];
     masFreq[j]=element; // вставляємо знайдену частоту
    countFreq++; } // збільшуємо кількість частот, більших 30
 if (countFreq+countMinFreq==0) return 0; // частоти відсутні
 countBit=0;
 if (countMinFreq>0) // частоти до 30 наявні
  {// аналізуємо частоти до 15 включно
   for (i=1; i<=15; i++)
nextElement:
    if (masCountMinFreq[i]>0) // наявна чергова кількість частот
     {if (masCountMinFreq[i]>1) // є однакові частоти
       {// знаходимо кількість пар однакових частот
        countParaFreq=masCountMinFreq[i]>>1;
        // враховуємо поєднання пар в довжині блоку
        countBit+=(countParaFreq<<1)*i;
        // поєднуємо пари однакових частот
        masCountMinFreq[i<<1]+=countParaFreq; }
      // залишилася одна частота – шукаємо їй пару
      if (masCountMinFreq[i] & 1)
       {element=i++;
        while (masCountMinFreq[i]==0 && i<=15) i++;
        if (i>15) // пару не знайдено – завершуємо аналіз
         {i=element; masCountMinFreq[i]=1; break; }
        // пару знайдено – враховуємо в довжині блоку
        countBit+=element+i;
        masCountMinFreq[element+i]++; // сумуємо частоти
        masCountMinFreq[i]--; // одну частоту опрацювали
```

```
      goto nextElement; }} // переходимо до більшої частоти
  // вставляємо поєднані частоти в кінець відсортованого масиву
  for (j=30; j>=i; j--)
   if (masCountMinFreq[j]>0)
    for (k=0; k<masCountMinFreq[j]; k++) masFreq[countFreq++]=j; }
if (countFreq==1)
 {if (countBit) return countBit; // результат поєднання малих частот
  else return masFreq[0]; } // один елемент кодується бітом
bottom=countFreq-1; // індекс найменшої частоти
for (i=bottom; i>0; i--)
 prev[i]=i-1; // список індексів більших частот в масиві
prev[0]=countAllFreq; // ознака закінчення списку
index=prev[prev[bottom]]; // позиція для поєднань
// циклічне поєднання двох найменших частот
while (prev[bottom]!=countAllFreq) // до однієї частоти
 {masFreq[prev[bottom]]+=masFreq[bottom];
  bottom=prev[bottom]; // посуваємо вершину списку
  countBit+=masFreq[bottom]; // враховуємо поєднання в довжині блоку
  // якщо сумарна частота більша від значення наступного елемента
  if (prev[bottom]<countAllFreq && masFreq[prev[bottom]]<masFreq[bottom])
   {i=bottom;
    bottom=prev[bottom]; // переміщуємо вершину
    // шукаємо позицію для вставки поєднання частот
    while (prev[index]<countAllFreq && masFreq[prev[index]]<masFreq[i])
     index=prev[index];
    // вставляємо суму частот в список
    prev[i]=prev[index]; prev[index]=i; index=i; }
  else index=prev[bottom]; }
return countBit; }
```

Отже, підвищити швидкість точного розрахунку довжини блоку кодів HUFF можна не лише за рахунок швидких алгоритмів сортування та використання списків, а й за допомогою врахування структури розподілу частот елементів, наприклад шляхом окремого попереднього опрацювання малих частот. Надалі використаємо розглянутий спосіб точного обчислення довжини блоку кодів HUFF в процесі стиснення зображень, оскільки він дає змогу точно передбачити розмір кожного з альтернативних стиснутих блоків (див. підрозділ 3.1).

2.3. Висновки до другого розділу

1. Прискорити формування розкладу LZ77 можливо не лише за рахунок відсікання хеш-ланцюгів, а й за допомогою ефективнішого використання результатів хешування, як у розглянутому алгоритмі вибору найкоротших хеш-ланцюгів. Даний алгоритм носить загальний характер, прискорює формування розкладу алгоритму LZ77 для синтезованих зображень максимум у 30 разів та не

сповільнює його генерацію для фотореалістичних знімків, а тому може бути рекомендований для використання не лише у процесі реалізації словникових алгоритмів, а й для інших алгоритмів, що застосовують хешування.

2. Розраховувати довжини блоків динамічних кодів HUFF за частотами елементів розподілів можна як наближено за допомогою ентропії, так і точно, використовуючи ітеративний принцип генерації цих кодів. Ентропійний спосіб хоча й оцінює довжину блоку кодів HUFF наближено, проте має нижчу складність програмної реалізації та дає змогу (з врахуванням розглянутої модифікації) пришвидшити такі розрахунки у порівнянні з варіантом безпосередньої генерації цих кодів для блоків літералів/довжин в середньому на 96.87 %, а для блоків зміщень – на 81.37 %. Точний спосіб обчислення цієї довжини з врахуванням описаних вдосконалень має вищу складність програмної реалізації та в середньому прискорює дані розрахунки відповідно на 96 % та 85.5 %. Ентропійний спосіб розрахунку довжини блоку кодів HUFF доцільно використовувати в процесі попереднього аналізу зображень, а точний – безпосередньо під час стиснення.

РОЗДІЛ 3
ЗМЕНШЕННЯ КОЕФІЦІЄНТІВ СТИСНЕННЯ ЗОБРАЖЕНЬ У ФОРМАТІ PNG

З метою виявлення можливостей зменшення КС зображень у форматі PNG проаналізуємо ще раз структуру та окремі способи кодування цього формату. Як зазначалося у розділі 1, фрагменти зображень у даному форматі зберігаються послідовно у відокремлених стиснутих блоках. Кожен стиснутий блок містить коди HUFF для розподілів літералів/довжин та зміщень і відповідні додаткові біти. Тобто коди окремих літералів, довжин та зміщень замін алгоритму LZ77 мають у межах кожного стиснутого блоку фіксовану довжину. Тому зменшити розміри цих блоків перед записом у PNG-файл можна шляхом відкидання замін, що записуються більшою кількістю бітів, ніж окремі літерали, які вони кодують [49; 52; 53]. Цей принцип покладено в основу алгоритму підрозділу 3.1.

Предиктори використовуються для збільшення рівня кодової надлишковості за рахунок міжелементної, причому кожен з п'яти предикторів, які можна обрати у форматі PNG для окремих рядків зображення, виконує це по різному. Зрозуміло, що під час кодування по замовчуванню для кожного рядка потрібно обирати такий предиктор, який дасть змогу найкраще виконати це перетворення [4; 5]. Ця можливість зменшення КС зображень реалізована у підрозділі 3.2.

Для зменшення КС кодування HUFF у підрозділі 3.3 ми розглянемо алгоритми фрагментування як рядків зображення, так і результатів розкладу алгоритму LZ77, що дають змогу збільшити нерівномірності розподілів частот окремих елементів [5].

Ефективність застосування алгоритму LZ77 для даних з високим рівнем міжелементної надлишковості залежить, насамперед, від обраного варіанту формування розкладу цього алгоритму (див. підрозділ 1.1.1). Тому у підрозділі 3.4 ми дослідимо можливості вдосконалення та запропонуємо власні модифікації існуючих розкладів алгоритму LZ77 [44; 45].

І, нарешті, у підрозділі 3.5 ми наведемо алгоритм попереднього аналізу зображень, який дає змогу досягнути найменших КС у форматі PNG не лише

завдяки врахуванню зазначених особливостей способів кодування цього формату, а й завдяки врахуванню їх взаємодії [1; 58], а у підрозділі 3.6 опишемо програмне забезпечення, яке реалізує таке стиснення.

3.1. Алгоритм мінімізації розміру стиснутого блоку

Опишемо алгоритм мінімізації довжини (загальної кількості бітів) кожного стиснутого блоку у форматі PNG перед його записом у файл шляхом заміщення неефективних замін відповідними літералами, враховуючи, що у цих блоках зберігаються коди літералів/довжин та зміщень розкладу алгоритму LZ77 і додаткові біти. Очевидно, що заміни алгоритму LZ77 слід вважати *ефективними*, якщо вони записуються не більшою кількістю бітів, ніж окремі літерали, які вони замінюють [53; 52; 49]. Оскільки окремі літерали і базові значення довжин та зміщення у форматі DEFLATE, який використовується для зберігання стиснутих блоків, записуються кодами HUFF, то заміна q довжини len_q, що виконується починаючи з літерала s_k за зміщенням $offset_q$, ефективна лише тоді, коли

$$\sum_{i=0}^{len_q-1} l_{s_{k+i}} \geq l_{len_q} + d_{len_q} + \lambda_{offset_q} + \delta_{offset_q}, \qquad (3.1)$$

де l_m – довжина коду HUFF літерала/довжини заміни m, d_m – кількість додаткових бітів для запису довжини заміни m, λ_m – довжина коду HUFF зміщення m, δ_m – кількість додаткових бітів для запису зміщення m.

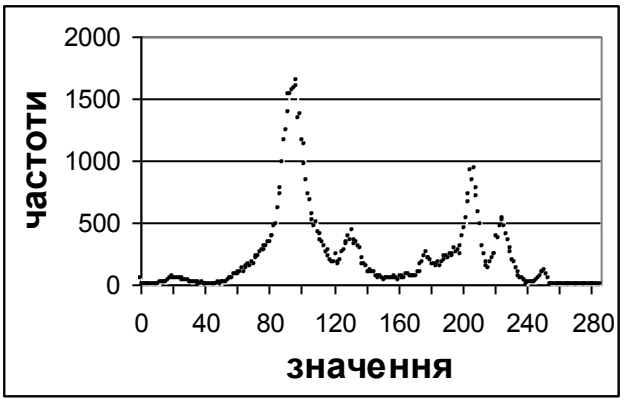
Для дослідження ефективності замін LZ77 проаналізуємо різні варіанти розподілів частот літералів/довжин у форматі PNG (оскільки коди цих елементів займають найбільше місця у стиснутих даних) на прикладі перших 64 Кб зображення *Lena.bmp*. Значення компонентів пікселів цього блоку мають слабо виражену нерівномірність розподілу частот (рис. 3.1 а), адже найбільша частота серед окремих елементів рівна 1650 (2.5% від загальної кількості). Ця нерівномірність значно посилюється після застосування предиктора Піфа (рис. 3.1 б), який дозволяє кардинально збільшити частоту нульового значення до 11405 (17.4 %), збільшити частоти елементів навколо нуля до понад 4000 (6.1%,

від'ємним результатам дії предикторів відповідають значення, збільшені на 256) та зменшити частоти всіх інших елементів.

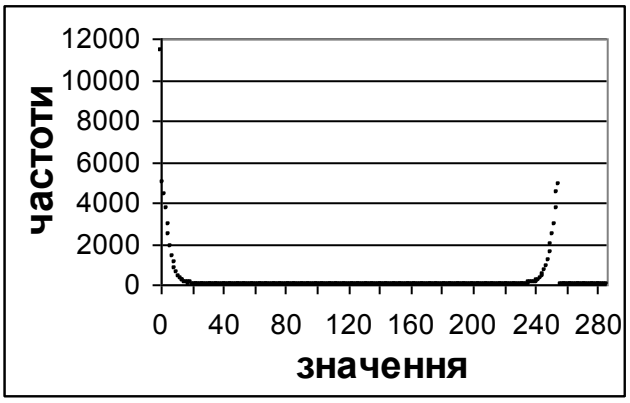
Збільшує нерівномірність розподілу частот і застосування замін однакових послідовностей (рис. 3.1 в). Але у фотореалістичних зображеннях, до яких належить і *Lena.bmp,* як правило, зустрічаються лише короткі повторення послідовностей елементів. Наприклад, у першому блоці цього зображення віднайдено 8016 повторень по 3 елементи (на рис. 3.1 відповідає значенню 257) та 933 повторення по 4 елементи (відповідає значенню 258). При цьому короткі однакові послідовності трапляються, як правило, серед елементів з найбільшою частотою, і тому застосування таких замін зменшує нерівномірність розподілу частот окремих елементів. Наприклад, частота елемента 96 зменшилася з 1650 до 513. Ще більше коротких однакових послідовностей зустрічається у фрагментах зображення після застосування предикторів (рис. 3.1 д). Наприклад, після застосування предиктора Піфа кількість 3-елементних однакових послідовностей зросла до 11066, а 4-елементних – до 3312. В той же час, зменшилася кількість найдовших однакових послідовностей з 23 до 22, оскільки застосування предикторів реалізує вплив суміжних елементів і здатне розбивати такі послідовності.

Зовсім інші властивості мають розподіли частот літералів/довжин у випадку неврахування коротких замін. Довгі заміни, як правило, зменшують максимальні частоти елементів, але не порушують характеру нерівномірності їх розподілу. Наприклад, застосування замін від 9 елементів зменшило максимальну частоту з 1650 (рис. 3.1а) лише до 1538 (рис. 3.1 е).

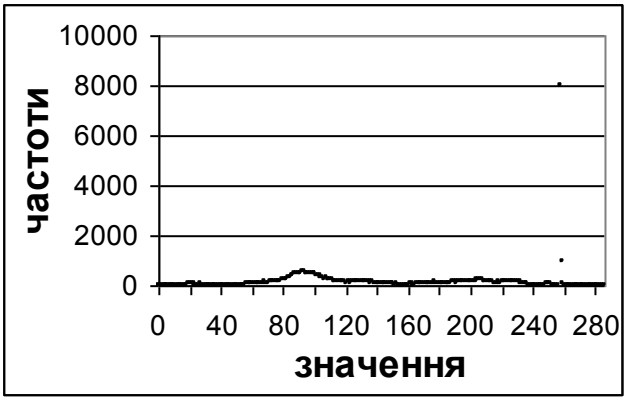
Отже, **короткі заміни виявляються ефективними** (рис. 3.1 в, д), як правило, **для стиснутих блоків, в яких інші короткі заміни тієї ж довжини теж враховуються.** Це пов'язано з тим, що врахування замін однакової довжини може суттєво підвищити частоту цієї довжини заміни в розподілі літералів/довжин і, як наслідок, зменшити довжину її коду HUFF. З іншого боку, врахування сукупності коротких замін може суттєво зменшити частоти окремих літералів і тому збільшити довжини їх кодів HUFF.



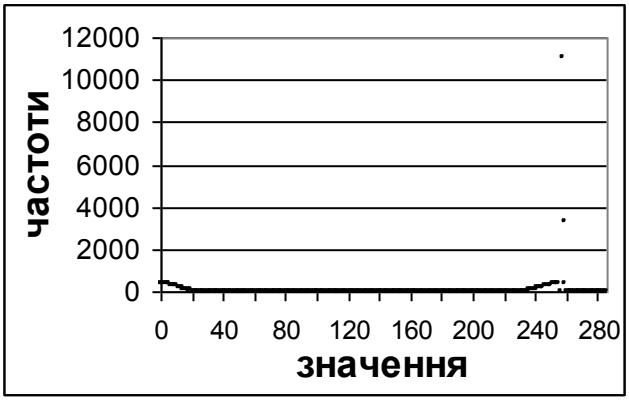
а)



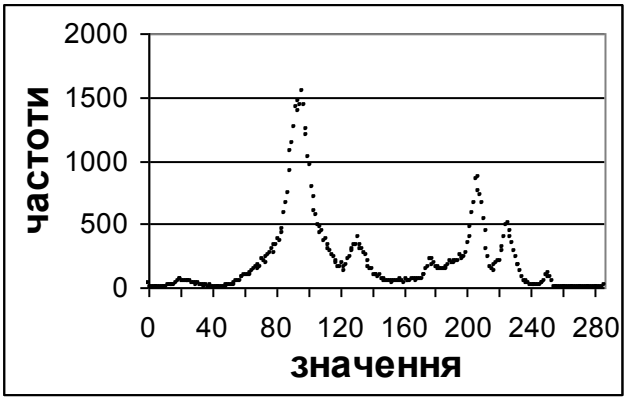
б)



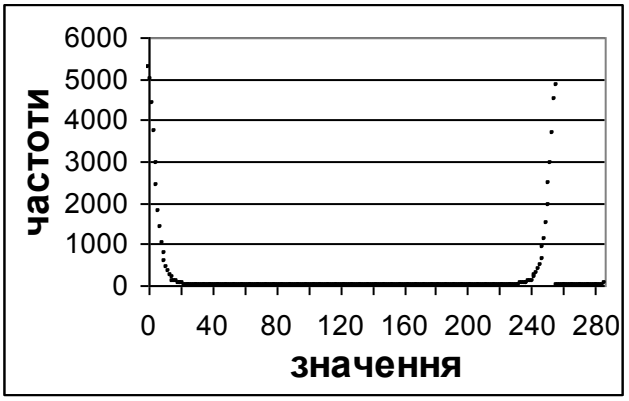
в)



д)



е)



ж)

Рис. 3.1. Розподіл частот елементів/довжин перших 64 Кб зображення *Lena.bmp*: а) без застосування предикторів та замін; б) після застосування предиктора Піфа; в) після застосування замін; д) після застосування предиктора Піфа та замін; е) після застосування замін від 9 елементів; ж) після застосування предиктора Піфа та замін від 9 елементів

Для різних блоків даних одного зображення найкоротшими можуть виявитися стиснуті блоки з відмінними максимальними довжинами замін, що переважно не враховуються – в основному це залежить від характеру розподілу частот їх літералів: якщо блок даних має більшу нерівномірність розподілу, то для нього, як правило, ця максимальна довжина виявляється теж більшою. Крім цього, на довжини стиснутих блоків суттєво впливають зміщення між однаковими фрагментами вхідного блоку даних, адже більші зміщення кодуються більшою кількістю додаткових бітів.

Як відомо, у форматі DEFLATE мінімальна довжина кодів HUFF рівна 1, максимальна – 15, максимальна кількість додаткових бітів довжини складає 5, а зміщення – 13 [62; 25, с. 288-289, 295-296]. Тому, згідно (3.1), для будь-яких блоків даних завжди ефективними будуть заміни послідовностей довжиною від 48 елементів. На практиці зображення, в яких окреме значення яскравості компонентів після застосування предикторів повторюється частіше від інших значень яскравостей разом узятих, зустрічаються надзвичайно рідко (хіба що однотонні заливки), як наслідок – довжини кодів літералів майже завжди перевищують 1, тому ефективними завжди будемо вважати заміни послідовностей, не коротші 24 елементів (навіть зменшення цього параметра до 12 елементів, яке ми застосували під час експериментів, збільшує розмір стиснутого файла не більше, ніж на 1 Кб). Ефективність же коротших замін, як слідує зі сказаного вище, залежить не лише від літералів, які вони замінюють, а й загалом від розподілів частот відповідного стиснутого блоку. Для чергового фрагменту зображення визначити наперед характер розподілу частот його літералів/довжин і, тим більше, зміщень, що забезпечить мінімальну довжину стиснутого блоку, неможливо. Тому **для кожного вхідного блоку даних ми пропонуємо наступний алгоритм мінімізації розміру відповідного стиснутого блоку:**

1. Створити альтернативні стиснуті блоки (чи розрахувати частоти елементів їх розподілів та кількості додаткових бітів) з різними максимальними довжинами неврахованих замін.

2. Обрати серед них найкоротший блок.

3. Ітеративно зменшити його довжину.

4. Використати сформований блок для зберігання даних у форматі DEFLATE PNG-файла.

Програми, що сьогодні використовуються для стиснення зображень у форматі PNG, або враховують всі заміни, або відкидають найкоротші заміни для всіх блоків. Ми ж, по суті, пропонуємо визначати максимальну довжину замін, що переважно не враховуються, для кожного блоку даних окремо так, щоб мінімізувати довжину відповідного стиснутого блоку.

Розглянемо тепер практичні аспекти реалізації лише перших трьох кроків цього алгоритму, оскільки четвертий крок полягає у звичайному записі сформованих кодів у вихідний файл:

1. Очевидно, що генерувати альтернативні стиснуті блоки доцільно лише для тих максимальних довжин неврахованих замін, які зустрічаються в розкладі LZ77 чергового блоку даних і можуть виявитися неефективними, тобто не перевищують 24. Враховуючи те, що довжина кожного стиснутого блоку складається з суми довжин закодованих літералів/довжин замін і зміщень та додаткових бітів, загальну довжину альтернативного блоку з максимальною довжиною неврахованих замін *j* для кожного блоку даних будемо розраховувати за формулою:

$$L_j = \sum_{i=0}^{255} n_{ij} l_{ij} + l_{256,j} + \sum_{i=257}^{285} n_{ij} (l_{ij} + d_i) + \sum_{i=0}^{29} \eta_{ij} (d_{ij} + \delta_i), \qquad (3.2)$$

де n_{ij} – частота елемента чи базової довжини *i*, а η_{ij} – частота базового зміщення *i* у випадку максимальної довжини неврахованих замін *j*. Як видно з цієї формули, для обчислення довжини альтернативного стиснутого блоку не потрібно зберігати сам блок. Досить знати лише частоти елементів розподілів альтернативного стиснутого блоку та згенерувати коди HUFF згідно цих розподілів. На практиці достатньо зберігати частоти розподілів літералів/довжин та зміщень з усіма врахованими замінами та перелік замін, що можуть виявитися неефективними, а для аналізу довжин інших альтернативних стиснутих блоків відкидати з цих розподілів частоти замін до визначеної довжини і враховувати відповідні літерали.

2. Визначати максимальну довжину неврахованих замін (а, отже, і індекс) найкоротшого стиснутого блоку з альтернативних будемо за формулою

$$indexMinBlock = \min\left\{k \mid L_k = \min_{j=2,24} L_j\right\}. \quad (3.3)$$

Прискорити визначення найкоротшого з альтернативних стиснутих блоків можна шляхом перебору максимальних довжин відкинутих замін не до 24, а лише до 12, адже довгі заміни, по-перше, зустрічаються доволі рідко, і, по-друге, неефективні довгі заміни можуть бути ліквідовані на третьому кроці алгоритму. Обчислювати довжину кожного альтернативного стиснутого блоку згідно (3.2) можна безпосередньо, після генерації відповідних кодів HUFF. Пришвидшити виконання цих розрахунків можна за допомогою окремого зберігання кількості додаткових бітів замін (оскільки вони не залежать від розподілів частот) та обчислення розміру блоків кодів HUFF літералів/довжин і зміщень лише за абсолютними частотами їх елементів, як це описано у підрозділі 2.2.

Після визначення максимальної довжини неврахованих замін, тобто найкоротшого з альтернативних стиснутих блоків, для подальшого зменшення його розміру необхідно спочатку розрахувати довжини кодів HUFF окремих елементів розподілів літералів/довжин та зміщень цього блоку. З цією метою потрібно:

а) згенерувати частоти розподілів літералів/довжин та зміщень найкоротшого з альтернативних стиснутих блоків, зменшуючи у частотах розподілів з усіма замінами для замін, що не враховуються (тобто довжина яких не перевищує визначену максимальну довжину неврахованих замін), частоти базових значень довжин та зміщень і збільшуючи частоти відповідних їм літералів та коригуючи кількість додаткових бітів;

б) на основі сформованих частот розподілів розрахувати довжини кодів HUFF літералів/довжин і зміщень;

в) присвоїти елементам розподілів, що не використовуються, довжину коду HUFF елемента з одиничною частотою, тобто максимальну довжину, оскільки такі елементи можуть з'явитися під час подальшого зменшення розміру найкоротшого з

альтернативних блоків.

3. Зменшити довжину обраного стиснутого блоку можна за рахунок відкидання неефективних врахованих та врахування ефективних відкинутих замін згідно (3.1). Але врахування ефективних замін зменшує частоти окремих елементів і, відповідно, може збільшити після перерахунку довжини їх кодів HUFF. Тому серед замін, неефективних з попередніми кодами HUFF, можуть виявитися ефективні з перерахованими такими кодами. І навпаки: відкидання неефективних замін збільшує частоти окремих елементів та, відповідно, може зменшити після перерахунку довжини їх кодів HUFF, тому серед замін, ефективних з попередніми кодами HUFF, можуть виявитися неефективні з перерахованими такими кодами. Ось чому для обраного найкоротшого блоку з альтернативних доцільно серед замін, коротших 24 літералів, спочатку ітеративно відкидати неефективні, а потім, якщо цей блок не враховує всі заміни, – ітеративно враховувати ефективні заміни (чи навпаки), доки кількість таких замін перевищує порогове значення (наприклад, 5), щоразу перераховуючи при цьому довжини кодів HUFF елементів розподілів літералів/довжин та зміщень. Максимально прискорити виконання цього кроку алгоритму можливо, якщо після першого відкидання неефективних замін виконати лише перерахунок довжин кодів розподілів і одну сумісну ітерацію для аналізу ефективності замін, коротших 24 літералів, та й то у випадку, коли найкоротший блок з альтернативних не враховує всі заміни, хоча така модифікація алгоритму і призводить до зростання розмірів стиснутих файлів на десятки байтів. Фрагмент програми мовою С, що реалізує три розглянуті кроки алгоритму, наведено у першому розділі додатка Ж.

Під час попереднього аналізу зображень достатньо виконати лише два перші кроки алгоритму, оскільки третій крок незначно зменшує розмір стиснутого блоку, вимагає зберігання даних замін, а не лише їх статистики та значно сповільнює процес стиснення. Всі ж кроки алгоритму доцільно виконувати лише в процесі безпосереднього кодування перед записом стиснутих блоків у PNG-файл. Приклад розподілу частот літералів/довжин проаналізованого вище фрагменту зображення після застосування алгоритму мінімізації розміру стиснутого блоку наведено на

рис. 3.2. Співставляючи його з розподілами частот рис. 3.1, приходимо до висновку, що найменший розмір стиснутого блоку для цього фрагменту зображення серед розглянутих варіантів забезпечує застосування предиктора Піфа та врахування замін LZ77 від 9 елементів включно.

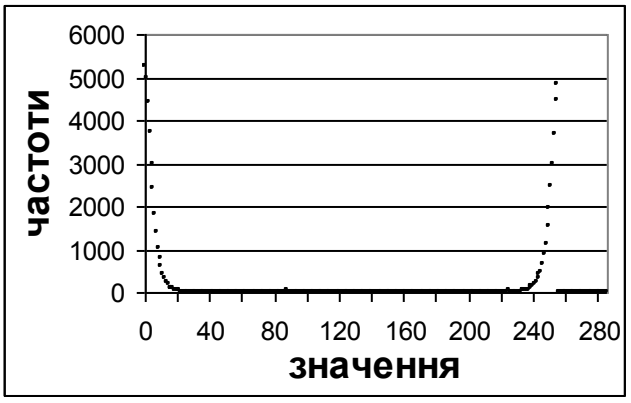


Рис. 3.2. Розподіл частот елементів/довжин перших 64 Кб зображення *Lena.bmp* після застосування алгоритму мінімізації розміру стиснутих блоків

Розглянемо **результати застосування** описаного алгоритму мінімізації розміру стиснутих блоків для компресії зображень набору ACT (див. табл. 1.3). Результати тестування наведено в табл. 3.1 та 3.2.

Як свідчать результати тестування, застосування описаного алгоритму покращило КС на 2 – 6 % для 63 % зображень, які є фотореалістичними з наявними неефективними короткими замінами чи синтезованими з шумами, та не вплинуло на компресію решти файлів, хоча й сповільнило кодування в середньому на 10 %. Використання предиктора *PaethPredict* виявилося недоцільним для синтезованих зображень, оскільки цей предиктор (як і інші) частково зменшує довжини однакових послідовностей.

Наголосимо, що цей алгоритм дає змогу суттєво покращити КС більшості фотореалістичних зображень за рахунок вибору для **кожного** блоку даних найкоротшого з альтернативних стиснутих блоків та ітеративного зменшення його розміру.

Таблиця 3.1

КС файлів зображень набору ACT у форматі PNG після застосування різних програм, %

Назва програми	№ файла								Середній КС
	1	2	3	4	5	6	7	8	
Photo Editor 2000	37.74	7.43	95.19	76.50	91.55	86.38	7.65	90.03	61.56
Miano без алгоритму мінімізації	23.80	9.77	67.75	55.68	58.65	67.65	9.97	61.14	44.30
Miano з алгоритмом мінімізації	20.99	9.77	61.25	52.91	53.97	67.65	9.97	57.85	41.79

Таблиця 3.2

Час кодування файлів зображень набору ACT у формат PNG різними програмами, с

Назва програми	№ файла								Середній час
	1	2	3	4	5	6	7	8	
Photo Editor 2000	2.27	2.48	1.39	1.88	1.48	2.15	1.40	1.92	1.87
Miano без алгоритму мінімізації	10.11	14.56	4.29	8.19	5.00	6.65	6.04	7.36	7.78
Miano з алгоритмом мінімізації	11.20	15.05	5.06	9.33	5.77	7.36	6.16	8.56	8.56

3.2. Організація вибору предикторів для рядків пікселів

3.2.1. Аналіз впливу однакових предикторів для всіх рядків на коефіцієнти стиснення зображень. Як показано в підрозділі 1.1, алгоритми, що використовуються у форматі PNG для стиснення даних, орієнтовані на зменшення різних видів надлишковості: якщо розклад LZ77 усуває міжелементну надлишковість між однаковими фрагментами даних, то кодування HUFF орієнтоване на зменшення кодової надлишковості між переважаючими елементами розподілів частот. Використання ж різних предикторів формату PNG хоча й не виконує безпосереднє стиснення, але підвищує кодову надлишковість за рахунок зменшення міжелементної і, як наслідок, неоднаково впливає на КС цих алгоритмів. Кодування HUFF у межах стиснутого блоку DEFLATE не збільшує довжину закодованих даних (див. пункт 1.1.2). Тому дослідимо спочатку вплив різних

предикторів (див. пункт 1.1.3) та стиснення HUFF на показники ефективності кодування у стандартному та модифікованому форматі PNG для зображень набору ACT (див. табл. 1.3) без застосування алгоритму LZ77 (табл. 3.3, 3.4).

Таблиця 3.3

КС файлів зображень набору ACT після застосування різних предикторів без використання алгоритму LZ77, %

Предиктор	№ файла								Середній КС
	1	2	3	4	5	6	7	8	
NonePredict (без предикторів)	92.24	53.31	95.97	92.45	94.41	90.98	65.23	94.88	84.94
LeftPredict	45.03	24.82	65.93	56.98	57.87	71.12	25.27	64.44	51.43
AbovePredict	46.36	29.18	61.77	57.33	59.30	74.93	26.09	62.45	52.18
AveragePredict	58.69	36.03	61.12	53.25	55.27	70.16	32.92	59.50	53.37
PaethPredict	**27.61**	**23.47**	61.64	53.08	53.97	69.04	**22.06**	58.02	46.11
MedPredict	**27.61**	23.74	**60.86**	**51.60**	**52.93**	**68.17**	22.27	**56.63**	**45.48**

Таблиця 3.4

Час кодування файлів зображень набору ACT внаслідок застосування різних предикторів без використання алгоритму LZ77, с

Предиктор	№ файла								Середній час
	1	2	3	4	5	6	7	8	
NonePredict (без предикторів)	9.17	13.40	3.46	5.05	3.41	5.00	5.49	5.11	6.26
LeftPredict	8.19	12.96	3.30	4.72	3.13	5.05	5.21	4.89	5.93
AbovePredict	8.13	12.90	3.13	4.62	3.13	5.00	5.16	4.84	5.86
AveragePredict	8.90	14.06	3.30	4.73	3.19	5.11	5.60	4.95	6.23
PaethPredict	8.74	14.61	3.57	5.16	3.46	5.44	5.82	5.27	6.51
MedPredict	8.35	14.01	3.46	4.94	3.30	5.33	5.60	5.06	6.26

Як і слід було чекати, без застосування алгоритму LZ77 найменші КС

зображень забезпечують нелінійні предиктори *PaethPredict* та *MedPredict*, (відповідні розміри файлів виділені у табл. 3.3 напівжирним шрифтом), оскільки вони максимально збільшують нерівномірність розподілу літералів/довжин. Причому *PaethPredict* найкраще прогнозує яскравості компонентів пікселів синтезованих зображень, а *MedPredict* – фотореалістичних. Час кодування у цьому випадку в основному залежить від розміру вхідного файла та складності розрахунку застосованого предиктора.

Проаналізуємо також вплив цих самих предикторів та кодування HUFF на показники ефективності стиснення у випадку використання "жадібного" розкладу алгоритму LZ77 (табл. 3.5, 3.6). У форматі PNG алгоритм LZ77 може стискувати дані максимум у 129, а кодування HUFF – максимум у 8 разів. Саме тому алгоритм LZ77 і використовується у цьому форматі, хоча він зменшує нерівномірність розподілу частот між окремими елементами і тому може збільшити КС кодування HUFF. Алгоритм мінімізації розміру стиснутого блоку дає змогу, по суті, залишити лише ті з замін алгоритму LZ77, що забезпечують менший КС, ніж кодування HUFF для відповідних літералів. Оскільки алгоритм LZ77 усуває надлишковості між однаковими яскравостями чи приростами яскравостей, то він ефективний, насамперед, для синтезованих зображень (див. КС зображень №№ 1, 2, 7 у табл. 3.3 і 3.5). З іншого боку, у фотореалістичних зображеннях яскравості компонентів пікселів, як правило, близькі, але не однакові. Тому довгі однакові послідовності яскравостей чи їх приростів в даних таких зображень практично відсутні, а отже використання для них алгоритму LZ77 є малоефективним і кодування HUFF у цьому випадку забезпечує хоча б задовільні КС (див. результати стиснення решти зображень у цих же таблицях). Таким чином, ключову роль в процесі стиснення різних зображень і навіть окремих їх фрагментів можуть відігравати як алгоритм LZ77 у поєднанні з алгоритмом мінімізації розміру кожного стиснутого блоку, так і кодування HUFF. Ось чому, по-перше, **всі ці три алгоритми потрібно застосовувати в процесі стиснення кожного зображення у форматі PNG**, і, по-друге, статичні предиктори, що використовують під час попередньої обробки зображень перед стисненням, не повинні негативно впливати на властивості потоку,

що використовуються основним алгоритмом стиснення (наприклад, для словникових методів предиктори не повинні зменшувати кількість однакових послідовностей).

Таблиця 3.5

КС файлів зображень набору АСТ після застосування різних предикторів та "жадібного" розкладу алгоритму LZ77, %

Предиктор	№ файла								Середній КС
	1	2	3	4	5	6	7	8	
NonePredict (без предикторів)	23.89	**6.85**	91.29	70.42	84.53	77.54	**7.17**	83.09	55.60
LeftPredict	21.28	9.14	65.28	55.94	57.35	68.78	9.29	64.01	43.88
AbovePredict	22.89	9.66	61.38	56.20	59.30	70.08	9.90	61.75	43.90
AveragePredict	53.55	14.25	60.86	53.08	55.14	66.18	14.75	59.32	47.14
PaethPredict	**20.99**	9.77	61.25	52.91	53.97	67.65	9.97	57.85	41.79
MedPredict	**20.99**	9.28	**60.47**	**51.43**	**52.93**	**65.31**	10.31	**56.46**	**40.90**

Таблиця 3.6

Час кодування файлів зображень набору АСТ внаслідок застосування різних предикторів та "жадібного" розкладу алгоритму LZ77, с

Предиктор	№ файла								Середній час
	1	2	3	4	5	6	7	8	
NonePredict (без предикторів)	8.08	12.63	4.45	6.15	4.40	6.59	5.16	6.48	6.74
LeftPredict	11.98	13.29	4.78	8.13	5.44	6.86	5.49	7.52	7.94
AbovePredict	7.80	13.68	4.78	8.13	5.11	6.92	5.55	7.69	7.46
AveragePredict	14.34	15.21	5.00	9.56	5.77	7.25	6.15	8.24	8.94
PaethPredict	11.20	15.05	5.06	9.33	5.77	7.36	6.16	8.56	8.56
MedPredict	10.60	14.66	4.95	9.34	5.88	7.25	5.93	8.62	8.40

Проаналізуємо далі ефективність використання окремих предикторів у випадку застосування даних алгоритмів (див. табл. 3.5, 3.6). Стиснення зображень у

форматі PNG без застосування предикторів відбувається за рахунок повторень та наявності нерівномірності розподілу частот значень яскравостей пікселів. Але в фотореалістичних зображеннях аналогічні фрагменти та однакові значення яскравостей компонент трапляються порівняно рідко, тому й таке стиснення призводить до високих КС (див. рис. 1.5 а та варіант *NonePredict* у табл. 3.5). Стиснення зображень без застосування предикторів ефективне, насамперед, для синтезованих зображень з високою роздільною здатністю. Це стиснення виконується найшвидше, оскільки не вимагає виконання додаткових обчислень значень предикторів та не генерує багато коротких однакових послідовностей для алгоритму LZ77.

Предиктор *LeftPredict* продукує горизонтальні прирости яскравостей між компонентами суміжних пікселів. Однакові прирости яскравостей трапляються у фотореалістичних зображеннях, як правило, частіше, ніж однакові фрагменти, тому застосування предикторів покращує для них КС алгоритму LZ77. Крім цього, застосування предиктора *LeftPredict* підвищує нерівномірність розподілу частот навколо нуля (див. рис. 1.5 б), зменшуючи цим самим КС кодування HUFF. Саме тому середній КС зображень набору ACT із застосуванням *LeftPredict* зменшився порівняно з варіантом без застосування предикторів на 11.82 %. Але підкреслимо, що при застосуванні *LeftPredict* можливе зменшення довжин повторень, які були між фрагментами оригіналу зображення, тому можливе і підвищення загального КС (дані зображень №№ 2, 7 в рядку *LeftPredict* у табл. 3.5). Отже, однозначно стверджувати, що стиснення з використанням *LeftPredict* ефективніше від стиснення без застосування предикторів неможливо. Середня тривалість компресії файлів набору ACT цим способом перевищує тривалість кодування без предикторів на 17.7 %, оскільки застосування предикторів вимагає розрахунку їх значень і породжує більше коротких однакових послідовностей, що сповільнює формування розкладу LZ77.

Аналогічно впливає на стиснення зображень і застосування предиктора *RightPredict*, який генерує вертикальні прирости яскравостей компонентів суміжних пікселів і описує, як і попередній предиктор, зміну форми зображених об'єктів у

обраному напрямку. Різні зображення мають різний рівень відмінностей пікселів по горизонталі та вертикалі. Саме тому КС в рядках *LeftPredict* та *RightPredict* табл. 3.5 різні. Цим самим пояснюється різниця в часі кодування на 53.58 % зображення № 1 для даних рядків у табл. 3.6, оскільки це зображення містить суттєво меншу кількість однакових приростів яскравостей компонент пікселів у вертикальному напрямку відносно горизонтального, що прискорює формування розкладу LZ77, але й підвищує КС. Крім цього, гірший КС з використанням предиктора *RightPredict* стосовно *LeftPredict* для більшої половини зображень набору пояснюється тим, що однакові прирости по вертикалі зустрічаються, як правило, в суміжних рядках. Зміщення ж між суміжними пікселами по вертикалі при обробці даних у форматі PNG дорівнює довжині рядка. Тобто вони значно більші від зміщень між суміжними пікселами по горизонталі і тому кодуються значно більшою кількістю додаткових бітів.

Предиктор *AveragePredict* врівноважує вплив суміжних пікселів по горизонталі та вертикалі. Він добре (відносно розглянутих вище варіантів) прогнозує яскравості компонент пікселів для зображень великих та розмитих фотореалістичних об'єктів, хоча й дуже чутливий до різких перепадів кольорів (наприклад, ліній променів світла у зображенні № 1) та чітких меж об'єктів синтезованих зображень. Сповільнення кодування зображень з використанням *AveragePredict* пояснюється як необхідністю виконання операції ділення чи бітового зсуву для розрахунку значення предиктора кожної компоненти всіх пікселів, так і низькою відносною швидкістю формування розкладу LZ77, під час формування якого необхідно опрацьовувати багато коротких однакових послідовностей.

Найкращі в середньому передбачення значень яскравостей компонентів пікселів по набору ACT у форматі PNG генерує предиктор *PaethPredict*, який прогнозує значення у напрямку найменшого приросту, але часто зменшує довжини повторень, що були між фрагментами оригіналу зображення. Цей предиктор на синтезованих зображеннях поступається всім розглянутим предикторам крім *AveragePredict*, а на фотореалістичних зображеннях забезпечує в середньому кращі КС відносно інших предикторів. Тому для забезпечення найшвидшого кодування у

форматі PNG доцільно використовувати саме предиктор *PaethPredict*.

Предиктор *MedPredict* забезпечує для фотореалістичних зображень КС у середньому на 1.6 % менші, ніж предиктор *PaethPredict*, витрачаючи для кодування приблизно стільки ж часу. Цей предиктор не входить у діючий стандарт формату PNG [62], тому ми повернемося до нього лише у наступному розділі.

Як бачимо, застосування предикторів зменшує КС фотореалістичних зображень на понад 9.9 %, що вказує на необхідність їх використання. Крім цього, застосування нелінійних еволюційно-шумових предикторів в цілому забезпечує краще стиснення від лінійних шумових на понад 2 %, хоча й вимагає виконання більшої кількості обчислень. З іншого боку, як свідчать результати експериментів, **для кожного рядка зображення слід обирати той предиктор, який дозволить підсилити дію ключового алгоритму стиснення чергового фрагменту** (LZ77 чи HUFF) і цим допоможе забезпечити найменший загальний КС. Ось чому не існує універсального предиктора, застосування якого сприяло б найкращому стисненню всіх типів зображень. Навіть для їх суміжних фрагментів оптимальними можуть виявитися різні предиктори. Тому для досягнення кращих КС **необхідно використовувати різні предиктори для різних рядків зображення.** Зрозуміло що комбінування предикторів сповільнює кодування, але воно дає змогу не лише зменшити розміри файлів зображень, а й за рахунок можливого використання лінійних предикторів замість нелінійних прискорити, хоча й незначно, декодування.

3.2.2. Ентропійні способи вибору предикторів для рядків пікселів. Для розробки ефективних способів вибору предикторів рядків узагальнимо, виходячи з попереднього аналізу, вплив різних предикторів на продуктивність базових алгоритмів стиснення формату PNG. Алгоритм LZ77 суттєво зменшує КС зображень без застосування предикторів або після дії предикторів *LeftPredict* чи *RightPredict*. Кодування ж HUFF краще стискає послідовності елементів зображень після дії предикторів (особливо після *AveragePredict* чи *PaethPredict*) як безпосередньо, так і після виконання коротких замін LZ77 (див. рис. 3.1), оскільки вони найефективніше підвищують нерівномірність розподілу частот (яка оцінюється ентропією) і, відповідно, збільшують кодову надлишковість. Обрати найкращий з трьох варіантів

використання предикторів стосовно алгоритму LZ77, що зменшує міжелементну надлишковість, для кожного рядка пікселів неможливо без виконання трьох попередніх розкладів зображення зі застосуванням кожного з цих варіантів, оскільки однакові фрагменти даних можуть зустрічатися в різних рядках. А **найкращий предиктор для окремих рядків пікселів стосовно кодування HUFF обиратимемо так, щоб забезпечити мінімальну прогнозовану довжину відповідного стиснутого блоку у форматі DEFLATE як з врахуванням так і без застосування лише коротких замін LZ77**, враховуючи, що довжина стиснутого блоку у цьому форматі складається з довжин блоків кодів HUFF для послідовностей літералів/довжин та зміщень, а також з додаткових бітів довжин та зміщень. Обчислювати довжину кожного блоку кодів HUFF будемо наближено за допомогою довжини ентропійного коду (2.1), оскільки точний підрахунок цієї довжини (див. підрозділ 2.2) хоча й можливий, але вимагає більше часу і, як показали експерименти, суттєво не впливає на вибір предиктора.

Розглянемо далі три способи вибору предиктора для окремих рядків пікселів, що мінімізують прогнозовану довжину коду у форматі DEFLATE стосовно кодування HUFF.

3.2.2.1. Безпосередній ентропійний спосіб полягає у виборі предиктора, що породжує для рядка пікселів після свого застосування мінімальну довжину ентропійного коду (2.1), тобто забезпечує мінімальний прогнозований КС без врахування замін алгоритму LZ77. Фрагмент програми для визначення номера предиктора, що породжує дані з найменшою ентропією елементів (надалі – *безпосереднього ентропійного предиктора*) записується, наприклад, так:

```
for (i=0; i<=4; ++i) // цикл по предикторах
 {memset(freq, 0, sizeof(freq));
  for (j=0; j<row_width; ++j) // цикл по елементах рядка
   freq[buffers[i][j]]++;//накопичення частот результатів і-го предиктора
  len=lenEntropiCode(freq);
  if (len<minLen) {predict1=i; minLen=len; }}
KC1=(double)minLen/(8*row_width);      .
```

3.2.2.2. Ентропійний спосіб після застосування коротких замін алгоритму LZ77 зводиться до вибору предиктора, який генерує для рядка пікселів послідовність, що найкраще стискається у форматі DEFLATE після виконання замін

алгоритму LZ77 з декількох елементів (надалі – *ентропійний предиктор коротких замін LZ77*). Адже короткі заміни, як показано в підрозділі 3.1, можуть суттєво збільшити частоти довжин замін, що їм відповідають, та зменшити частоти окремих літералів. А це, в свою чергу, може призвести до збільшення нерівномірності розподілу частот елементів і, відповідно, до зменшення КС (з урахуванням довжин кодів зміщень та додаткових бітів). Виконувати попереднє стиснення LZ77 для результатів дії всіх предикторів рядків недоцільно, оскільки це призводить до різкого сповільнення процесу кодування. Тому ми оцінювали КС рядка пікселів після застосування триелементних замін за допомогою хеш-таблиці по чотирьох останніх бітах трьох суміжних елементів, в якій зберігаються абсолютні позиції останніх входжень подібних груп. Використання значень цієї таблиці дає змогу імітувати розклад LZ77 з підрахунком частот літералів, триелементних замін та їх зміщень і додаткових бітів, після чого прогнозований КС рядка визначається як відношення суми кількостей додаткових бітів та розмірів ентропійних кодів літералів/триелементних замін і зміщень (розраховуються за частотами їх елементів згідно (2.1)) до кількості бітів рядка. Фрагмент програми мовою С для визначення номера предиктора, що породжує дані з найменшим КС після імітації коротких замін LZ77, наведений у другому розділі додатка Ж.

3.2.2.3. Комбінований ентропійний спосіб. Для різних зображень і навіть для різних фрагментів одного зображення короткі заміни LZ77 можуть виявитися як ефективними (тобто кількість бітів для їх зберігання є не більшою від кількості бітів для зберігання відповідних їм окремих літералів) так і неефективними, а передбачити наперед необхідність імітації коротких замін неможливо. Тому для цілого зображення (чи для чергового блоку рядків, як буде показано в підрозділі 3.3) доцільно обирати предиктори, що забезпечують прогнозований мінімальний КС. Отримувати цей КС будемо за результатами порівняння середнього КС рядків після використання безпосередніх ентропійних предикторів (в попередніх фрагментах програм – *КС1*) та середнього КС рядків (*КС2*) після застосування ентропійних предикторів коротких замін LZ77. Покроково алгоритм реалізації цього способу вибору предикторів для зображення або чергового блоку рядків записується так:

1. Визначити для кожного рядка та зберегти в першому масиві номери безпосередніх ентропійних предикторів. Обчислити їх середній КС (*averageKC1*).

2. Визначити для кожного рядка та зберегти в другому масиві номери ентропійних предикторів коротких замін LZ77. Обчислити їх середній КС (*averageKC2*).

3. Якщо *averageKC1*<*averageKC2*, то використати для подальшого стиснення номери предикторів рядків з першого масиву, інакше – застосувати з цією метою номери предикторів рядків з другого масиву.

Визначення безпосередніх ентропійних предикторів та ентропійних предикторів коротких замін LZ77 не передбачає формування повного розкладу даного алгоритму і тому не враховує вплив його довгих замін на КС. Ось чому вибір за наведеним алгоритмом масиву номерів предикторів рядків для подальшого стиснення може виявитися помилковим. Тому у підрозділі 3.5 ми запропонуємо ефективніший алгоритм вибору між безпосередніми ентропійними предикторами та ентропійними предикторами коротких замін LZ77 (і не тільки), що враховує довгі заміни алгоритму LZ77 під час визначення прогнозованого розміру стиснутого зображення.

3.2.3. Емпіричні способи вибору предикторів для рядків пікселів. Розглянуті ентропійні способи вибору предикторів для окремих рядків пікселів, що орієнтуються на мінімальну прогнозовану довжину коду у форматі DEFLATE, спрямовані, насамперед, на зменшення кодової надлишковості. Вони не беруть до уваги повною мірою існування міжелементної надлишковості між різними фрагментами, яка переважає у синтезованих зображеннях та зменшується внаслідок дії алгоритму LZ77. Тому розглянемо далі два емпіричні способи вибору предикторів для окремих рядків пікселів, що частково враховують результати розкладу цього алгоритму.

3.2.3.1. Комбінований ентропійний спосіб з додатковим попіксельним розкладом алгоритму LZ77 оригіналу зображення. Як свідчать експерименти, міжелементна надлишковість максимально проявляється між фрагментами зображень, якщо предиктори до них не застосовуються (див. рядок *NonePredict* у

табл. 3.3) і виявляється меншою мірою у випадку використання *LeftPredict* чи *RightPredict*. Тому обирати предиктор для рядка пікселів ентропійним способом доцільно лише тоді, коли стиснення LZ77 зі застосуванням довгих замін для його оригіналу неефективне. Як зазначалося вище, визначити ефективність стиснення для окремих рядків зображення алгоритмом LZ77 неможливо, оскільки однакові послідовності елементів можуть зустрічатися і в різних рядках, тому проаналізуємо її за результатами додаткового розкладу цим алгоритмом всього зображення без застосування предикторів. Ми вважатимемо стиснення рядка зображення алгоритмом LZ77 ефективним, якщо, по-перше, за результатами даного розкладу для понад 75 % елементів цього рядка вдається віднайти однакові послідовності, не коротші 9 елементів або даний рядок знаходиться між двома рядками, для яких стиснення LZ77 теж ефективне, і, по-друге, якщо цей рядок разом, можливо, з іншими суміжними рядками, для яких стиснення LZ77 теж ефективне, містить не менше 8192 елементів (25 % розміру словника LZ77). Для прискорення визначення ефективності стиснення рядків зображення алгоритмом LZ77 ми використали попіксельний "жадібний" розклад, який формується втричі швидше від поелементного. Алгоритм реалізації цього способу вибору предикторів для окремих рядків пікселів покроково записується так:

1. Виконати попередній попіксельний "жадібний" розклад оригіналу зображення з підрахунком кількостей елементів кожного рядка, для яких вдається віднайти однакові послідовності, не коротші 9 елементів.

2. Встановити ознаку ефективності стиснення алгоритмом LZ77 для рядків, що містять понад 75 % таких елементів.

3. Встановити ознаку ефективності стиснення алгоритмом LZ77 для рядків, що знаходяться між двома рядками, для яких стиснення цим алгоритмом теж ефективне.

4. Зняти ознаку ефективності стиснення алгоритмом LZ77 для блоків рядків зі встановленою такою ознакою, які містять менше 8192 елементів.

5. Встановити рядкам пікселів зображення, для яких алгоритм LZ77 виявився ефективним ознаку стиснення предиктором *NonePredict* (тобто встановити ознаку

незастосування предикторів, присвоївши номеру їх предиктора значення 0).

6. Вибрати предиктор для інших рядків комбінованим ентропійним способом, як описано у підпункті 3.2.2.3.

Зрозуміло, що цей спосіб вибору предикторів для окремих рядків пікселів не може були найкращим для всіх зображень, оскільки містить фіксовані числові критерії визначення ефективності алгоритму LZ77. Але, як показують результати експериментів наступного пункту, він забезпечує в середньому менші КС від ентропійних способів, оскільки краще враховує міжелементну надлишковість. Даний спосіб вибору предикторів для рядків пікселів використано, наприклад, у тестових програмах, що застосовувалися для дослідження ефективності алгоритмів зменшення розмірів стиснутих блоків [53; 52; 49]. Фрагмент програми для визначення ефективності стиснення окремих рядків алгоритмом LZ77 наведено у третьому розділі додатка Ж.

3.2.3.2. Спосіб з прогнозуванням результатів розкладів алгоритму LZ77 полягає у виборі предиктора, що забезпечує прогнозований мінімальний КС рядка у форматі DEFLATE після свого застосування з врахуванням словника цього алгоритму [4]. Іншими словами, для кожного рядка пікселів серед всіх предикторів обиратимемо той, що генерує з його елементів послідовність з найменшою прогнозованою довжиною коду у цьому форматі. Цей спосіб не вимагає попереднього стиснення опрацьованих предикторами елементів рядків. Він лише оцінює кількість бітів, що знадобиться для стиснення при використанні кожного з предикторів. Згідно стандарту PNG [62], в процесі стиснення алгоритмом LZ77 елементів рядка, опрацьованих обраним предиктором, використовується словник з елементів попередніх рядків після застосування обраних для них предикторів. Тому **прогнозування** елементів розкладу цього алгоритму для результатів дії кожного предиктора **виконаємо з використанням словника, що формується з елементів декількох попередніх рядків зображення, перетворених обраними для них предикторами**. Оцінювати довжину коду у форматі DEFLATE для результатів розкладу алгоритму LZ77 елементів чергового рядка після застосування кожного предиктора будемо так само, як і для ентропійного способу після застосування

коротких замін даного алгоритму (тобто, як суму довжин ентропійних кодів для послідовностей літералів/довжин та зміщень і кількостей додаткових бітів довжин та зміщень). На практиці, для прискорення формування цих розкладів, ми використали словник з елементів двох попередніх рядків, перетворених обраними для них предикторами. Мовою С фрагмент програми для вибору предиктора рядка цим способом записується, наприклад, так:

```
current_predict = pprPredict; // номер предиктора попереднього рядка
memcpy (buffers + 2*row_width, predict_buffers[current_predict], row_width);
// обчислюємо довжину коду чергового рядка з предиктором попереднього рядка
long longestrun = countBitPackBuffer(buffers, 3*row_width, 2*row_width);
// порівнюємо з результати дій інших предикторів
for (i = 0; i <= 4; ++i)
 {if (i == pprPredict) continue; // дію цього предиктора вже оцінено
  memcpy(buffers + 2*row_width, predict_buffers[i], row_width);
  long run = countBitPackBuffer(buffers, 3*row_width, 2*row_width);
  if (run < longestrun)
   {current_predict = i; longestrun = run; }}
// готуємося до аналізу дії предикторів для наступного рядка
pprPredict=current_predict; // зберігаємо номер обраного предиктора
// посуваємо коди попереднього і запам'ятовуємо коди активного рядка
memcpy(buffers, buffers+row_width, row_width);
memcpy(buffers+row_width, predict_buffers[current_predict], row_width);
// формуємо хеш-таблицю та хеш-ланцюги для нового словника
initHashPredict(2*row_width);   .
```

Реалізація функції *countBitPackBuffer*, що виконує "жадібний" розклад алгоритму LZ77 для елементів чергового рядка після дії кожного предиктора та обчислює прогнозовану довжину відповідного коду у форматі DEFLATE наведена у четвертому розділі додатка Ж.

Зрозуміло, що цей спосіб обирає предиктори окремих рядків значно повільніше від попереднього, оскільки виконує додаткові розклади LZ77 не лише для окремих рядків оригіналу зображення, а й для результатів дії кожного предиктора (5 розкладів для кожного рядка замість одного для цілого зображення). З цієї ж причини він дає змогу забезпечити менші КС окремих зображень. Досягнути ще менших КС зображень відносно цього способу можливо лише врахувавши, що вибір предиктора для рядка пікселів впливає на КС не лише цього, а й наступних рядків (що й буде зроблено в підрозділі 3.5), оскільки формує словник для їх розкладу алгоритмом LZ77.

3.2.4. Аналіз впливу різних варіантів вибору предикторів для рядків пікселів на коефіцієнти стиснення зображень. Проаналізуємо вплив різних варіантів вибору предикторів рядків пікселів, що розроблялися попередніми дослідниками та пропонуються в цій роботі, (табл. 3.7, 3.8) на ефективність стиснення зображень набору ACT (див. табл. 1.3).

Стандарт PNG [62] рекомендує обирати той предиктор для кожного рядка, застосування якого зумовлює мінімальну суму значень знакових байт (з діапазону [-128; 127]). Зменшення загальної суми значень рядка після використання предиктора свідчить про наближення розподілу частот до симетричного, але чи вказує цей критерій на зменшення ентропії джерела? Результати застосування такого способу вибору предиктора для кожного рядка дають негативну відповідь на це запитання (див. рядок *Варіант 1* в табл. 3.7).

Інший спосіб визначення оптимального предиктора для кожного рядка полягає у пошуку найбільшої кількості повторень однакових значень, що йдуть підряд (запропонований в [25, с. 317]). Повторення значень сприяє покращенню показників стиснення алгоритмом LZ77, але цей спосіб не враховує можливості повторень груп різних байтів, що також покращують стиснення алгоритмом LZ77, та змін ентропії джерела. Результати застосування такого способу вибору оптимального предиктора кожного рядка наведено у табл. 3.7, 3.8 у рядку *Варіант 2*.

В [24] та незалежно в [4] запропоновано обирати той предиктор, застосування якого зумовлює мінімальну суму **модулів** значень знакових байтів (з діапазону [-128; 127]). Зменшення загальної суми модулів значень рядка після використання предиктора свідчить про збільшення нерівномірності розподілу частот навколо нуля (див. рис. 1.5 б) та покращує результати застосування контекстно-незалежного алгоритму. Результати застосування такого способу вибору оптимального предиктора кожного рядка наведено у табл. 3.7, 3.8 у рядку *Варіант 3*. Такий спосіб вибору предиктора рядка покращує середній КС стосовно предиктора Піфа (див. табл. 3.5, 3.6) на 0.28 %, причому зменшення розмірів файлів спостерігається майже для всіх зображень, хоча він і сповільнює кодування на 26 %.

Таблиця 3.7

КС файлів зображень набору ACT у форматі PNG після застосування різних варіантів вибору предикторів, %

Назва варіанта	№ файла								Середній КС
	1	2	3	4	5	6	7	8	
Варіант 1	21.04	9.58	63.07	63.05	69.18	73.55	10.04	78.58	48.51
Варіант 2	20.99	9.75	90.64	55.94	57.48	69.12	10.04	61.67	46.95
Варіант 3	20.94	9.72	60.21	52.56	53.84	66.96	10.04	57.85	41.51
Варіант 4	20.89	9.61	60.21	52.56	53.84	66.96	10.04	57.76	41.48
Варіант 5	22.94	8.56	60.21	52.73	53.97	66.18	9.08	58.02	41.46
Варіант 6	22.94	8.56	60.21	52.56	53.84	66.18	9.08	57.76	41.39
Варіант 7	22.94	6.85	60.21	52.56	53.84	66.18	7.17	57.76	40.94
Варіант 8	20.89	7.18	60.47	52.91	56.44	66.61	9.49	58.89	41.61

Таблиця 3.8

Час кодування файлів зображень набору ACT у формат PNG програмами зі застосуванням різних варіантів вибору предикторів, с

Назва варіанта	№ файла								Середній час
	1	2	3	4	5	6	7	8	
Варіант 1	13.56	19.89	6.31	9.67	6.53	8.89	8.07	8.90	10.23
Варіант 2	14.39	21.86	5.99	10.43	6.86	9.17	8.79	9.89	10.92
Варіант 3	13.95	20.38	6.32	11.15	6.92	9.29	8.29	10.21	10.81
Варіант 4	14.28	20.49	6.43	11.37	7.09	9.34	8.34	10.33	10.96
Варіант 5	15.05	22.74	7.09	12.30	7.74	10.33	9.28	11.32	11.98
Варіант 6	17.14	25.22	7.90	13.52	8.79	11.32	10.44	12.74	13.38
Варіант 7	18.23	18.62	8.46	15.05	9.45	12.74	8.35	13.73	13.08
Варіант 8	357.01	464.73	163.18	246.50	133.91	301.76	138.80	289.29	261.90

Середній КС зображень набору ACT зі застосуванням безпосереднього ентропійного способу вибору предикторів рядків зменшився в порівнянні з способом вибору предикторів на основі найменшої суми модулів значень знакових

байтів на 0.03 % (рядок *Варіант 4* у табл. 3.7, 3.8), причому розмір стиснутих файлів не збільшився для жодного зображення, а час стиснення зріс лише на 1.3 %. Використання цього способу ефективне, насамперед, для фотореалістичних зображень декількох великих об'єктів. У фрагментах зображень, в яких ентропійний спосіб вибору предикторів рядків є найефективнішим, короткі заміни (3-4 літерали), як правило, не застосовуються. Оптимальними предикторами при такому способі вибору найчастіше є *AveragePredict* чи *PaethPredict*.

Середній КС зображень набору ACT з використанням ентропійних предикторів коротких замін LZ77 зменшився порівняно з середнім КС ентропійних поелементних предикторів на 0.02 % (рядок *Варіант 5* у табл. 3.7, 3.8), хоча покращення спостерігається лише на синтезованих зображеннях та фотореалістичних зображеннях, для яких короткі заміни ефективні. Це й не дивно, адже розрахунок ентропійного КС після імітації коротких замін найкраще з розглянутих варіантів моделює для них стиснення у форматі PNG. Середня тривалість компресії файлів набору ACT внаслідок імітації коротких замін під час вибору предикторів рядків зросла на 9.33 %. Оптимальними предикторами при виборі за мінімальним ентропійним КС після застосування коротких замін алгоритму LZ77 найчастіше виявляються *LeftPredict* чи *RightPredict*. Використання цього способу ефективне, насамперед, для фотореалістичних зображень з багатьма об'єктами.

Застосування комбінованого ентропійного способу вибору предикторів рядків (рядок *Варіант 6* у табл. 3.7, 3.8) фактично зводиться до вибору кращого з двох попередніх способів. Але реалізація цього способу, як зазначалося в підпункті 3.2.2.3, не враховує впливу довгих замін алгоритму LZ77 на КС, тому й зроблений вибір може виявитися помилковим (див., наприклад, результати стиснення зображення № 1). Застосування цього способу зменшує середній КС відносно кращого з двох попередніх на 0.07 %, хоча й сповільнює кодування на понад 21 %, оскільки виконує вдвічі більше обчислень довжин кодів під час вибору предикторів.

Емпіричний комбінований ентропійний спосіб вибору предикторів з додатковим попіксельним розкладом алгоритму LZ77 оригіналу зображення (рядок

Варіант 7 у табл. 3.7, 3.8) дає змогу зменшити середній КС файлів набору АСТ на 0.45 % відносно попереднього способу виключно за рахунок синтезованих зображень, на яких він забезпечує КС предиктора *NonePredict* (див. табл. 3.5), як цього і слід було чекати. Формування додаткового попіксельного розкладу в процесі реалізації цього способу у середньому хоча й сповільнює кодування фотореалістичних зображень на 9.5 % та синтезованих з шумами – на 6.4 %, зате прискорює стиснення синтезованих зображень на 24 % за рахунок невикористання для них комбінованого ентропійного способу вибору предикторів. Даний спосіб вибору предикторів для рядків пікселів орієнтований на зменшення як кодової надлишковості фотореалістичних зображень, так і міжелементної надлишковості синтезованих зображень та забезпечує найменші КС серед розглянутих способів, тому саме його ми використаємо у двох наступних підрозділах цього розділу.

Емпіричний спосіб з прогнозуванням результатів розкладів алгоритму LZ77 (рядок *Варіант 8* у табл. 3.7, 3.8) хоча й дає змогу несуттєво зменшити КС окремих файлів (наприклад, № 1), але в середньому не забезпечує найменші розміри стиснутих зображень та передбачає формування багатьох розкладів цього алгоритму, що в десятки разів сповільнює кодування. Саме тому цей спосіб не використовувався нами у роботі надалі.

Оскільки різні фрагменти зображень можуть мати відмінні рівні міжелементної та кодової надлишковості, то, співставляючи дані табл. 3.5 і 3.7, приходимо до висновку, що **для кожного рядка зображення найефективнішим може виявитися один з п'яти варіантів компресії**: без використання предикторів, зі застосуванням *LeftPredict*, з використанням *RightPredict*, зі застосуванням безпосереднього ентропійного способу вибору предикторів та з використанням ентропійного способу вибору предикторів після застосування коротких замін алгоритму LZ77.

3.3. Фрагментування зображень

3.3.1. Алгоритм розбиття зображень на блоки рядків. Звичайно, кожен рядок зображення можна розмістити в окремому стиснутому блоці формату

DEFLATE. Але загалом таке стиснення не буде найефективнішим, оскільки в заголовку кожного стиснутого блоку динамічних кодів HUFF міститься опис довжин кодів його розподілів літералів/довжин та зміщень, який може займати до 1324 бітів, а такі витрати неприпустимі для кожного рядка. З іншого боку, поєднання у стиснутих блоках даних довільних суміжних рядків може суттєво збільшити довжину коду HUFF для їх послідовностей літералів/довжин та зміщень і тому, враховуючи наступне твердження, негативно вплинути на загальний КС.

Твердження 3.1. **Довжина ентропійного коду** (до якої наближається довжина коду HUFF) **поєднання двох послідовностей елементів є не меншою від суми довжин ентропійних кодів цих послідовностей**.

Доведення. Нехай у першій послідовності довжини N' елемент s_i зустрічається N_i' разів, а в другій послідовності довжини N'' цей же елемент зустрічається N_i'' разів. Тоді довжина ентропійного коду таких елементів у коді першої послідовності, згідно з фундаментальним положенням теорії інформації [8, с. 619-620], складе $L_i' = N_i' \log\left(\frac{N'}{N_i'}\right)$, у другій становитиме $L_i'' = N_i'' \log\left(\frac{N''}{N_i''}\right)$, а у послідовності, створеній в результаті поєднання цих двох послідовностей, довжина ентропійного коду таких елементів буде дорівнювати $L_i = (N_i' + N_i'') \log\left(\frac{N' + N''}{N_i' + N_i''}\right)$.

Обчислимо приріст довжини ентропійного коду елементів s_i внаслідок поєднання двох послідовностей:

$$dL_i = L_i - L_i' - L_i'' = (N_i' + N_i'') \log\left(\frac{N' + N''}{N_i' + N_i''}\right) - N_i' \log\left(\frac{N'}{N_i'}\right) - N_i'' \log\left(\frac{N''}{N_i''}\right). \quad (3.4)$$

Дослідивши функцію (3.4) на екстремум по змінних N_i' та N_i'', приходимо до висновку, що вона досягає мінімального значення, яке дорівнює нулю, при

$$\frac{N_i'}{N'} = \frac{N_i''}{N''}. \quad (3.5)$$

У всіх інших випадках приріст довжини ентропійного коду елементів s_i додатній. Оскільки приріст довжини ентропійного коду кожного з елементів внаслідок поєднання двох послідовностей невід'ємний, то й загальна довжина ентропійного коду внаслідок такого поєднання не зменшується. ■

Розглянемо, наприклад, залежність приросту довжини ентропійного коду поєднання двох послідовностей довжини 100 елементів для всіх представників елемента s_i від частот даного елемента у цих вхідних послідовностях (рис. 3.3).

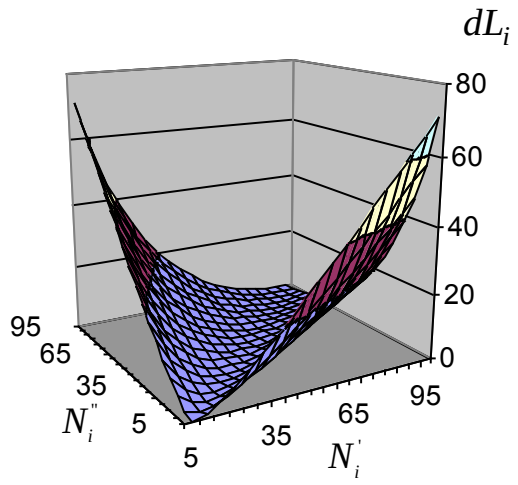


Рис. 3.3. Залежність приросту довжини ентропійного коду поєднання двох вхідних послідовностей від частот елемента s_i у цих вхідних послідовностях

Як видно з цього рисунка, приріст довжини ентропійного коду дорівнює нулю, лише тоді, коли відносні частоти елемента у двох вхідних послідовностях однакові (як це і слід було чекати з (3.5)). У інших випадках цей приріст додатній, причому він зростає зі збільшенням різниці між відносними частотами елемента у вхідних послідовностях. Отже, **довжина ентропійного коду незначно зростає лише внаслідок поєднання блоків рядків з близькими відносними частотами окремих елементів**. Результати експериментів показали, що цей висновок поширюється і на довжину коду HUFF, хоча й вивести аналітичну залежність його приросту від частот елементів у вхідних послідовностях аналогічно (3.4) неможливо, оскільки невідомі аналітичні формули для вираження довжини коду HUFF [25, с. 33].

Тому ми пропонуємо в процесі попереднього аналізу зображень перед стисненням у форматі PNG **поєднати суміжні рядки з близькими розподілами**

відносних частот у блоки, якщо предиктор для них обирається комбінованим ентропійним способом (див. алгоритм підпункту 3.2.3.1). Кожен блок рядків в процесі подальшого стиснення може належати одному, а може й бути розбитим на декілька стиснутих блоків, адже розмір стиснутого блоку, як правило, не перевищує 64 Кб. Але **різні блоки рядків мають обов'язково належати різним стиснутим блокам**, адже вони мають різні нерівномірності розподілів. Зрозуміло, що **поєднувати між собою суміжні блоки рядків доцільно лише тоді, коли прогнозоване збільшення довжини коду від їх поєднання не перевищує розмір заголовка нового стиснутого блоку** (з врахуванням можливого ефекту алгоритму LZ77 – в середньому 1500 бітів). Оцінювати прогнозовану довжину коду блоку рядків після застосування обраних предикторів можна як з використанням довжини ентропійного коду, так і з застосуванням довжини коду HUFF, алгоритми обчислень яких розглядалися в підрозділі 2.2. При цьому довжина коду HUFF хоча й розраховується довше, але іноді забезпечує менші КС, оскільки саме це кодування використовується у форматі PNG.

Покроково алгоритм формування блоків з рядків, предиктор для яких обирається комбінованим ентропійним способом, запишемо так:

1. Визначити для кожного з цих рядків пікселів зображення предиктор, який породжує елементи з найменшою довжиною ентропійного коду (2.1), як описано у пункті 3.2.2, та частоти окремих елементів після його застосування. Віднести кожен рядок до окремого блоку рядків.

2. Розрахувати для всіх пар суміжних блоків за частотами їх елементів прогнозоване збільшення довжини коду внаслідок їхнього можливого поєднання. Визначити пару суміжних блоків, що породжує найменше таке збільшення.

3. Якщо залишився лише один блок рядків або найменше збільшення довжини коду перевищує 1500 бітів, то завершити виконання алгоритму. Інакше поєднати пару суміжних блоків, що породжує дане збільшення, просумувати частоти їх елементів і повернутися до кроку 2.

Зауважимо, що перераховувати після кожної ітерації прогнозоване збільшення довжини коду для всіх пар суміжних блоків не потрібно. Достатньо це зробити під

час першої ітерації і надалі перераховувати прогнозоване збільшення довжини коду лише для пар суміжних блоків, які включають блок, утворений в результаті поєднання.

Розбиття даних на блоки передбачено самим форматом PNG [62]. Розглянутий алгоритм, по суті, лише зміщує окремі межі між цими блоками перед їх відокремленим стисненням для підвищення нерівномірностей їх розподілів, і тому дає змогу ефективніше застосовувати кодування HUFF та проводити аналіз зображень перед стисненням (див. підрозділ 3.5). Загалом, розбиття зображень на блоки пікселів з близькими значеннями ентропії та наступне поблочне стиснення за рахунок підвищення нерівномірності розподілу окремих фрагментів покращує КС довільного контекстно-незалежного алгоритму, а отже може з успіхом використовуватись у всіх форматах, де такі алгоритми застосовуються. **Після формування блоків рядків доцільно для кожного з них окремо обрати предиктори комбінованим ентропійним способом** (див. алгоритми підпункту 3.2.2.3), оскільки різні такі блоки обов'язково належать різним стиснутим блокам PNG-файлів і для кожного з них генеруються різні коди HUFF, а тому застосування коротких замін алгоритму LZ77 у різних блоках може виявитися як ефективним, так і неефективним. Фрагменти програми мовою C, що реалізують розглянутий алгоритм, наведено у п'ятому розділі додатка Ж.

3.3.2. Алгоритм розбиття результатів розкладу LZ77. Альтернативний спосіб фрагментування даних для стиснутих блоків полягає в аналізі розподілів частот елементів після виконання алгоритму LZ77. З цією метою ми розбивали потік літералів/довжин (оскільки він займає найбільше місця у стиснутих даних) на блоки зі застосуванням методу, запропонованого О. А. Ратушняком [34, с. 46-52; 24, с. 239-246] та ітеративно поєднували утворені суміжні блоки, якщо це зменшувало КС. Покроково цей алгоритм подамо так:

1. Для кожних 16000 елементів потоку літералів/довжин (оскільки менші блоки лише сповільнюють кодування, суттєво не впливаючи на КС) визначити позицію з найбільшою сумою модулів відхилень між частотами елементів вікон зі 100 елементів зліва та справа від неї і знайти ще дві аналогічні позиції зліва та

справа від цих вікон.

2. Розбити даний потік на блоки з межами у віднайдених позиціях.

3. Ітеративно поєднати окремі з утворених суміжних блоків для зменшення КС, як це описано в пунктах 2-3 попереднього алгоритму.

Після застосування цього алгоритму кожен фрагмент результатів розкладу LZ77, що відповідає черговому блоку літералів/довжин, потрібно зберегти в окремому блоці стиснутих даних формату DEFLATE. Інший спосіб фрагментування результатів розкладу LZ77 (хоча й з нижчою ефективністю), що реалізує розбиття на мінімальні блоки по 512 байт (замість перших двох кроків наведеного алгоритму) наведено у [44].

3.3.3. Аналіз результатів застосування запропонованих алгоритмів фрагментування. Проаналізуємо **результати застосування** розглянутих алгоритмів фрагментування (табл. 3.9, 3.10) для стиснення зображень набору ACT (див. табл. 1.3). Наведені алгоритми фрагментування застосовувалися у програмі з емпіричним комбінованим ентропійним способом вибору предикторів та додатковим попіксельним розкладом алгоритму LZ77 оригіналу зображення (рядок *Варіант 7* у табл. 3.7, 3.8).

Таблиця 3.9

КС файлів зображень набору ACT після застосування різних варіантів фрагментування, %

Варіант фрагментування	№ файла								Середній КС
	1	2	3	4	5	6	7	8	
Без алгоритмів фрагментування	22.94	6.85	60.21	52.56	53.84	66.18	7.17	57.76	40.94
Розбиття зображень на блоки рядків	21.70	6.85	60.21	52.56	53.84	66.18	7.17	57.85	40.79
Розбиття результатів розкладу LZ77 на блоки	22.80	6.82	60.21	52.47	53.84	66.18	7.10	57.76	40.90
Розбиття зображень на блоки рядків та результатів розкладу LZ77 на блоки	21.61	6.82	60.21	52.56	53.84	66.18	7.10	57.85	40.77

Таблиця 3.10

Час кодування файлів зображень набору ACT внаслідок застосування різних варіантів фрагментування, с

Варіант фрагментування	№ файла								Середній час
	1	2	3	4	5	6	7	8	
Без алгоритмів фрагментування	18.23	18.62	8.46	15.05	9.45	12.74	8.35	13.73	13.08
Розбиття зображень на блоки рядків	21.42	24.77	9.23	15.93	10.16	13.52	11.48	14.50	15.13
Розбиття результатів розкладу LZ77 на блоки	19.11	18.73	9.17	16.54	10.16	13.51	8.52	15.05	13.85
Розбиття зображень на блоки рядків та результатів розкладу LZ77 на блоки	22.74	25.27	10.11	17.14	10.98	14.33	11.76	15.76	16.01

Бачимо, що застосування алгоритму розбиття на блоки рядків ефективніше для зображень з шумами, а фрагментування результатів розкладу LZ77 – для синтезованих зображень, хоча обидва ці алгоритми зменшують розміри окремих файлів у форматі PNG максимум лише на десяті долі відсотка, що вказує на однорідність результатів дії предикторів та коректність обмеження у програмних реалізаціях максимального розміру блоку стиснутих даних формату DEFLATE 65536 елементами. Дані алгоритми збільшують рівень кодової надлишковості даних зображення на різних етапах їх опрацювання (перший – між яскравостями компонентів пікселів, другий – між фрагментами розкладу LZ77), тому у випадку сумісного використання можуть забезпечити додаткове зменшення КС. У наступних підрозділах цього розділу застосуємо підхід алгоритму розбиття зображення на блоки рядків, оскільки додаткове застосування ще й алгоритму розбиття результатів розкладу LZ77 несуттєво впливає на КС та ускладнює програмну реалізацію.

3.4. Зменшення коефіцієнтів стиснення розкладів алгоритму LZ77

3.4.1. Мінімізація зміщень результатів розкладу алгоритму LZ77. Опишемо алгоритм підвищення ефективності **результатів** сформованих розкладів алгоритму LZ77, які виконують пошук однакових послідовностей не зі всіх позицій потоку (наприклад, серед розглянутих у пункті 1.1.1 – це всі розклади, крім майже

оптимального). Цей алгоритм не вимагає додаткових пошуків однакових послідовностей у словнику, і тому не призводить до кардинального збільшення часу кодування відповідними програмами.

Розглянемо пару замін, які розташовуються у сформованому розкладі підряд. Нехай довжина чергової заміни рівна *m*, а попередньої – *n* (верхня частина рис. 3.4).

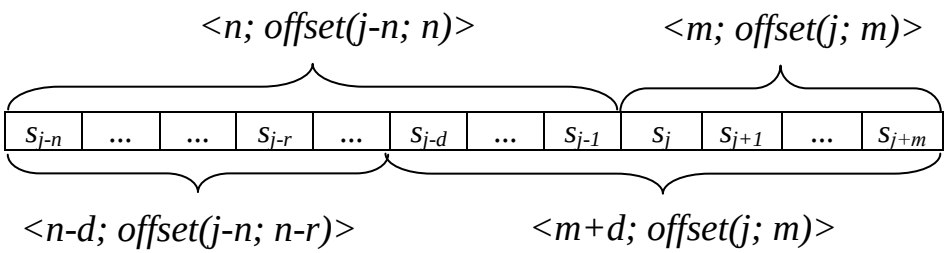


Рис. 3.4. Приклад мінімізації зміщень результатів розкладу LZ77

Визначимо максимальну кількість елементів *r*, на яку можна розширити чергову заміну за рахунок попередньої за тим самим зміщенням *offset(j; m)*, тобто

$$r = \max_{k=0,n-3,n-1} \left\{ k \;\middle|\; \forall\, i = \overline{0,k}\; s_{j-i} = s_{j-offset(j;m)-i} \right\}. \tag{3.6}$$

Фактично для знаходження *r* порівняння елементів послідовності необхідно здійснювати не вперед, а назад. Подальший аналіз пари замін виконуємо за умови $r>0$. Якщо $r=n-1$, то замість попередньої заміни запишемо елемент s_{j-n} а чергову заміну розширимо до $m+r$ елементів, зменшивши тим самим прогнозовану кількість бітів для кодування потоку. Інакше порівнюємо між собою зміщення попередньої заміни до та після можливого зменшення її довжини. Враховуючи (1.1), у випадку

$$offset\,(j-n;n-r) < offset\,(j-n;n)\,, \tag{3.7}$$

тобто **коротшу** однакову послідовність вдається віднайти за **меншим** зміщенням, замість попередньої заміни запишемо *<n-d; offset(j-n; n-r)>*, де

$$d = \min_{k=1,r} \left\{ k \;\middle|\; offset(j-n;n-r) = offset(j-n;n-k) \right\}, \tag{3.8}$$

тобто *n-d* – це максимальна довжина однакової послідовності з початку попередньої заміни за мінімально допустимим зміщенням *offset(j-n; n-r)*. У цьому випадку

активна заміна перезапишеться як *<m+d; offset(j; m)>* (див. нижню частину рис. 3.4). Виходячи з логіки описаного алгоритму, пари суміжних зміщень слід аналізувати у розкладі LZ77 з кінця до початку, а під час його формування доцільно для коротших віднайдених замін запам'ятовувати менші зміщення.

Мінімізуємо, наприклад, зміщення отриманого у підпункті 1.1.1.1 *жадібного* розкладу LZ77 "2, 4, 1, 2, <3; 4>, 3, <4; 8>, <3; 7>" потоку "2, 4, 1, 2, 2, 4, 1, 3, 2, 4, 1, 2, 4, 1, 3". Суміжна пара зміщень зустрічається в кінці цього розкладу. Останню заміну можна розширити за рахунок передостанньої на один елемент, тобто, згідно (3.6), r=1. Мінімальне зміщення для звуженої передостанньої заміни "2, 4, 1" рівне 4, що менше 8, отже вимога (3.7) виконується. Використовуючи (3.8) знаходимо, що d=1 і після перезапису двох останніх замін, оскільки інші суміжні заміни відсутні, отримаємо розклад LZ77 "2, 4, 1, 2, <3; 4>, 3, <3; 4>, <4; 7>". У цьому вдосконаленому розкладі зменшено зміщення передостанньої заміни, тому після використання алгоритму HUFF він буде закодований не більшою (як правило, меншою) кількістю бітів. Структура розглянутих суміжних замін до та після вдосконалення наведена на рис. 3.5. На ньому розширення наступної заміни за рахунок попередньої виведено на сірому фоні. Мінімізація ж зміщень візуально виявляється у зменшенні загальної довжини дуг зміщень замін.

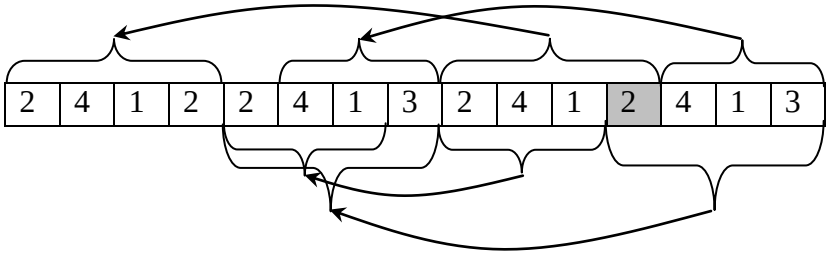


Рис. 3.5. Суміжні заміни розкладу LZ77 (початки стрілок) потоку "2, 4, 1, 2, 2, 4, 1, 3, 2, 4, 1, 2, 4, 1, 3" до (у верхній частині) та після (у нижній частині) застосування алгоритму мінімізації зміщень

Проаналізуємо **результати застосування** наведеного алгоритму мінімізації зміщень розкладу LZ77 для стиснення зображень набору ACT (див. табл. 1.3). Тестування проводилося базовою програмою з застосуванням алгоритму розбиття

зображень на блоки (див. пункт 3.3.1) та емпіричного комбінованого ентропійного способу вибору предикторів для рядків пікселів (див. підпункт 3.2.3.1). Результати тестування для всіх розкладів, розглянутих у пункті 1.1.1, наведено у табл. 3.11 та 3.12. Для майже оптимального розкладу LZ77 запропонований алгоритм не застосовувався, оскільки цей розклад забезпечує використання оптимальних зміщень.

Як свідчать результати тестування, використання алгоритму мінімізації зміщень ефективне, насамперед, для синтезованих зображень, оскільки в них переважає міжелементна надлишковість, яка усувається алгоритмом LZ77. Застосування описаного алгоритму зменшило КС програми, що використовувала "жадібний" розклад / "лінивий" розклад / розклад Кадача, максимум на 0.25 % / 0.13 % / 0.17 %, а у середньому по набору АСТ – на 0.11 % / 0.07 % / 0.1 % та скоротило відставання від результатів програми, що використовувала майже оптимальний розклад у середньому на 25 % / 28 % / 45 %. Нижча ефективність алгоритму мінімізації зміщень у випадку його застосування до "лінивого" розкладу пояснюється тим, що цей розклад відділяє окремі елементи, які ефективніше кодуються окремо, тобто зменшує кількість суміжних замін та виконує частину роботи описаного алгоритму. З іншого боку, застосування розглянутого алгоритму сповільнило стиснення в середньому лише на 1.9 % / 2.7 % / 0.7 % в той час, як програма з майже оптимальним розкладом виконується повільніше в середньому у 4 рази.

Отже, алгоритм мінімізації зміщень результатів розкладу LZ77 доцільно використовувати, насамперед, для зменшення КС на понад 0.1 % синтезованих зображень без шумів у випадку формування "жадібного" розкладу чи розкладу Кадача. Застосування "лінивого" розкладу / розкладу Кадача у середньому дає змогу забезпечити менший КС відносно "жадібного" розкладу на 0.1 % / 0.13 %, хоча й сповільнює кодування на 13 % / 12 %.

Таблиця 3.11

Зміни КС файлів зображень набору АСТ у форматі PNG внаслідок застосування різних варіантів формування розкладу LZ77 та алгоритму мінімізації його зміщень, %

Варіант розкладу LZ77	Застосування алгоритму мінімізації зміщень	№ файла								Середній КС
		1	2	3	4	5	6	7	8	
"Жадібний"	Ні	21.70	6.85	60.21	52.56	53.84	66.18	7.17	57.85	40.79
	Так	21.66	6.60	60.21	52.39	53.84	66.00	6.97	57.76	40.68
"Лінивий"	Ні	21.66	6.76	60.21	52.30	53.84	65.83	7.17	57.76	40.69
	Так	21.66	6.60	60.21	52.21	53.71	65.83	7.04	57.68	40.62
Кадача	Ні	21.66	6.74	60.21	52.39	53.84	65.57	7.10	57.76	40.66
	Так	21.66	6.57	60.21	52.21	53.71	65.48	6.97	57.68	40.56
Майже опт.	Ні	21.80	6.41	60.21	52.12	53.71	64.87	6.83	57.59	40.44

Таблиця 3.12

Зміни часу кодування файлів зображень набору АСТ у формат PNG внаслідок застосування різних варіантів формування розкладу LZ77 та алгоритму мінімізації його зміщень, с

Варіант розкладу LZ77	Застосування алгоритму мінімізації зміщень	№ файла								Середній час
		1	2	3	4	5	6	7	8	
"Жадібний"	Ні	21.42	24.77	9.23	15.93	10.16	13.52	11.48	14.5	15.13
	Так	22.08	25.32	9.33	15.99	10.32	13.84	11.65	14.72	15.41
"Лінивий"	Ні	24.50	27.74	10.00	19.11	11.59	14.72	12.52	16.54	17.09
	Так	25.16	28.23	10.17	19.99	12.20	14.94	12.63	17.14	17.56
Кадача	Ні	24.11	27.19	9.94	18.67	11.65	14.50	12.69	16.37	16.89
	Так	24.28	27.69	10.00	18.79	11.70	14.72	12.58	16.42	17.02
Майже опт.	Ні	88.93	211.57	12.14	30.43	15.82	17.46	85.91	22.63	60.61

3.4.2. Врахування прогнозованих довжин кодів елементів розподілів в процесі формування модифікованого "лінивого" та майже оптимального розкладів алгоритму LZ77. Як зазначалося у підпункті 1.1.1.2, для ефективного формування майже оптимального розкладу необхідно наперед знати приблизні вартості (довжини, кількості бітів) кодів HUFF для літералів/довжин та зміщень. Оскільки середня довжина коду HUFF у кожному стиснутому блоці близька до ентропії (1.3), то, використовуючи фундаментальне положення теорії інформації [8, с. 620], приблизну довжину цього коду $\tilde{L}_i$ без врахування впливу алгоритму LZ77 для літерала s_i доцільно розраховувати за формулою

$$\tilde{L}_i = -\log(p_i) = -\log\left(\frac{N_i}{N}\right). \qquad (3.9)$$

Визначити ж наперед прогнозовані довжини кодів HUFF для довжин замін та зміщень неможливо без формування розкладу LZ77, який, до того ж, зменшує частоти окремих елементів і тому збільшує їх прогнозовані довжини. Тому покладемо їх значення рівними емпірично визначеним константам, враховуючи, що заміни зустрічаються частіше і, відповідно, мають меншу довжину у блоках з переважаючою міжелементною надлишковістю (першого типу). Рідше такі заміни зустрічаються у блоках з переважаючою кодовою надлишковістю, для яких ефективні короткі заміни алгоритму LZ77 (третього типу). Ще рідше дані заміни зустрічаються (і, відповідно, мають найбільшу прогнозовану довжину коду HUFF) у блоках з переважаючою кодовою надлишковістю та неефективними короткими заміни алгоритму LZ77 (другого типу). Розбиття зображень на блоки рядків та визначення їх типів доцільно виконувати за допомогою алгоритмів, викладених відповідно у пункті 3.3.1 та підпункті 3.2.3.1. Фрагмент програми мовою С для визначення прогнозованих довжин кодів елементів розподілів кожного блоку рядків записується, наприклад, так:

```
lenDistance=5; // прогнозована довжина коду базового значення зміщення
for (i=0; i<countBlockFilter; i++) // цикл по блоках рядків
 {switch (tupBlockFilter[i]) // вибір по типу чергового блоку рядків
   {case 1:plusBitLenLiteral=2; // к-ть додаткових бітів для літералів
           lenZamina=2; break; // к-ть бітів для довжини заміни
```

```
   case 2:plusBitLenLiteral=1.5; lenZamina=4; break;
   case 3:plusBitLenLiteral=2; lenZamina=4; break; }
 memset(freq, 0, 256*sizeof(UBYTE4)); // обнулення масиву частот
 l=nextPozStartBlockFilter[i]-k; // кількість елементів чергового блоку
 k=nextPozStartBlockFilter[i];
 while (currentPozImage++<k) // цикл по літералах чергового блоку
  freq[imageData[currentPozImage]]++; // накопичення частот літералів
 for (j=0; j<256; j++)
  if (freq[j]) // розрахунок прогнозованих довжин кодів, виходячи з (3.9)
   prognozLenCodeLiteralBF[i][j]=-log2((double) freq[j]/l)+plusBitLenLiteral;
  else prognozLenCodeLiteralBF[i][j]=0;
 for (j=257; j<=285; j++) // цикл по кодах базових значень довжин замін
  prognozLenCodeLiteralBF[i][j]=lenZamina;
 if (tupBlockFilter[i]==3) // для блоків цього типу короткі заміни
  // враховуються, тому мають меншу довжину
  prognozLenCodeLiteralBF[i][257]=prognozLenCodeLiteralBF[i][258]=
  prognozLenCodeLiteralBF[i][259]=lenZamina-2;
 for (j=0; j<=29; j++) // цикл по кодах базових значень зміщень
  prognozLenCodeDistanceBF[i][j]=lenDistance; }      .
```

Для визначення точніших значень прогнозованих довжин кодів HUFF елементів розподілів доцільно виконати попередній "жадібний" розклад алгоритму LZ77 для даних зображення після кодування обраними предикторами, в межах кожного блоку рядків застосувати до отриманих літералів/довжин та зміщень алгоритм мінімізації розміру стиснутого блоку (див. підрозділ 3.1), згенерувати за їх частотами коди HUFF та використати довжини отриманих кодів для формування майже оптимального чи, як буде показано далі, модифікованого "лінивого" розкладу.

Визначення прогнозованих довжин кодів літералів/довжин замін та зміщень дає змогу не лише ефективно застосовувати майже оптимальний, а й модифікувати "лінивий" розклад. Класичний такий розклад відмовляється від виконання заміни з позиції j, якщо з наступної позиції вдається віднайти довшу заміну і ніяк не враховує коди цих замін у стиснутому блоці. Ми ж пропонуємо (*модифікація 1*) відмовлятися від заміни з позиції j, якщо використання літерала та заміни з наступної позиції дасть змогу забезпечити менший КС, тобто:

$$\frac{price(j, j) + price(j+1, j+len(j+1))}{1+len(j+1)} < \frac{price(j, j+len(j)-1)}{len(j)},$$

або без операцій ділення:

$$\begin{gathered}(price(i, j) + price(i+1, j+len(i+1)) \times len(i)) \\ < price(i, j+len(i)-1) \times (+len(i+1)),\end{gathered} \tag{3.10}$$

Відмовлятися від заміни з позиції *j* можливо (*модифікація 2*) також, коли вона кодує **відповідні літерали** більшою кількістю бітів, ніж літерал з чергової позиції та заміна з наступної позиції:

$$price(i, j) + \frac{price(i+1, j+len(i+1)) \times (len(i)-1)}{len(i+1)} < price(i, j+len(i)-1),$$

або без операцій ділення:

$$\begin{gathered}price(i+1, j+len(i+1)) \times (len(i)-1) \\ < (price(i, j+len(i)-1) - price(i, j)) \times len(i+1),\end{gathered} \tag{3.11}$$

Крім цих двох модифікацій ми застосовували також (зокрема, для формування "лінивих" розкладів реалізації алгоритму наступного підрозділу) відкидання замін за допомогою (3.11) для блоків, у яких прогнозований КС алгоритму LZ77 менший 0.128, та використовуючи (3.10) – для всіх інших блоків (*модифікація 3*).

Проаналізуємо **результати застосування** модифікацій "лінивого" та попереднього "жадібного" розкладів алгоритму LZ77 (табл. 3.13, 3.14) для стиснення зображень набору ACT (див. табл. 1.3). Тестування виконувалося програмою попереднього пункту з використанням зазначених розкладів. Попередній "жадібний" розклад не застосовувався перед класичним "лінивим" розкладом, оскільки цей розклад не використовує прогнозовані довжини кодів елементів розподілів.

Як свідчать результати тестування, використання модифікацій "лінивого" розкладу алгоритму LZ77 дає змогу у середньому досягнути менших КС, ніж у випадку класичного варіанту цього розкладу, на 0.01 %, хоча й сповільнює кодування на 1.8 %. Застосування ж попереднього "жадібного" розкладу у середньому зменшує КС модифікованих "лінивих" розкладів на 0.02 – 0.05 %, а майже оптимального – аж на 0.16 %, оскільки останній розклад використовує прогнозовані довжини кодів розподілів літералів/довжин та зміщень для всіх позицій потоку, а всі модифікації "лінивого" – лише для окремих його позицій.

Таблиця 3.13

Зміни КС файлів зображень набору ACT у форматі PNG внаслідок застосування модифікацій "лінивого" та попереднього "жадібного" розкладів LZ77, %

Варіант розкладу LZ77	Застосування попереднього "жадібного" розкладу	№ файла								Середній КС
		1	2	3	4	5	6	7	8	
"Лінивий"	Ні	21.66	6.76	60.21	52.30	53.84	65.83	7.17	57.76	40.69
"Лінивий", модифікація 1	Ні	21.66	6.74	60.21	52.39	53.84	65.65	7.17	57.76	40.68
	Так	21.66	6.74	60.21	52.21	53.71	65.74	7.10	57.68	40.63
"Лінивий", модифікація 2	Ні	21.70	6.74	60.21	52.39	53.84	65.65	7.17	57.76	40.68
	Так	21.66	6.74	60.21	52.21	53.71	66.00	7.10	57.68	40.66
"Лінивий", модифікація 3	Ні	21.66	6.74	60.21	52.39	53.84	65.65	7.17	57.76	40.68
	Так	21.66	6.74	60.21	52.21	53.71	65.74	7.10	57.68	40.63
Майже оптимальний	Ні	21.80	6.41	60.21	52.12	53.71	64.87	6.83	57.59	40.44
	Так	21.28	6.24	60.21	52.12	53.71	64.44	6.56	57.68	40.28

Таблиця 3.14

Зміни часу кодування файлів зображень набору ACT у формат PNG внаслідок застосування модифікацій "лінивого" та попереднього "жадібного" розкладів LZ77, с

Варіант розкладу LZ77	Застосування попереднього "жадібного" розкладу	№ файла								Середній час
		1	2	3	4	5	6	7	8	
"Лінивий"	Ні	24.50	27.74	10.00	19.11	11.59	14.72	12.52	16.54	17.09
"Лінивий", всі модифікації	Ні	24.83	27.85	10.16	19.66	12.08	14.94	12.63	17.02	17.40
	Так	30.26	34.55	13.07	26.25	15.87	18.45	15.32	22.30	22.01
Майже оптимальний	Ні	88.93	211.57	12.14	30.43	15.82	17.46	85.91	22.63	60.61
	Так	94.47	219.65	14.33	36.30	19.22	21.37	88.87	27.29	65.19

3.5. Попередній аналіз зображень з розбиттям на мінімальні та однорідні блоки рядків

Опишемо алгоритм попереднього аналізу зображень (вперше наведений у [58]), що використовується для їх розбиття на блоки рядків, вибору предиктора для кожного рядка пікселів і прогнозування довжин кодів елементів розподілів літералів/довжин замін і зміщень кожного блоку з метою подальшого ефективного застосування модифікованого "лінивого" чи майже оптимального розкладів алгоритму LZ77 для зменшення КС зображень у форматі PNG та проаналізуємо результати застосування. Для обгрунтування доцільності етапів цього алгоритму узагальнимо результати досліджень попередніх підрозділів цього розділу. З одного боку, для кожного рядка зображення найефективнішим може виявитися один з п'яти варіантів компресії (див. висновки підрозділу 3.2). Три з цих варіантів орієнтовані на фрагменти зображень з переважаючою міжелементною надлишковістю (без застосування предикторів, з використанням *LeftPredict*, з застосуванням *RightPredict*), а два інших – на фрагменти з переважаючою кодовою надлишковістю (з застосуванням безпосереднього ентропійного способу вибору предикторів та з використанням ентропійного способу вибору предикторів після застосування коротких замін алгоритму LZ77). Визначити для кожного рядка найефективніший з цих варіантів компресії неможливо без виконання відповідних попередніх розкладів алгоритму LZ77 (чи прогнозування їх результатів) для даних зображення. З іншого боку, алгоритм LZ77 в процесі стиснення чергового рядка використовує словник з даних попередніх рядків, тобто вибір варіанта компресії рядка впливає на стиснення не лише цього рядка, а й на компресію найближчих наступних рядків. Цей вплив посилюється, якщо суміжні рядки подібні між собою (а отже мають близькі розподіли частот елементів) та стискаються з використанням однакових варіантів компресії. Тому визначення варіантів компресії для блоків суміжних рядків з близькими розподілами частот елементів (а не для окремих рядків) дає змогу частково врахувати вплив словника з даних попередніх рядків в процесі виконання алгоритму LZ77. Але під час формування розкладу цього алгоритму у форматі PNG

використовується словник розміром до 32 Кб. Ось чому обмеження розмірів блоків рядків цим значенням (формування *мінімальних* блоків рядків) дає змогу оперативно змінювати варіант компресії між суміжними фрагментами зображення та забезпечує залежність КС чергового мінімального блоку в основному лише від попереднього мінімального блоку. Крім цього, поєднання суміжних рядків з близькими нерівномірностями розподілів частот елементів у блоки ітеративно виконується так, щоб максимально зменшити прогнозовану довжину коду HUFF (див. пункт 3.3.1), а отже дає змогу ефективніше застосовувати це кодування в процесі подальшого стиснення. Саме тому в процесі попереднього аналізу зображень **варіанти компресії доцільно обирати для мінімальних блоків суміжних рядків** з близькими нерівномірностями розподілів частот елементів. Під час вибору варіантів компресії для мінімальних блоків рядків слід враховувати, що кожна зміна цих варіантів між суміжними блоками зменшує ймовірність повторень фрагментів даних для алгоритму LZ77 і, внаслідок зменшення нерівномірності розподілу літералів/довжин, призводить до виникнення нового стиснутого блоку (а, отже, і до необхідності зберігання його заголовка).

Для додаткового підвищення ефективності кодування HUFF після визначення варіанта компресії для кожного мінімального блоку доцільно поєднати суміжні мінімальні блоки рядків з однаковими варіантами компресії та близькими нерівномірностями розподілів частот літералів/довжин в *однорідні* блоки рядків, розрахувати параметри для їх наступного ефективного стиснення та забезпечити подальшу компресію таких блоків в різних стиснутих блоках PNG-файлів. Кожен однорідний блок рядків (як і блок рядків під час фрагментування) в процесі наступного стиснення може належати одному, а може бути й розбитим на декілька стиснутих блоків, адже розмір стиснутого блоку, як правило, не перевищує 64 Кб. Але різні однорідні блоки рядків мають обов'язково належати різним стиснутим блокам, оскільки вони мають різні нерівномірності розподілів.

Отже, аналіз зображень перед стисненням у форматі PNG необхідно виконувати поетапно:

1. Поєднати суміжні рядки з близькими нерівномірностями розподілів у

мінімальні блоки рядків (до 32 Кб).

2. Визначити для кожного мінімального блоку оптимальний варіант компресії (з п'яти можливих) та відповідні предиктори рядків методом динамічного програмування так, щоб забезпечити **загальний** мінімальний прогнозований КС зображення.

3. Поєднати суміжні мінімальні блоки рядків з однаковими варіантами компресії та близькими нерівномірностями розподілів в однорідні блоки рядків.

4. Розрахувати для кожного однорідного блоку рядків прогнозовані довжини кодів розподілів літералів/довжин замін та зміщень для подальшого ефективного кодування алгоритмом LZ77 з модифікованим "лінивим" чи майже оптимальним розкладом в процесі безпосереднього стиснення.

Приклад меж мінімальних та однорідних блоків рядків для тестового зображення (стиснуте в 4 рази по горизонталі та вертикалі зображення № 21 з набору KTCI) наведено на рис. 3.6 відповідно зліва та справа. Восьмий мінімальний блок тестового зображення складається лише з останнього рядка, що містить однакові пікселі.

Рис. 3.6. Межі мінімальних та однорідних блоків тестового зображення

Як видно з цього прикладу, мінімальні блоки складаються з рядків, що мають близькі нерівномірності розподілів елементів, а однорідні блоки рядків дають змогу поєднати мінімальні блоки рядків з близькими нерівномірностями розподілів

літералів/довжин та зміщень і ліквідують найменші такі блоки для зменшення загального КС зображення.

Для реалізації першого з наведених етапів необхідно застосувати алгоритм розбиття зображень на блоки рядків (див. пункт 3.3.1), врахувавши обмеження максимального розміру мінімальних блоків рядків у 32 Кб, з метою виконання третього етапу – використати кроки 2 - 3 цього ж алгоритму, враховуючи відповідні частоти розподілів літералів/довжин та зміщень, а для розрахунку прогнозованих довжин їх кодів на четвертому етапі – скористатися алгоритмом генерування динамічних кодів HUFF (див. підрозділ 1.1.2 та [66; 24, с. 31-34; 40, с. 52-54; 17]). Тому подамо лише кроки алгоритму для реалізації другого етапу:

2.1. Визначити для кожного мінімального блоку рядків i після стиснення кожним з п'яти варіантів компресії j (0 – без застосування предикторів, 1 – з використанням *LeftPredict*, 2 – з застосуванням *RightPredict*, 3 – з використанням безпосереднього ентропійного способу вибору предикторів, 4 – з застосуванням ентропійного способу вибору предикторів після виконання коротких замін LZ77) прогнозовані розміри кодів $size_{i,j}$ в бітах ($i = \overline{0, m-1}$, де m – кількість мінімальних блоків після поєднання рядків) з використанням алгоритму мінімізації розміру стиснутих блоків, викладеному в підрозділі 3.1. Запам'ятати також для кожного мінімального блоку рядків і кожного варіанту компресії частоти елементів розподілів літералів/довжин та зміщень (а не лише окремих елементів, як для рядків перед їх поєднанням у мінімальні блоки). Для реалізації цього кроку до пікселів зображення необхідно почергово застосувати предиктори кожного з п'яти варіантів компресії, виконати "жадібний" розклад LZ77 отриманих результатів та визначити мінімальні розміри стиснутих блоків в межах кожного з мінімальних блоків рядків. Виконання цього кроку займає 50 - 71 % загального часу стиснення, причому більша частина цього часу витрачається на формування п'яти додаткових розкладів LZ77. Реалізація цього кроку добре піддається розпаралеленню з використанням п'яти паралельних процесів для кожного з п'яти варіантів компресії, адже визначення прогнозовано розміру чергового мінімального блоку рядків після застосування кожного варіанту компресії не залежить від інших варіантів компресії. Застосування

такого розпаралелення дає змогу підвищити швидкість виконання даного кроку в 4 - 5 разів, і, відповідно, прискорити стиснення на 38 - 56 %. Прогнозовані розміри мінімальних блоків зображення рис. 3.6 після стиснення різними варіантами компресії наведено в табл. 3.15.

Таблиця 3.15

Прогнозовані розміри мінімальних блоків рядків тестового зображення після стиснення різними варіантами компресії, бітів

№ мінімального блоку рядків	№ варіанта компресії				
	0	1	2	3	4
0	36103	39575	43398	37950	42293
1	62868	61936	63845	60749	61479
2	70769	66953	72486	66509	66558
3	17815	16997	18249	17232	16985
4	31019	29139	31359	29309	29201
5	114948	115870	121578	117652	117417
6	58604	55905	62519	56454	56451
7	10	15	3694	16	16

2.2. Визначити методом динамічного програмування для кожного мінімального блоку рядків *i* оптимальний варіант компресії $compres_i$ та обрати відповідні частоти елементів розподілів літералів/довжин та зміщень цього варіанту для подальшого аналізу (ці частоти були розраховані для всіх варіантів компресії на попередньому кроці). Можливість застосування цього методу забезпечує максимальний розмір мінімальних блоків рядків (32 Кб), який гарантує залежність прогнозованого розміру чергового стиснутого блоку в основному лише від попереднього блоку.

З метою застосування методу динамічного програмування встановимо штрафи $fine_{k,j}$ (прогнозовані додаткові витрати пам'яті у стиснутому зображенні, виражені в бітах) за відхилення варіанта компресії чергового мінімального блоку рядків *j* від варіанта компресії попереднього блоку *k* (табл. 3.16). Ці штрафи обумовлені як необхідністю зберігання заголовка нового стиснутого блоку, що неодмінно виникає

у випадку зміни варіанту компресії (біля 1000 бітів), так і змінами словників LZ77, що погіршують КС, зокрема: штраф у 7000 бітів за перехід від варіанта компресії з предикторами до варіанта без предикторів обумовлений відсутністю потрібного словника LZ77 для стиснення початку блоку рядків без предикторів, за рахунок якого, в основному, і відбувається його компресія; штраф за зворотний перехід складає лише 1800 – 2000 бітів, оскільки стиснення початку блоку рядків з предикторами відбувається, в основному, за рахунок використання кодів HUFF, хоча такий перехід і змінює кардинально елементи словника LZ77; штраф за перехід між варіантами стиснення з предикторами складає всього 1300 – 1500 бітів, оскільки такі переходи хоча й призводять до створення нових блоків, але не змінюють кардинально елементи словника LZ77.

Таблиця 3.16

Штрафи за відхилення між варіантами компресії суміжних мінімальних блоків, бітів

№ варіанта компресії попереднього блоку	№ варіанта компресії чергового блоку				
	0	1	2	3	4
0	0	2000	2000	1800	1800
1	7000	0	1500	1300	1300
2	7000	1500	0	1300	1300
3	7000	1500	1500	0	1300
4	7000	1500	1500	1300	0

Під час прямого ходу методу динамічного програмування визначимо мінімальний накопичений прогнозований розмір $size'_{i,j}$ від початку зображення до кожного мінімального блоку рядків i включно за умови застосування до нього варіанта компресії $j = \overline{0,4}$:

$$\begin{gathered} size'_{0,j} = size_{0,j}; \\ size'_{i,j} = \min_{k=0,4} \left(size'_{i-1,k} + fine_{k,j} \right) + size_{i,j},\ i = \overline{1,\ m-1}. \end{gathered} \tag{3.12}$$

Тобто, у відповідності до (3.12), мінімальний накопичений прогнозований розмір зображення для кожного варіанту компресії від початку до чергового мінімального блоку включно дорівнює мінімуму по всіх варіантах компресії попереднього блоку суми накопичених розмірів від початку до попереднього блоку та штрафів за відхилення між варіантами компресії попереднього і чергового блоків, збільшеному на відповідний розмір кодів чергового блоку.

Мінімальний прогнозований розмір всього зображення *minSize* обчислимо з умови мінімуму по всіх варіантах компресії накопичених розмірів до останнього блоку включно, а оптимальний варіант компресії кожного блоку $compres_i$ визначимо під час зворотного ходу методу динамічного програмування так, щоб забезпечити цей мінімальний розмір:

$$minSize = \min_{j=\overline{0,4}} size'_{m-1,j} \ , \tag{3.13}$$

$$compres_{m-1} = k \mid size'_{m-1,k} = minSize \ , \tag{3.14}$$

$$compres_i = k \mid size'_{i+1,compres_{i+1}} = size'_{i,k} + fine_{k,compres_{i+1}} + size_{i+1,compres_{i+1}}, \quad i = \overline{m-2,0} \ . \tag{3.15}$$

На практиці під час прямого ходу методу динамічного програмування для кожного блоку і кожного варіанту компресії доцільно запам'ятати номер варіанту компресії попереднього блоку, що дає змогу мінімізувати його накопичений розмір, а в процесі зворотного ходу лише використати ці номери. Прогнозовані накопичені розміри тестового зображення від початку до кожного мінімального блоку за умови застосування до нього різних варіантів компресії наведено у табл. 3.17. У цій же таблиці мінімальний прогнозований розмір всього зображення виведено на темно-сірому фоні, накопичені прогнозовані розміри блоків з оптимальними варіантами компресії, що забезпечують цей мінімум – на сірому фоні, а послідовність визначення оптимальних варіантів компресії для мінімальних блоків відображена стрілками.

Співставляючи дані табл. 3.15 і табл. 3.17, приходимо до висновку, що

оптимальний варіант компресії блоку не завжди співпадає з варіантом, який визначає мінімальний розмір цього блоку (мінімальні блоки №№ 3, 5, 7), адже оптимальна стратегія покликана забезпечити мінімальний розмір стиснутого зображення і вона враховує зміни між варіантами компресії суміжних мінімальних блоків.

Таблиця 3.17

Прогнозовані накопичені розміри для кожного мінімального блоку рядків тестового зображення після їх стиснення різними варіантами компресії, бітів

№ мінімального блоку рядків	№ варіанта компресії				
	0	1	2	3	4
0	36103	39575	43398	37950	42293
1	98971	100039	101948	98952	99392
2	169740	166992	172638	165161	165940
3	187555	183658	184910	182393	182925
4	218574	212797	215252	211702	212126
5	333522	328667	334780	329354	329543
6	392126	384572	392696	385808	385994
7	391582	384587	389766	385824	385888

Уточнити розраховані (за результатами четвертого етапу попереднього аналізу) для кожного однорідного блоку рядків прогнозовані довжини кодів HUFF розподілів літералів/довжин замін та зміщень (у випадку відсутності обмежень по часу) можна з розподілів, що утворюються за результатами додаткового попереднього розкладу даних зображення з застосуванням обраних предикторів, "жадібного" / "лінивого" розкладу алгоритму LZ77, кодування HUFF та алгоритму мінімізації розміру стиснутого блоку, викладеному у підрозділі 3.1.

Фрагмент програми для визначення оптимального варіанту компресії (з п'яти можливих) для кожного мінімального блоку рядків методом динамічного програмування наведено у шостому розділі додатка Ж.

Проаналізуємо **результати застосування** описаного алгоритму попереднього аналізу зображень (табл. 3.18, 3.19) для стиснення у форматі PNG файлів набору

ACT (див. табл. 1.3).

Таблиця 3.18

КС файлів зображень набору ACT у форматі PNG після застосування різних варіантів програм, %

Варіант програми	Застосування попереднього розкладу	№ файла								Середній КС
		1	2	3	4	5	6	7	8	
Базова	Ні	23.94	10.27	67.75	55.85	58.65	67.65	10.38	61.23	44.47
"Лінивий" розклад LZ77	Ні	20.94	6.74	60.21	52.04	53.84	65.74	7.04	57.50	40.51
	Так	20.94	6.71	60.21	52.04	53.84	65.57	7.04	57.50	40.48
Майже оптим. розклад LZ77	Ні	20.89	6.24	60.21	52.04	53.84	64.44	6.56	57.59	40.23
	Так	20.85	6.24	60.21	51.60	53.71	64.53	6.56	57.33	40.13
OptiPng, станд. перебір	Ні	22.37	6.79	60.34	52.12	53.97	66.52	7.10	57.68	40.86
OptiPng, макс. перебір	Ні	22.18	6.79	60.34	52.04	53.84	65.83	7.10	57.59	40.71

Таблиця 3.19

Час кодування файлів зображень набору ACT у формат PNG різними варіантами програм, с

Варіант програми	Застосування попереднього розкладу	№ файла								Середній час
		1	2	3	4	5	6	7	8	
Базова	Ні	10.76	15.27	4.78	9.83	5.99	7.90	6.10	8.73	8.67
"Лінивий" розклад LZ77	Ні	50.26	69.37	20.60	42.90	26.25	30.15	29.44	36.81	38.22
	Так	61.24	79.26	24.72	53.83	33.01	35.26	33.50	45.48	45.79
Майже оптим. розклад LZ77	Ні	56.52	156.15	21.14	52.67	28.61	32.85	62.39	40.70	56.38
	Так	67.73	165.38	25.43	63.28	35.48	37.85	66.57	49.43	63.89
OptiPng, станд. перебір	Ні	71.51	77.83	13.84	33.45	15.93	20.76	22.57	26.20	35.26
OptiPng, макс. перебір	Ні	996.19	684.15	289.34	733.20	408.15	486.31	336.25	562.00	561.95

Споріднені результати по набору KTCI наведено у першому розділі додатка Е. Для

отримання даних цих таблиць описаний алгоритм попереднього аналізу зображень застосовувався перед "лінивим" та майже оптимальним розкладом алгоритму LZ77 безпосередньо (відповідно, другий і четвертий рядки таблиць) та після додаткового попереднього "лінивого" розкладу для уточнення прогнозованих довжин кодів розподілів (відповідно, третій і п'ятий рядки).

Крім цього, для порівняння ефективності стиснення у першому рядку даних таблиць наведено результати застосування базової програми без використання алгоритмів цього та попереднього розділів, а у шостому та сьомому – результати стиснення програми OptiPng з параметрами стандартного та максимального перебору предикторів, розмірів стиснутих блоків та стратегій стиснення.

З даних табл. 3.19 слідує, що в середньому для зображень цього набору визначення предикторів рядків ентропійних варіантів та кожен попередній "жадібний" розклад LZ77 триває близько 5 с, лінивий розклад LZ77 та додатковий попередній розклад для уточнення прогнозованих довжин кодів – 6.7 с, майже оптимальний розклад LZ77 – 14.8 с.

Співставляючи дані табл. 3.18 з результатами експериментів попередніх підрозділів цього розділу, приходимо до висновків, що, по-перше, застосування комбінованого ентропійного способу вибору предикторів рядків з додатковим попіксельним розкладом алгоритму LZ77 оригіналу та розбиттям зображення на блоки рядків (другий рядок табл. 3.9) дозволяє досягнути кращих середніх КС, ніж програма OptiPNG з параметрами стандартного перебору (шостий рядок табл. 3.18), витрачаючи для цього в 2.33 рази менше часу і, по-друге, використання базових описаних алгоритмів та майже оптимального розкладу LZ77 дає змогу в середньому зменшити КС зображень у форматі PNG для набору ACT – на понад 4.34 %, а для KTCI – на 5.12%. Причому близько 8 % від цього зменшення досягається за допомогою майже оптимального розкладу, 57 % – алгоритму мінімізації розміру стиснутого блоку, 9 % – ентропійним способам вибору предикторів рядків, 4 % – за допомогою алгоритму розбиття зображень на блоки рядків, 2 % – попереднього "лінивого" розкладу, 20 % – завдяки наведеному алгоритму вибору варіанта компресії для кожного мінімального блоку за допомогою методу динамічного

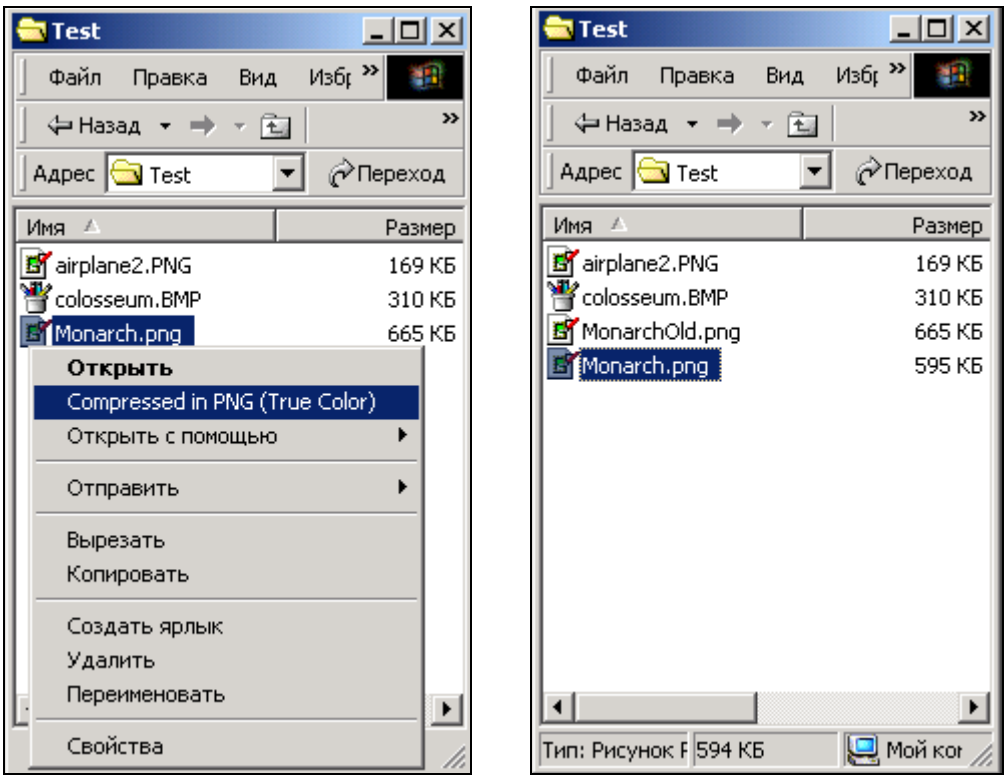
програмування. Застосування цих алгоритмів дає змогу отримати кращі в середньому КС від програми OptiPNG з параметрами максимального перебору на 0.58 % для набору ACT та на 2.3 % – для набору KTCI, витрачаючи для компресії відповідно у 8.8 та 12.6 рази менше часу, причому зменшення КС спостерігається для **всіх** зображень наборів. Навіть застосування "лінивого" розкладу алгоритму LZ77 замість майже оптимального без попереднього розкладу дозволяє зменшити КС у середньому на понад 3.96 % та отримати по цьому показнику менші значення відносно цього варіанту програми OptiPNG на 0.2 % для набору ACT і на 0.91 % – для набору KTCI та не більші КС для кожного зображення зокрема, використовуючи для компресії відповідно у 14.7 та 17.93 рази менше часу. Використання ж додаткового попереднього розкладу для уточнення прогнозованих довжин кодів розподілів однорідних блоків рядків дає змогу зменшити середній КС на понад 0.025 %, хоча й збільшує час кодування у середньому на 6.7-7.5 с.

3.6. Використання розробленої утиліти MinPNG для зменшення коефіцієнтів стиснення зображень у форматі PNG

Для стиснення файлів зображень у форматі PNG нами, використовуючи результати дослідження цього розділу, на базі програми з CD до [25] розроблена утиліта MinPNG. Зокрема, в цій утиліті реалізований попередній аналіз зображень (підрозділ 3.5) перед додатковим "лінивим" та основним майже оптимальним розкладом алгоритму LZ77, який захищений авторським свідоцтвом [1]. Інсталяція програми MinPNG розповсюджується як безкоштовне сервісне програмне забезпечення і доступна для завантаження разом з вихідними текстами цієї утиліти з Web-сторінки http:\\apserver.org.ua/peregl.php?=5. Дану утиліту можна застосовувати в операційних системах сімейств DOS та Widows за умов дотримання положень "Ліцензійної угоди" та наявності вільної оперативної чи дискової пам'яті з розміром, достатнім для розміщення п'яти вхідних файлів зображень у форматі BMP.

Після встановлення утиліта MinPNG запускається за допомогою контекстного меню "Compressed in PNG (TrueColor)" PNG- та BMP-файлів (рис. 3.7 а) та виконує їх конвертування у PNG-файли зменшеного розміру. В процесі використання

утиліти для довільного BMP-файла створюється новий файл з тією ж назвою та розширенням PNG, а до назви чергового вхідного PNG-файла дописується "Old", після чого стиснутий файл створюється під оригінальною назвою вхідного PNG-файла (рис. 3.7 б). Вікно відповідного процесу закривається автоматично після успішного завершення роботи утиліти.



а) б)

Рис. 3.7. Розмір файла зображення *Monarch.png*: а) до застосування програми MinPNG; б) після її виконання

Виконуваний файл утиліти *MinPNG.exe* зберігається у папці, зазначеній при інсталяції, разом з іншими супровідними файлами. Швидкий доступ до цієї папки виконується за допомогою меню *Пуск – Программы – MinPNG*.

Додатково, з дослідницькою метою, утиліта MinPNG може завантажуватися з командного рядка у форматі

MinPNG [-I -F -M -P -R -S] inputFile [outputFile].

Основні аргументи виконуваного файла утиліти такі: *inputFile* – назва довільного вхідного файла з розширенням та типу BMP або PNG для стиснення; *outputFile* – назва вихідного файла з розширенням та типу PNG для збереження результатів стиснення. При відсутності введеної назви вихідного файла вона

покладається рівною назві вхідного файла з розширенням PNG. У випадку однакових назв вхідного та вихідного файлів до назви вхідного файла дописується "Old".

Параметри виконуваного файла утиліти (вказуються необов'язково) використовуються для розширення його функціональності та задаються довільними комбінаціями з наступного переліку: -I – виводити додаткову інформацію про хід процесу стиснення; -F – аналізувати лише перші елементи хеш-ланцюгів в процесі виконання алгоритму LZ77; -M – аналізувати всі елементи хеш-ланцюгів в процесі виконання алгоритму LZ77; -P – не використовувати предиктори; -R – не аналізувати ефективність замін алгоритму LZ77 у DEFLATE-блоках; -S – виконувати додатковий статистичний аналіз DEFLATE-блоків.

Застосування утиліти MinPNG дає змогу отримати найменші КС зображень у форматі PNG (див. рядок 5 табл. 3.18) у порівнянні з іншими програмами, що виконують стиснення у цьому форматі.

3.7. Висновки до третього розділу

1. Для підвищення ефективності стиснення даних у форматах, що послідовно використовують алгоритми декількох методів (як у форматі PNG), доцільно враховувати взаємний вплив цих алгоритмів. Під час попередньої обробки даних перед стисненням у таких форматах для розрахунку параметрів компресії бажано враховувати всі види надлишковостей, що обробляються їх окремими алгоритмами, а також розбивати дані на однорідні блоки.

2. У форматі PNG для кожного рядка пікселів зображення предиктори слід обирати залежно від обмежень на час компресії: у випадку найшвидшого стиснення доцільно використати нелінійний предиктор *PaethPredict* для всіх рядків; під час стандартного стиснення варто скористатися ентропійним способом вибору предикторів; в процесі повільнішої компресії – обирати предиктори емпіричним комбінованим ентропійним способом з додатковим попіксельним розкладом алгоритму LZ77 оригіналу зображення, для повільного максимального стиснення – вибирати предиктори за результатами попереднього аналізу з розбиттям на блоки

мінімальних та однорідних рядків. В процесі реалізації всіх варіантів стиснення доцільно застосовувати алгоритм мінімізації розміру стиснутого блоку (див. підрозділ 3.1), який дає змогу суттєво зменшити КС більшості фотореалістичних та окремих синтезованих зображень за рахунок вибору для **кожного** блоку даних найкоротшого з альтернативних стиснутих блоків та ітеративного зменшення його розміру.

3. Запропонований спосіб попереднього аналізу (див. підрозділ 3.5) дозволяє отримати найменші КС зображень у форматі PNG за рахунок, насамперед, поділу зображень на мінімальні блоки рядків пікселів з близькими значеннями ентропії; вибору для кожного мінімального блоку рядків варіанта компресії, що забезпечує мінімальний прогнозований КС всього зображення за допомогою методу динамічного програмування; поєднання мінімальних блоків рядків в однорідні блоки та визначення прогнозованих довжин кодів розподілів отриманих блоків для наступного стиснення алгоритмом LZ77.

4. Описані алгоритми для зменшення КС RGB-зображень у форматі PNG функціонують у межах затвердженого стандарту цього формату [62], а тому не вимагають модифікації декодера чи програм перегляду зображень і за рахунок зменшення розмірів відповідних файлів лише прискорюють їх роботу.

5. Всі алгоритми цього розділу носять універсальний характер, і тому можуть ефективно застосовуватися у процесі компактного збереження даних в інших форматах, стандартах і програмах, що використовують підходи та способи кодування аналогічні формату PNG, а саме: алгоритм мінімізації розміру стиснутого блоку – у стандартах, що використовують формат словникового стиснення DEFLATE; алгоритми вибору предикторів для рядків пікселів – для форматів, які застосовують LPC; алгоритми фрагментування – у програмах з контекстно-незалежним кодуванням; модифіковані "ліниві" та майже оптимальний розклади алгоритму LZ77 та алгоритм мінімізації зміщень його результатів – для програм, що формують такі розклади; алгоритм попереднього аналізу зображень з розбиттям на мінімальні та однорідні блоки рядків – у форматах з контекстно-залежним та контекстно-незалежним кодуванням.

РОЗДІЛ 4
МОДИФІКАЦІЇ ФОРМАТУ PNG ДЛЯ ПІДВИЩЕННЯ ЕФЕКТИВНОСТІ СТИСНЕННЯ КОЛЬОРОВИХ ЗОБРАЖЕНЬ

Як відомо, у даному форматі можуть послідовно використовуватися три способи кодування: предиктори, що підвищують рівень кодової надлишковості за рахунок міжелементної, алгоритм LZ77, який зменшує міжелементну надлишковість між однаковими фрагментами даних, та кодування HUFF для ліквідації кодової надлишковості. Тому наведемо у цьому розділі для кожного з вказаних способів кодування відповідний розроблений нами алгоритм, що підвищує ефективність його кодування (формування та переходу до різницевої кольорової моделі [55; 57; 2], використання декількох ковзаючих вікон алгоритму LZ77 для результатів застосування різних предикторів [45] та алгоритм формування палітри для групового статистичного кодування [51; 46; 48]) і проаналізуємо результати використання алгоритму коригування значень предиктора (зі стандарту JPEG-LS [71; 72; 35]) та сукупного застосування зазначених і додаткових алгоритмів для модифікацій [6] формату PNG, а також опишемо програмний комплекс, в якому реалізовані ці алгоритми.

4.1. Використання декількох ковзаючих вікон для результатів застосування різних предикторів у процесі формування модифікованого розкладу алгоритму LZ77

Формат PNG (див. пункт 1.1.3 та [62]) передбачає використання єдиного предиктора для цілого рядка. Але для різних фрагментів рядка найкращими можуть виявитися різні предиктори. Тому ми пропонуємо для зменшення КС зображень у цьому форматі замість алгоритму словникового стиснення LZ77 застосувати розглянутий нижче розроблений нами альтернативний алгоритм LZPR (Ziv - Lempel predictor, вперше опублікований у [45]), який дозволяє використовувати різні статичні предиктори для різних фрагментів рядка. Це дає змогу ефективніше використовувати переваги контекстно-залежного словникового алгоритму та формувати послідовності з меншою ентропією джерела перед застосуванням

контекстно-незалежного ентропійного алгоритму.

Опишемо механізм дії алгоритму LZPR на основі "жадібного" розкладу вхідної послідовності. Нехай послідовність елементів потоку $s_1 \ldots s_{j-1}$ вже закодована. Тоді, згідно цього алгоритму, на черговому кроці виконують пошук однакової послідовності максимальної довжини не лише для незакодованих елементів буфера $s_j^0 s_{j+1}^0 \ldots$ у словнику $s_i^0 s_{i+1}^0 \ldots$ ($i<j$), але й для всіх елементів буферів $s_j^k s_{j+1}^k \ldots$ у відповідних словниках $s_i^k s_{i+1}^k \ldots$, де k вказує на номер застосованого предиктора (вхідну послідовність вважають результатом дії предиктора *NonePredict*). Тобто пошук однакових послідовностей ведуть не лише серед елементів вхідного потоку, але й серед результатів дії всіх допустимих предикторів. Найкращим для позиції j вважають предиктор l, що дає змогу віднайти однакову послідовність для елементів буфера $s_j^l s_{j+1}^l \ldots$ максимальної довжини *len*. **Тобто найкращим вважають предиктор, який дає змогу закодувати послідовність найбільшої довжини.** У випадку виявлення однакової послідовності елементи $s_j \ldots s_{j+len-1}$ кодують трійкою чисел *<довжина; зміщення від закінчення закодованої частини потоку; номер застосованого предиктора>*, тобто *<len; j-i; l>*, і компресію продовжують з позиції *j+len*. Якщо ж жоден з предикторів не дає змоги віднайти однакову послідовність, то в закодовані дані заносять елемент s_j^m, де m – номер предиктора, який дає змогу забезпечити в цілому найменшу ентропію джерела, і кодування продовжують з позиції $j+1$.

У випадку, коли декілька предикторів дають змогу закодувати послідовність максимальної довжини *len*, обирають той предиктор, який породжує найменше зміщення. Якщо ж породжені зміщення теж виявляться однаковими, то вибирають предиктор, простіший для розрахунку.

Застосуємо, наприклад, наведений алгоритм LZPR до потоку "3, 4, 6, 0, 3, 4, 6, 2, 3, 5, 1" (рис. 4.1). Як і слід було чекати, кількість віднайдених однакових послідовностей виявилася максимальною, що дає змогу використати переваги алгоритму LZ77. Крім того, зменшилася ентропія окремих закодованих літералів, що дозволяє в подальшому ефективніше використати контекстно-незалежний

алгоритм.

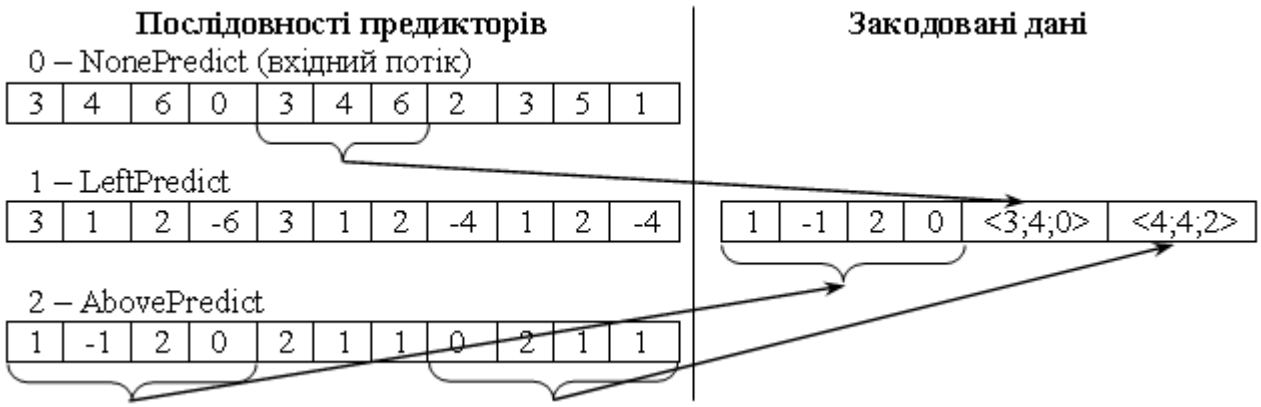


Рис. 4.1. Застосування алгоритму LZPR до потоку "3, 4, 6, 0, 3, 4, 6, 2, 3, 5, 1"

Для апробації алгоритму LZPR ми внесли такі модифікації у формат словникового стиснення DEFLATE для опрацювання 24-бітних зображень: довжини однакових послідовностей обмежили цілими пікселами, тобто зменшували їх до найближчого числа, кратного 3; зміщення однакових послідовностей обмежили цілими пікселами та зберігали зміщення в пікселах, а не в елементах, що дало змогу втричі розширити розмір словника; пошук однакових послідовностей виконували серед елементів вхідного потоку (*NonePredict*), результатів дії *LeftPredict* та *AbovePredict* (відповідають шумовій моделі), а вибір літералів для кодування – з результатів дії *MedPredict* (відповідає еволюційно-шумовій моделі), оскільки цей предиктор забезпечує в середньому найменшу ентропію окремих елементів (див. шостий рядок з табл. 3.3, 3.5). Використання лише трьох предикторів дозволило зберігати номер обраного предиктора в позиції кодування разом з довжиною однакової послідовності у вигляді їх суми, що дало змогу не змінювати внутрішню структуру формату DEFLATE. В програмних реалізаціях стиснення у модифікованому форматі DEFLATE для зменшення КС ми додатково застосовували відповідно відкоригований алгоритм мінімізації розміру стиснутого блоку (див. підрозділ 3.1). Використання трьох синхронних ковзаючих вікон фактично дає змогу поєднати переваги перших трьох варіантів компресії алгоритму попереднього аналізу зображень з розбиттям на мінімальні та однорідні блоки рядків (див. підрозділ 3.5), а встановлення коду літерала з буфера предиктора з найменшою

ентропією та застосування алгоритму мінімізації розміру стиснутого блоку – двох останніх варіантів компресії цього алгоритму.

Мовою С реалізація даного варіанту алгоритму LZPR для кодування потоку на черговому кроці має такий вигляд:

```
unsigned int length, offset, predict;
// шукаємо найдовшу однакову послідовність у словнику потоку
LongestMatch(length, offset);
predict=0;
unsigned int vlength, voffset, glength, goffset;
// шукаємо найдовшу однакову послідовність у словнику предиктора LeftPredict
GLongestMatch(glength, goffset, length, offset);
if (glength>length || (glength==length && goffset<offset))
 {length=glength; offset=goffset;
  predict=1; } // цей предиктор кодує краще
// шукаємо найдовшу однакову послідовність у словнику предиктора AbovePredict
VLongestMatch (vlength, voffset, length, offset);
if (vlength>length || (vlength==length && voffset<offset))
 {length=vlength; offset=voffset;
  predict=2; } //цей предиктор кодує краще
if (length == 0)
 { // кодуємо черговий символ з буфера MedPredict }
else
 { // кодуємо пару чисел <length+predict; offset> }
```

Запропонована реалізація алгоритму LZPR відносно кодування у стандарті формату PNG хоча й вимагає втричі більше пам'яті для функціонування кодера, оскільки пошук однакових послідовностей ведеться не лише у словнику оригіналу зображення, а й у словниках результатів застосування предикторів, але не призводить до значного збільшення витрат часу на кодування, тому що зміщення в ній обмежуються цілими пікселами.

Модифікований формат DEFLATE було використано для коригування формату файлів PNG, в якому додатково, за непотрібністю, також було вилучено байт номера предиктора перед даними кожного рядка.

Розглянемо тепер питання **вибору найкоротших хеш-ланцюгів** у процесі пошуку найдовшої однакової послідовності у декількох альтернативних словниках. Поширимо спосіб підвищення швидкості виконання програм компресії (які використовують хешування) за допомогою вибору найкоротших хеш-ланцюгів (див. підрозділ 2.1 та [47]) на випадок декількох альтернативних словників, оскільки сучасні алгоритми пошуку найдовшої однакової послідовності в декількох

альтернативних словниках ніяк не використовують інформацію про довжину вже віднайденої однакової послідовності і виконують пошук у кожному наступному словнику автономно. Нехай під час пошуку у попередніх словниках вже віднайдено однакову послідовність максимальної довжини *prevLen*. Тоді для покращення результатів пошуку у наступному словнику потрібно віднайти довшу однакову послідовність або однакову послідовність такої ж довжини, але за меншим зміщенням. Тому, розпочинаючи пошук у цьому словнику, доцільно відразу порівняти кількість елементів хеш-ланцюга, що відповідає ключу початку буфера, з *prevLen*-3 (тобто з кількістю нерозглянутих елементів ключів у найкоротшій з допустимих однакових послідовностей). Якщо перше число перевищує друге, то пошук потрібно розпочати по найкоротшому хеш-ланцюгу серед ключів початку буфера довжини *prevLen*. Інакше пошук розпочинається традиційно, по хеш-ланцюгу, що відповідає ключу початку буфера. Прискорення пошуку в даному випадку досягається теж за рахунок використання коротших хеш-ланцюгів. Це прискорення могло б бути значно більшим, якби в процес аналізу відразу можна було б включити останній ключ розширеної послідовності буфера, але така модифікація не дозволяє віднаходити однакові послідовності тієї ж довжини *prevLen* за меншим зміщенням і тому негативно впливає на КС.

Проаналізуємо **результати застосування** наведених алгоритмів для стиснення зображень набору ACT (див. табл. 1.3). Розглянемо спочатку кількості ефективних замін послідовностей по предикторах для файлів набору ACT при застосуванні модифікованого формату PNG з алгоритмом LZPR (табл. 4.1). Як видно з цієї таблиці, словник незакодованих елементів (*NonePredict*) найчастіше використовується для зображень (№№ 1, 2, 7) з переважаючими синтезованими фрагментами. Словники ж результатів дії предикторів найчастіше використовуються для фотореалістичних зображень (№№ 3-6,8), оскільки в них однакові прирости яскравостей кольорів зустрічаються частіше, ніж однакові яскравості кольорів пікселів.

Таблиця 4.1

Кількості ефективних замін послідовностей по предикторах для файлів зображень набору ACT внаслідок застосування алгоритму LZPR

Предиктор	№ файла							
	1	2	3	4	5	6	7	8
NonePredict (без предикторів)	67007	54560	322	28158	4403	77692	21039	10878
LeftPredict	28685	7225	313	30829	35394	123321	5114	19390
RightPredict	13906	21590	758	40935	10670	107616	11119	35219
Всього:	109598	83375	1393	99922	50467	308629	37272	65487

Проаналізуємо тепер зміни показників компресії внаслідок застосування алгоритмів LZPR та врахування максимальної довжини однакової послідовності з попередніх словників. Відповідні результати тестування програм з використанням стандартного та модифікованого за допомогою цих алгоритмів формату PNG для стиснення зображень набору ACT наведені у табл. 4.2, 4.3 та 4.4.

Таблиця 4.2

КС файлів зображень набору ACT у стандартному та модифікованому (з використанням алгоритму LZPR) форматі PNG, %

Опис програми	№ файла								Середній КС
	1	2	3	4	5	6	7	8	
Майже оптимальний розклад LZ77 після повного попереднього аналізу зображень	20.85	6.24	60.21	51.60	53.71	64.53	6.56	57.33	40.13
"Жадібний" розклад LZPR, всі варіанти вибору найкоротших хеш-ланцюгів	20.13	5.69	60.47	47.27	51.24	58.63	6.01	54.73	38.02

Таблиця 4.3

Час кодування файлів зображень набору ACT у стандартний та модифікований (з використанням алгоритму LZPR) формат PNG, с

Опис програми	№ файла								Середній час
	1	2	3	4	5	6	7	8	
Майже оптимальний розклад LZ77 після повного попереднього аналізу зображень	67.73	165.38	25.43	63.28	35.48	37.85	66.57	49.43	63.89
"Жадібний" розклад LZPR, автономний вибір найкоротших хеш-ланцюгів	11.10	22.74	7.86	17.24	10.27	15.77	9.67	14.44	13.64
"Жадібний" розклад LZPR, пов'язаний вибір найкоротших хеш-ланцюгів	11.04	22.19	7.96	17.14	10.38	15.87	9.50	14.56	13.58

Таблиця 4.4

Час декодування файлів зображень набору ACT з стандартного та модифікованого (з використанням алгоритму LZPR) формату PNG, с

Опис програми	№ файла								Середній час
	1	2	3	4	5	6	7	8	
Майже оптимальний розклад LZ77 після повного попереднього аналізу зображень	3.79	4.23	2.31	3.13	2.19	3.35	1.60	3.46	3.01
"Жадібний" розклад LZPR, всі варіанти вибору найкоротших хеш-ланцюгів	3.41	4.28	2.36	2.86	2.04	3.18	1.70	3.19	2.88

Дані цих таблиць свідчать, що, по-перше, у середньому застосування для тестових зображень контекстно-залежного алгоритму LZPR (рядок 2) у порівнянні з найкращим на сьогодні способом стиснення у стандарті формату PNG (рядок 1) зменшило КС на 2.11 %, прискорило кодування у 4.68 рази та декодування – на понад 4.3 %, що вказує на ефективність даного алгоритму та доцільність його застосування у наступних версіях цього формату. Розроблений алгоритм LZPR досягає значного покращення КС як внаслідок ефективного застосування переваг

контекстно-залежного словникового алгоритму, так і завдяки формуванню послідовності з меншою ентропією джерела, використовуючи *MedPredict,* перед застосуванням контекстно-незалежного алгоритму стиснення. І, по-друге, алгоритм врахування максимальної віднайденої довжини однакової послідовності у попередніх альтернативних словниках дає змогу додатково прискорити стиснення у середньому всього лише на 0.4 % (рядки 2, 3 з табл. 4.3). Він ефективний для синтезованих зображень, але, як правило, сповільнює компресію фотореалістичних зображень, ускладнює програмну реалізацію пошуку однакових послідовностей і тому не використовувався у подальших модифікаціях формату PNG.

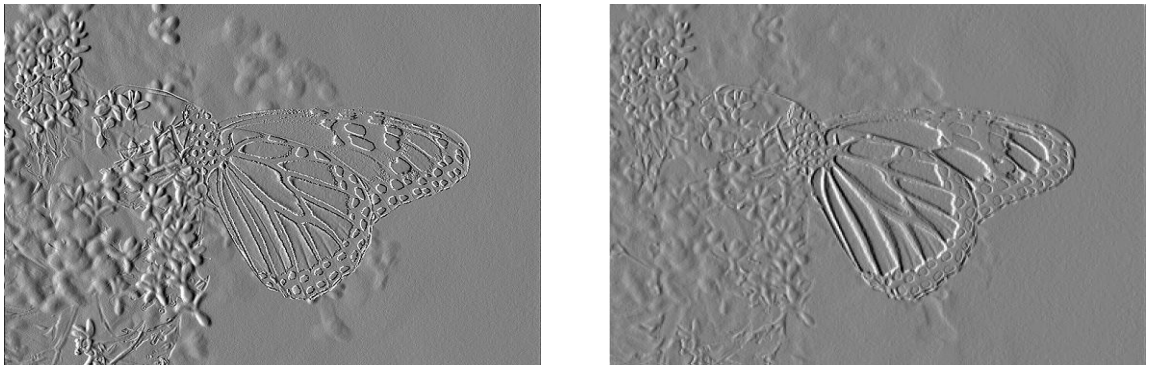
4.2. Застосування різницевих кольорових моделей для підвищення ефективності використання предикторів в процесі стиснення RGB-зображень без втрат

4.2.1. Доцільність застосування різницевих кольорових моделей. Сучасні формати стиснення RGB-зображень без втрат опрацьовують пікселі у фіксованій кольоровій моделі (наприклад, формати BMP – в моделі B, G, R; PNG – в моделі R, G, B; формат архіватора RAR – в моделі R-G, G, B-G) і не використовують можливості вибору ефективної кольорової моделі для кожного зображення, яка максимально зменшує ентропію за рахунок міжкомпонентної декореляції, адже різні компоненти зображення представляють достатньо подібні по геометрично-просторовій структурі дані (як, наприклад, на рис. 4.2).

а) б)

Рис. 4.2. Друга (а) та третя (б) компоненти зображення *Monarch.bmp* в кольоровій моделі RGB

Тому, враховуючи, що колір кожного піксела можна подати у вигляді координат по трьох лінійно-незалежних векторах будь-яких базових кольорів [36, с. 14], **метою підрозділу** є обґрунтування та опис алгоритму (вперше розглянутому у [55]) формування та переходу до альтернативної різницевої кольорової моделі для кожного RGB-зображення у форматах, що використовують предиктори (зокрема, у форматі PNG), з метою покращення КС за рахунок зменшення ентропії. Наприклад, зменшити ентропію третьої компоненти зображення *Monarch.bmp* у кольоровій моделі RGB, якщо використовується *Left*-предиктор, доцільно за допомогою переходу до кольорової моделі R, G-B, B-R (рис. 4.3).



а) б)

Рис. 4.3. Відхилення яскравостей суміжних пікселів третьої компоненти зображення *Monarch.bmp* після застосування *Left*-предиктора в RGB (а) та R, G-B, B-R (б) кольорових моделях

Замінювати окрему компоненту кольорової моделі лінійною комбінацією з іншою компонентою доцільно лише тоді, коли довжина ентропійного коду результатів дії обраного предиктора над комбінацією компонентів буде меншою за довжину ентропійного коду результатів дії обраного предиктора для даної компоненти, адже зменшення довжини ентропійного коду найчастіше призводить до зменшення КС. Для забезпечення однозначності декодування, в зображенні можна виконати **максимум дві** заміни значень різних компонентів різницями з іншими компонентами по всіх пікселах. Відповідно, виникає задача **вибору під час попередньої обробки зображень, серед можливих шести таких замін** (значень R_{ij} різницями R_{ij}-kG_{ij} або R_{ij}-kB_{ij}, значень G_{ij} різницями G_{ij}-kR_{ij} або G_{ij}-kB_{ij} та значень B_{ij} різницями B_{ij}-kR_{ij} або B_{ij}-kG_{ij}) **тих двох, які максимально зменшать ентропійну**

довжину коду результатів дії обраного предиктора. При цьому компоненту, що не підлягає перетворенням, як і в [12], назвемо яскравістю зображення. Розглянемо підходи до формування різницевих кольорових моделей як з дійсними, так і з цілими коефіцієнтами *k*.

4.2.2. Формування різницевих кольорових моделей з дійсними коефіцієнтами. Подальші пояснення зробимо на прикладі компоненти R_{ij} та її заміни лінійною комбінацією R_{ij}-kG_{ij} маючи на увазі, що запропоновані викладки можуть бути застосовані до будь-яких компонент та їх замін лінійними комбінаціями. Отож, виконувати заміну для всіх пікселів зображення значення компоненти R_{ij} лінійною комбінацією R_{ij}-kG_{ij} доцільно лише тоді, коли

$$W(k) = L(\Delta R - [kG]) < L(\Delta R), \tag{4.1}$$

де $\Delta R = \Delta R_{ij}$, $\Delta R - [kG] = \Delta R_{ij} - [kG_{ij}]$, $i = \overline{0, height-1}$, $j = \overline{0, width-1}$, *height* – кількість рядків зображення, *width* – кількість пікселів у кожному його рядку, а [.] відображає процес заокруглення до найближчого цілого, який зумовлений необхідністю оперування з дискретними значеннями яскравостей. З (4.1) слідує, що, для забезпечення максимального стиснення *k* слід обирати таким, щоб довжина ентропійного коду результатів застосування предиктора до лінійної комбінації компонентів пікселів зображення *W(k)* була мінімальною. Нехай цей мінімум досягається при $k = k_0$:

$$k_0 : L(\Delta R - [k_0 G]) = \min_k W(k), k \in \Re. \tag{4.2}$$

Оскільки лінійна комбінація кольорів фактично задає інший колір, то (4.2) відображає процес пошуку напрямку у площині *RG*, стосовно якого довжина ентропійного коду результатів застосування предикторів є мінімальною. Згідно цієї формули обчислення k_0 можна виконати методом **послідовного перебору значень**. Для безпосереднього розрахунку ентропійних довжин кодів компонентів та різниць компонентів в (4.1) та (4.2) після дії предиктора необхідно спочатку підрахувати частоти окремих елементів, а потім скористатися формулою (2.1).

Для уникнення використання методу перебору значень чи громіздких

алгоритмів інтерполяції та пошуку напрямку збіжності значення k_0 ми пропонуємо метод наближеного обчислення k_0 шляхом визначення **дисперсії** значень лінійних предикторів. Використаємо для модулювання значення Δ розподіл Лапласа, як це рекомендується в [37, с. 122-123]. Густина ймовірностей розподілу Лапласа задається виразом $f(x) = \frac{\lambda}{2} e^{-\lambda|x|}$, тобто нерівномірність цього розподілу зростає зі збільшенням λ. В свою чергу, λ виражається через дисперсію розподілу співвідношенням $\lambda = 2/\sqrt{D(X)}$. Якщо прийняти, що дані зображень після дії предикторів апроксимуються розподілом Лапласа, тоді зі зменшенням дисперсії нерівномірність розподілу частот значень компонентів реальних зображень навколо нуля мала б зростати, що, в свою чергу, відповідає зменшенню довжини ентропійного коду. Опираючись на ці викладки, наближене значення k_0 можемо визначити з умови

$$k_0 \approx \bar{k}_0 : D(R - \bar{k}_0 G) = \min_k V(k), k \in \Re, \tag{4.3}$$

де

$$V(k) = D(R - kG).$$

Дисперсія за умови нульового математичного сподівання визначається виразом

$$D(\xi) = \left(\sum_{i=0}^{height-1} \sum_{j=0}^{width-1} \xi_{ij}^2 \right) \Big/ (height \times width). \tag{4.4}$$

Відповідно, з (4.3) і (4.4) одержимо

$$V(k) = \left(\sum_{i=0}^{height-1} \sum_{j=0}^{width-1} (R_{ij} - kG_{ij})^2 \right) \Big/ (height \times width) \to \min. \tag{4.5}$$

Чисельник виразу (4.4) називають *енергією зображення* [37]. Отже вираз (4.5) відображає процес пошуку напрямку в площині *RG*, стосовно якого енергія приростів яскравостей є мінімальною. Визначимо $\bar{k}_0$ методом найменших квадратів. Враховуючи, що для лінійних предикторів

$$\Delta(R_{ij} - kG_{ij}) = \Delta R_{ij} - k\,\Delta G_{ij}, \tag{4.6}$$

прирівняємо похідну функції *V(k)* з (4.5) до нуля:

$$\frac{dV(k)}{dk} = -2\left(\sum_{i=0}^{height-1} \sum_{j=0}^{width-1} (\Delta R_{ij} - k\,\Delta G_{ij})\,\Delta G_{ij} \right) \Big/ (height \times width) = 0,$$

звідки

$$k_0 \approx \bar{k}_0 = \left(\sum_{i=0}^{height-1} \sum_{j=0}^{width-1} \Delta R_{ij}\,\Delta G_{ij} \right) \Big/ \left(\sum_{i=0}^{height-1} \sum_{j=0}^{width-1} (\Delta G_{ij})^2 \right). \qquad (4.7)$$

Для фотореалістичних зображень збільшення/зменшення яскравості однієї компоненти між суміжними пікселами найчастіше супроводжується збільшенням/зменшенням яскравостей інших відповідних компонентів. Тобто для таких зображень більшість ΔR_{ij}, ΔG_{ij} та ΔB_{ij} мають однаковий знак (їх попарні добутки додатні) та пропорційну величину і зменшувати дисперсію доцільно, насамперед, за рахунок цих пікселів. Тому для фотореалістичних зображень наближене значення k_0 визначимо також за формулою

$$k_0 \approx \bar{\bar{k}}_0 = \frac{\left(\sum_{i=0}^{height-1} \sum_{j=0}^{width-1} \Delta R_{ij}\,\Delta G_{ij} \mid \Delta R_{ij}\,\Delta G_{ij} > 0 \right)}{\left(\sum_{i=0}^{height-1} \sum_{j=0}^{width-1} (\Delta G_{ij})^2 \mid \Delta R_{ij}\,\Delta G_{ij} > 0 \right)}. \qquad (4.8)$$

Формули (4.7) та (4.8) обчислюють відношення перехресної кореляції приростів яскравостей до дисперсії приростів компоненти відповідно для всіх пікселів та пікселів, чиї зміни на зображенні мають однаковий характер для аналізованих компонентів. Після наближеного обчислення k_0 за допомогою (4.2), (4.7), чи (4.8) необхідно переконатися в ефективності отриманої лінійної комбінації компонентів, використовуючи вираз (4.1).

4.2.3. Формування різницевих кольорових моделей з цілими коефіцієнтами. Розглянемо алгоритм формування різницевих кольорових моделей для k_0=1 і лінійних предикторів, що збільшують нерівномірність розподілу частот елементів в основному за рахунок суміжних пікселів з близькими приростами яскравостей по різних компонентах (такі прирости яскравостей зустрічаються в

зображеннях найчастіше). Параметри таких моделей з цілими коефіцієнтами розраховуються найшвидше, оскільки не вимагають виконання операцій з дійсними числами та заокруглень. Ці ж причини дають змогу суттєво прискорити декодування. У цьому випадку необхідно оцінити доцільність заміни значень компоненти R_{ij} різницями R_{ij}-G_{ij} або R_{ij}-B_{ij}, значень G_{ij} різницями G_{ij}-R_{ij} або G_{ij}-B_{ij} та значень B_{ij} різницями B_{ij}-R_{ij} або B_{ij}-G_{ij}.

Крім цього, відсутність заокруглень дає змогу після вибору двох замін значень компонентів відповідними різницями обрати для компоненти яскравості зображення замість компоненти, що не замінювалася різницями, ту з вхідних компонентів, яка має найменшу довжину ентропійного коду після застосування предикторів, і цим покращити КС. Проте найчастіше обраною виявляється все таж компонента, що не замінювалася різницями (принаймні, на всіх досліджених нами зображеннях), тому такий додатковий аналіз лише сповільнює кодування/декодування та майже ніколи не зменшує КС і в пропонованому алгоритмі формування різницевих кольорових моделей не реалізовувався.

Запишемо досліджувані ентропійні довжини у вигляді матриці аналізу A:

$$A=\begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{pmatrix}=\begin{pmatrix} L(R) & L(R-\Delta G) & L(R-\Delta B) \\ L(G-\Delta R) & L(G) & L(G-\Delta B) \\ L(B-\Delta R) & L(B-\Delta G) & L(B) \end{pmatrix}. \quad (4.9)$$

Враховуючи (2.1), неважко пересвідчитись, що $a_{mn}=a_{nm}$, тобто для формування цієї матриці необхідно обчислити значення лише шести елементів. Нехай в межах кожного пікселя компонента R має індекс 0, G – 1, B – 2. Бачимо, що різницева кольорова модель визначається щонайбільше двома недіагональними елементами різних рядків матриці A, які серед елементів, менших за діагональні елементи своїх рядків сумарно найбільше від них відхиляються (забезпечують максимальні зменшення ентропійних довжин кодів). У випадку наявності таких елементів, індекс рядка кожного з них визначає зменшувану компоненту, а індекс стовпця – компоненту, яка від неї віднімається. Наприклад, вибір елемента a_{12} вказує, що в альтернативній кольоровій моделі для кожного пікселя зображення необхідно

значення компоненти *G* зменшити на значення компоненти *B*.

Покроково алгоритм формування та переходу до різницевої кольорової моделі з цілими коефіцієнтами перед стисненням зображення з використанням предикторів записується так:

1. Розрахувати для кожної компоненти всіх пікселів зображення результати дії обраного лінійного предиктора.

2. Обчислити ентропійні довжини кодів компонентів і різниць компонентів для результатів дії обраного предиктора та зберегти їх в матриці *A* згідно (4.9).

3. Занести значення 0 у змінні *index11*, *index12*, *index21* та *index22*, що визначають можливі різниці альтернативної кольорової моделі.

4. Визначити в матриці *A* два елементи, які не належать головній діагоналі, не симетричні відносно неї, менші за діагональні елементи своїх рядків, і сумарно найбільше від них відхиляються. Якщо такі елементи присутні, то занести в змінні *index11* та *index12* відповідно номер рядка та номер стовпця першого з цих елементів, а в змінні *index21* та *index22* – номер рядка та номер стовпця другого такого елемента. Інакше визначити в матриці *A* недіагональний елемент, менший за діагональний елемент свого рядка. Якщо і такий елемент відсутній, то перейти до кроку 8. Інакше занести в змінну *index11* номер рядка цього елемента, а в змінну *index12* – номер його стовпця.

5. Якщо значення змінних *index11* та *index22* однакові, то переставити значення змінних *index11* з *index21* та *index12* з *index22*, тобто змінити черговість віднімання компонентів кольорової моделі.

6. Зменшити для кожного піксела зображення значення компоненти з індексом *index11* на значення компоненти з індексом *index12*.

7. Якщо значення змінних *index21* та *index22* різні, тобто друга різниця кольорової моделі визначена, то зменшити для кожного піксела зображення значення компоненти з індексом *index21* на значення компоненти з індексом *index22*.

8. Завершити формування та перехід до різницевої кольорової моделі і виконати подальше стиснення перетвореного зображення з використанням

предикторів.

Для практичного формування альтернативних кольорових моделей ми обрали предиктор *LeftPredict*, оскільки він використовує лише значення компонентів попереднього піксела активного рядка і тому розраховується найшвидше, хоча з цією метою можна використовувати і інші лінійні предиктори. Зауважимо також, що, по-перше, під час декодування значення компонентів слід додавати у зворотній послідовності: спочатку до значень компоненти *index21* значення компоненти *index22*, а вже потім до значень компоненти *index11* значення компоненти *index12*, і, по-друге, якщо під час кодування доводиться відмовлятися від застосування предикторів, то слід також відмовитися і від використання альтернативної кольорової моделі, оскільки вона орієнтується саме на дію предикторів. Фрагменти програм для реалізації запропонованого алгоритму та повернення від різницевої кольорової моделі наведені у сьомому розділі додатка Ж.

4.2.4. Аналіз результатів застосування різницевих кольорових моделей. Проаналізуємо **результати застосування** запропонованих методів формування різницевих кольорових моделей для стиснення восьми файлів 24-бітних зображень стандартного набору ACT (див. табл. 1.3). Тестування проводили за допомогою базової програми з використанням алгоритмів, що реалізують ці методи, алгоритму попереднього аналізу зображень з розбиття на мінімальні та однорідні блоки рядків (див. підрозділ 3.5) і "лінивого" розкладу алгоритму LZ77. Визначені різними методами різницеві кольорові моделі наведені у табл. 4.5, КС файлів зображень подано у табл. 4.6, час кодування – у табл. 4.7, а час декодування – у табл. 4.8.

Дійсні коефіцієнти k_0 для компонентів різницевих кольорових моделей розраховували трьома способами (див. дані табл. 4.5 – 4.7): послідовним перебором (рядки 1, 2) значень інтервалу (0; 1.5] з кроком 0.05 згідно (4.2), з використанням дисперсії приростів яскравостей всіх пікселів (рядок 3) згідно (4.7) та дисперсії приростів яскравостей пікселів одного знаку (рядок 4) згідно (4.8).

Таблиця 4.5

Варіанти різницевих кольорових моделей для стиснення зображень набору ACT з застосуванням предикторів

Спосіб формування різницевої кольорової моделі	№ файла			
	1	**2**	**3**	**4**
По *LeftPredict* методом послідовного перебору	R,G-R,B	R-B,G-B,B	R,G-0.85R, B-0.75G	R-0.85B,G, B-0.95G
По *PaethPredict* методом послідовного перебору	R,G-R,B	R,G-B,B-R	R,G-0.85R, B-0.75G	R,G-R,B-G
По *LeftPredict* згідно (4.7)	R,G-0.31R, B-0.16R	R,G,B	R,G-0.85R, B-0.7G	R-0.84B,G, B-0.83G
По *LeftPredict* згідно (4.8)	R,G-0.79R,B	R,G-0.88B, B-0.93R	R,G-1.12R, B-0.83G	R-0.98B, G-1.01B,B
По *LeftPredict* для k_0=1	R,G-R,B	R-B,G-B,B з повер ненням до R,G,B	R,G-R,B-G	R,G-B,B-R

Продовж. табл. 4.5

Спосіб формування різницевої кольорової моделі	№ файла			
	5	**6**	**7**	**8**
По *LeftPredict* методом послідовного перебору	R,G-R,B-0.9G	R-G,G,B-G	R-G,G-B,B	R,G-R,B-G
По *PaethPredict* методом послідовного перебору	R-0.8G,G, B-0.95G	R-G,G-B,B	R-G,G-B,B	R,G-R,B-G
По *LeftPredict* згідно (4.7)	R-0.68G,G, B-0.79G	R-0.98G,G, B-0.93G	R,G,B	R,G-0.93R, B-0.96G
По *LeftPredict* згідно (4.8)	R,G-R, B-0.91G	R-0.99G,G, B-0.96G	R,G-0.89R,B	R,G-0.99R, B-G
По *LeftPredict* для k_0=1	R,G-R,B-G	R-G,G,B-G	R-G,G-B,B з повер ненням до R,G,B	R,G-R,B-G

Таблиця 4.6

КС файлів зображень набору ACT у форматі PNG після застосування різних кольорових моделей, %

Спосіб формування різницевої кольорової моделі	№ файла								Середній КС
	1	2	3	4	5	6	7	8	
По *LeftPredict* методом послідовного перебору	20.85	6.87	58.26	44.67	47.72	52.73	7.04	48.66	35.85
По *PaethPredict* методом послідовного перебору	20.85	6.87	58.26	44.41	47.59	52.82	7.04	48.66	35.81
По *LeftPredict* згідно (4.7)	20.94	6.74	58.00	45.10	47.85	53.43	7.04	48.92	36.00
По *LeftPredict* згідно (4.8)	20.85	6.82	58.52	44.67	47.72	52.99	7.10	48.66	35.92
По *LeftPredict* для k_0=1	20.85	6.87	58.78	44.41	47.72	52.73	7.04	48.66	35.88
По *LeftPredict* для k_0=1 з аналізом доцільності повернення до RGB	20.85	6.74	58.78	44.41	47.72	52.73	7.04	48.66	35.86
RGB (для порівняння)	20.94	6.74	60.21	52.04	53.84	65.74	7.04	57.50	40.51

Таблиця 4.7

Час кодування файлів зображень набору ACT з використанням різних кольорових моделей у формат PNG (в т.ч. перехід до моделі), с

Спосіб формування різницевої кольорової моделі	№ файла								Середній час
	1	2	3	4	5	6	7	8	
По *LeftPredict* методом послідовного перебору	463.68 (413.64)	780.82 (710.08)	171.53 (151.10)	275.12 (226.73)	179.88 (151.10)	264.69 (226.79)	317.75 (287.75)	269.41 (227.72)	340.36 (299.36)
По *PaethPredict* методом послідовного перебору	899.02 (848.66)	1527.09 (1457.17)	331.48 (311.04)	513.61 (466.48)	339.99 (310.88)	506.41 (468.13)	620.38 (590.40)	508.11 (466.65)	655.76 (614.93)
По *LeftPredict* згідно (4.7)	60.97 (11.21)	83.65 (14.83)	24.27 (4.12)	54.49 (6.09)	32.96 (4.07)	43.39 (6.16)	35.27 (6.04)	47.01 (6.10)	47.75 (7.33)
По *LeftPredict* згідно (4.8)	60.15 (9.67)	86.34 (18.51)	24.45 (4.17)	53.77 (6.21)	33.40 (4.17)	44.38 (6.37)	35.27 (6.59)	48.00 (6.15)	48.22 (7.73)
По *LeftPredict* для k_0=1	51.47 (1.32)	74.15 (2.52)	20.76 (0.55)	49.16 (0.82)	29.49 (0.49)	38.89 (0.82)	31.42 (0.99)	42.51 (0.83)	42.23 (1.04)
По *LeftPredict* для k_0=1 з аналізом доцільності повернення до RGB	51.47 (1.32)	73.82 (3.18)	20.76 (0.55)	49.16 (0.82)	29.49 (0.49)	38.89 (0.82)	31.31 (1.27)	42.51 (0.83)	42.18 (1.16)
RGB (для порівняння)	50.26	69.37	20.60	42.90	26.25	30.15	29.44	36.81	38.22

Таблиця 4.8

Час декодування файлів зображень набору ACT з використанням різних кольорових моделей з формату PNG (в т.ч. перехід від моделі), с

Спосіб формування різницевої кольорової моделі	№ файла								Середній час
	1	2	3	4	5	6	7	8	
Всі моделі з дійсними коефіцієнтами	4.18 (0.50)	5.83 (1.53)	2.80 (0.33)	3.74 (0.55)	2.47 (0.33)	3.90 (0.49)	2.36 (0.72)	3.79 (0.49)	3.63 (0.62)
По *LeftPredict* для k_0=1	3.90 (0.16)	4.72 (0.60)	2.36 (0.11)	3.24 (0.22)	2.25 (0.17)	3.52 (0.22)	1.87 (0.22)	3.41 (0.22)	3.16 (0.24)
По *LeftPredict* для k_0=1 з аналізом доцільності повернення до RGB	3.90 (0.16)	4.01 (0.00)	2.36 (0.11)	3.24 (0.22)	2.25 (0.17)	3.52 (0.22)	1.59 (0.00)	3.41 (0.22)	3.04 (0.14)
RGB (для порівняння)	3.79	4.01	2.31	3.13	2.25	3.24	1.59	3.46	2.97

Межі інтервалу для способу повного перебору обумовлені тим, що надзвичайно рідко зустрічаються змістовні зображення, в яких прирости яскравостей різних компонентів переважно мають різні знаки або відрізняються більш ніж в півтора рази. Для формування цих різницевих кольорових моделей ми використовували результати застосування нелінійного предиктора *PaethPredict* (рядок 2) та лінійного предиктора *LeftPredict* (рядки 1, 3, 4). Середній час декодування цими способами наведений у рядку 1 табл. 4.8.

Розраховані різницеві кольорові моделі з цілими коефіцієнтами для k_0=1 згідно розглянутого алгоритму та результати їх застосування наведено у рядках 5, 6 табл. 4.5 – 4.7, а відповідний час декодування – у рядках 2, 3 табл. 4.8. Результати тестування додатково порівнювали з результатами аналогічної програми у кольоровій моделі RGB (див. рядок 2 у табл. 3.18, 3.19, рядок 7 табл. 4.6, 4.7 та рядок 4 табл. 4.8). З метою демонстрації впливу різницевих кольорових моделей на стиснення синтезованих зображень № 2, 7 у рядках 1 – 5 табл. 4.6, 4.7 та рядках 1, 2 табл. 4.8 наведені дані тестування з використанням цих моделей, хоча для даних зображень застосування предикторів виявилося неефективним, що й зумовлює для них (див. останнє зауваження перед табл. 4.5) необхідність повернення до кольорової моделі RGB (див. відповідні дані у рядку 6 табл. 4.6, 4.7 та рядку 3

табл. 4.8).

Як свідчать дані рядків 1, 2 табл. 4.6, 4.7, для розрахунку різницевих кольорових моделей доцільно використовувати лінійні предиктори, які хоча й забезпечують гірші КС в середньому на 0.04 % від нелінійних, зате дають змогу прискорити кодування більш, ніж у два рази. Спосіб послідовного перебору (див. рядки 1 – 6 цих же таблиць) недоцільно використовувати на практиці, оскільки він хоча й здатний знизити КС на десяті долі процента відносно інших способів, але формує різницеві кольорові моделі повільніше від них у понад 7 разів.

Серед всіх способів формування різницевих кольорових моделей **найефективнішим виявився спосіб з цілими коефіцієнтами** (для k_0=1, реалізований згідно розглянутого алгоритму) **з додатковим аналізом доцільності повернення до кольорової моделі RGB**. Він дає змогу отримати близькі до мінімальних КС, забезпечує найшвидше кодування і, основне, декодування стосовно інших розглянутих способів (хоча для досягнення максимального стиснення у випадку відсутності обмежень по часу доцільно згенерувати різні різницеві кольорові моделі чотирма розглянутими способами і обрати серед них ту, що дозволяє одержати мінімальну довжину ентропійного коду). Тому розглянемо детальніше результати застосування цього найефективнішого способу формування різницевих кольорових моделей. Як видно з даних табл. 4.5, для стиснення різних зображень доцільно використовувати різні різницеві кольорові моделі, які розраховуються згідно розглянутого алгоритму. Найчастіше в різницевих кольорових моделях зменшується значення компонентів G та B, оскільки вони, як правило, мають найбільшу довжину ентропійного коду результатів дії предикторів. Внаслідок застосування різницевих кольорових моделей у середньому по набору АСТ КС зменшився на понад 4.65 % в основному за рахунок фотореалістичних зображень, для яких цей показник зменшився в середньому на 7.41 %. Аналогічне зменшення КС отримано як для інших зображень, так і для інших форматів (JPEG-LS, BMF), що не виконують міжкомпонентну декореляцію. Співставляючи дані табл. 1.3 та 4.6, приходимо до висновку, що ефективність застосування різницевих кольорових моделей для фотореалістичних зображень майже завжди підвищується

зі зменшенням відсотка унікальних кольорів. Застосування різницевих кольорових моделей з цілими коефіцієнтами може суттєво збільшити час кодування (у середньому – на 10.3 %, максимально – до 30 %) не лише внаслідок розрахунків параметрів кольорової моделі, а й через орієнтацію на контекстно-незалежний алгоритм, що обробляє окремі елементи. Зате час декодування зображень внаслідок використання цих кольорових моделей збільшується лише на десяті долі секунди, що в сукупності з суттєвим зменшення КС дає змогу ефективно використовувати їх на практиці.

Проаналізуємо також результати застосування різницевих кольорових моделей з цілими коефіцієнтами та алгоритму коригування значень предиктора [71; 72; 35] для стиснення цих же зображень з використанням алгоритму LZPR, описаному у попередньому підрозділі (табл. 4.9, 4.10, 4.11). Як свідчать дані цих таблиць, використання різницевих кольорових моделей суттєво не вплинуло на КС синтезованих зображень та зменшило КС фотореалістичних зображень у середньому на 5.29 %, сповільнивши при цьому кодування на 6.5 % та декодування на 7 %. Низька ефективність різницевих кольорових моделей для стиснення синтезованих зображень у цьому випадку пояснюється тим, що алгоритм LZPR виконує стиснення даних зображень в основному за рахунок однакових послідовностей пікселів у ковзаючих вікнах результатів використання лінійних предикторів *NonePredict*, *LeftPredict* та *RightPredict*, а застосування таких моделей, враховуючи співвідношення типу (4.6), ніяк не впливає на довжину віднайдених однакових послідовностей. Тому різницеві кольорові моделі дають змогу зменшити довжину ентропійного коду лише для літералів алгоритму LZPR, що є суттєвим лише для фотореалістичних зображень. Ось чому у випадку формування розкладу з незначною кількістю літералів (у наших експериментах – менше 2 % від загальної кількості яскравостей компонентів всіх пікселів зображення), зокрема і для синтезованих зображень, для прискорення декодування доцільно повернутися до кольорової моделі RGB та виконати стиснення ще раз (рядки 2, 3 табл. 4.10, 4.11). Такий додатковий аналіз доцільності повернення до кольорової моделі RGB хоча й у середньому сповільнює кодування на 24 %, але прискорює декодування на 3 %.

Таблиця 4.9

КС файлів зображень набору ACT у модифікованому за допомогою алгоритму LZPR форматі PNG після застосування комбінацій різницевих кольорових моделей та коригувань значень предиктора, %

Комбінація алгоритмів попереднього опрацювання	№ файла								Середній КС
	1	2	3	4	5	6	7	8	
Без алгоритмів попереднього опрацювання	20.13	5.69	60.47	47.27	51.24	58.63	6.01	54.73	38.02
Використання різницевих кольорових моделей	20.13	5.69	58.78	42.15	46.42	51.34	6.01	47.18	34.71
Коригування значень предиктора без аналізу доцільності їх застосування	20.42	5.72	58.13	46.75	49.80	58.54	6.01	53.51	37.36
Коригування значень предиктора після використання різницевих кольорових моделей з аналізом доцільності їх застосування	20.13	5.69	56.18	40.85	43.95	50.39	6.01	45.10	33.54

Таблиця 4.10

Час кодування файлів зображень набору ACT у модифікований за допомогою алгоритму LZPR форматі PNG після застосування комбінацій різницевих кольорових моделей та коригувань значень предиктора, с

Комбінація алгоритмів попереднього опрацювання	№ файла								Середній час
	1	2	3	4	5	6	7	8	
Без алгоритмів попереднього опрацювання	11.10	22.74	7.86	17.24	10.27	15.77	9.67	14.44	13.64
Використання різницевих кольорових моделей без аналізу доцільності повернення до RGB	12.08	24.94	8.24	17.74	10.55	17.09	10.49	15.05	14.52
Використання різницевих кольорових моделей з аналізом доцільності повернення до RGB	12.14	44.49	8.29	17.84	10.60	17.09	19.01	15.05	18.06
Коригування значень предиктора без аналізу доцільності їх застосування	11.69	23.95	8.13	17.68	10.55	16.21	10.21	14.88	14.16
Коригування значень предиктора після використання різницевих кольорових моделей з аналізом доцільності їх застосування	15.49	30.76	8.95	18.95	11.20	18.29	12.79	16.15	16.57

Таблиця 4.11

Час декодування файлів зображень набору ACT з модифікованого за допомогою алгоритму LZPR формату PNG після застосування комбінацій різницевих кольорових моделей та коригувань значень предиктора, с

Комбінація алгоритмів попереднього опрацювання	№ файла								Середній час
	1	2	3	4	5	6	7	8	
Без алгоритмів попереднього опрацювання	3.41	4.28	2.36	2.86	2.04	3.18	1.70	3.19	2.88
Використання різницевих кольорових моделей без аналізу доцільності повернення до RGB	3.57	4.78	2.47	2.97	2.20	3.41	1.98	3.24	3.07
Використання різницевих кольорових моделей з аналізом доцільності повернення до RGB	3.57	4.28	2.47	2.97	2.20	3.41	1.70	3.24	2.98
Коригування значень предиктора без аналізу доцільності їх застосування	4.89	6.86	2.75	3.62	2.53	3.95	2.85	3.85	3.91
Коригування значень предиктора після використання різницевих кольорових моделей з аналізом доцільності їх застосування	3.41	4.28	2.80	3.62	2.53	3.90	1.70	3.79	3.25

Коригування значень предиктора у середньому підвищило КС синтезованих зображень на 0.11 %, зменшило КС фотореалістичних зображень на 1.12 %, уповільнило кодування на 3.81 % та декодування – аж на 35.76 %. Цей метод виявився неефективним для даних синтезованих кольорових зображень, оскільки збільшує їх ентропію і нерідко зменшує довжини однакових послідовностей. Тому ми пропонуємо використовувати коригування значень предиктора лише тоді, коли вони зменшують прогнозовану ентропійну довжину коду зображення (2.1). Для прискорення декодування у випадку відмови від застосування цього методу (яке відбувається, як правило, для синтезованих зображень) ми також поверталися до кольорової моделі RGB (рядок 4 табл. 4.9 та рядок 5 табл. 4.10, 4.11). Такий додатковий аналіз доцільності застосування методів коригування значень предиктора та переходу до різницевих кольорових моделей хоча й у середньому сповільнює кодування ще на 17 %, але забезпечує незростання КС для всіх зображень та прискорює декодування на тих же 17 %. Додаткове зменшення КС

фотореалістичних зображень внаслідок сукупного застосування цих двох методів підвищення ефективності використання предикторів на 0.76 % пояснюється зменшенням енергії по двох перетворених компонентах різницевої кольорової моделі, що дає змогу ефективніше використовувати для них метод коригування значень предиктора.

Отже, метод коригування значень предиктора [71; 72; 35] варто застосовувати для стиснення резервних копій зображень у складі архіваторів (а не у графічних форматах) після алгоритму переходу до різницевих кольорових моделей, якщо таке використання зменшує прогнозовану ентропійну довжину коду (2.1), оскільки цей метод суттєво сповільнює декодування, хоча й зменшує КС фотореалістичних зображень. Алгоритми ж переходу до різницевих кольорових моделей та LZPR доцільно впровадити в наступні версії стандарту формату PNG, оскільки вони суттєво зменшують КС зображень, кардинально не впливаючи при цьому на час декодування.

В подальшому, для досягнення менших КС за допомогою ефективнішої міжкомпонентної декореляції**,** планується розробити алгоритм фрагментування зображень на області однакових різницевих кольорових моделей та алгоритм компактного зберігання таких областей.

4.3. Використання палітри для групового статистичного кодування трикомпонентних зображень без втрат

У цьому підрозділі обґрунтуємо можливість та опишемо алгоритм **попіксельного** (а **не покомпонентного**) застосування ентропійного кодування до результатів дії предикторів (1.4) над компонентами RGB-зображень з використанням палітри [51; 46; 48] та рівномірного кодування [14]. Оскільки рівномірне декодування виконується значно швидше від ентропійного, то така оптимізація дасть змогу прискорити роботу декодера загалом, а раціональне формування палітри дозволить, крім цього, покращити КС ентропійного кодування.

Безпосереднє попіксельне ентропійне кодування результатів дії предикторів згідно (1.4), малоефективне, оскільки значення яскравостей компонентів пікселів

після застосування предикторів (надалі у цьому підрозділі скорочено – *пікселів відхилень*) повторюються рідко і можуть знаходитися в межах [0; 256^3-1]. Тому ми пропонуємо **спочатку перетворити значення кольорів пікселів відхилень з використанням палітри** за таким алгоритмом:

1. Згрупуємо кольори пікселів відхилень по паралелепіпедах (тут і надалі – прямокутних), що не перетинаються.

2. Збережемо в палітрі найменші значення компонентів пікселів відхилень, що належать кожному з утворених паралелепіпедів та їх розміри по кожній осі.

3. Подамо значення кольору кожного піксела відхилень у вигляді індекса паралелепіпеда в палітрі, якому він належить, та зміщення по кожній координаті в середині цього паралелепіпеда.

Таке перетворення дає змогу **замість ентропійного кодування** трьох компонентів пікселів відхилень **використати групове кодування: ентропійне кодування** лише **індексів відповідних паралелепіпедів в палітрі** та **рівномірне кодування зміщень пікселів відхилень по кожній координаті в середині цих паралелепіпедів**.

Ефективне застосування групового статистичного кодування з використанням палітри можливе, насамперед, за рахунок надлишкової індексації кольорів у кольоровій моделі RGB: значення яскравості кожної компоненти цієї моделі для аналізованих нами зображень лежить у межах від 0 до 255. Отже загалом ця модель адресує $256 \times 256 \times 256 = 16777216$ кольорів і витрачає для збереження кольору (чи результату дії предиктора) піксела 24 біти. У кожному ж зображенні безпосередньо чи після застосування предикторів використовується лише невеличка частина спектру кольорів (наприклад, в зображеннях набору ACT після застосування предиктора *NonePredict* – менше 1 %, див. графу 6 з табл. 1.3). Максимальна кількість різних трикомпонентних кольорів (чи результатів дії предикторів) в зображенні у найгіршому випадку не перевищує кількості пікселів. Наприклад, у фотореалістичному зображенні *Lena.bmp* використовується 148279 кольорів 262144 пікселами, тобто лише 0,88% можливих кольорів моделі RGB. Розглядаючи навіть проекцію кольорів пікселів цього зображення на площину RG (рис. 4.4), можна

помітити, що використовується менше 30% індексованого простору площини. Групове статистичне кодування з використанням палітри індексує лише кольори в межах визначених паралелепіпедів, і тому витрачає для зберігання кольору кожного піксела, як буде показано далі, в середньому менше 24 бітів.

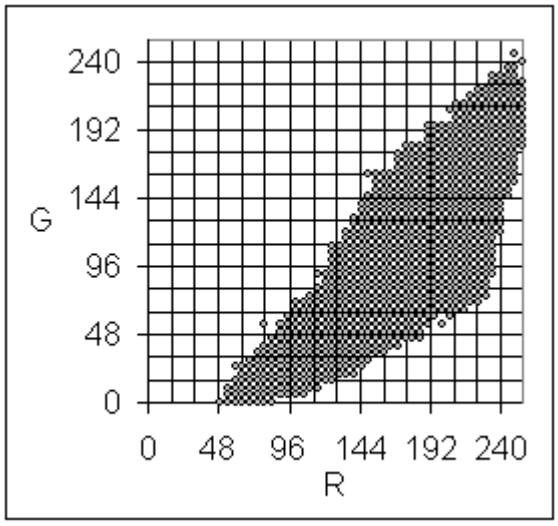


Рис. 4.4. RG-проекція кольорів пікселів зображення *Lena.bmp*

Розглянемо тепер питання формування палітри. Обмежимо максимальну кількість кольорів палітри значенням 256, щоб кардинально не змінювати внутрішню структуру формату DEFLATE.

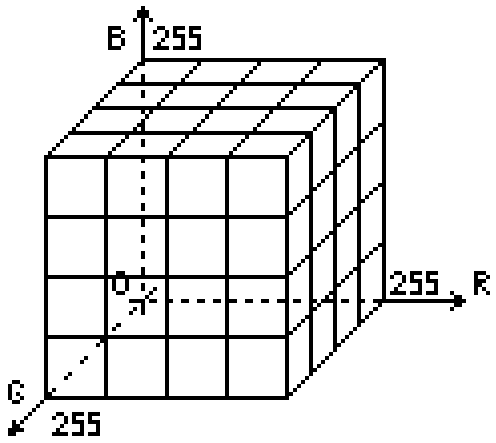


Рис. 4.5. Початкове розбиття простору кольорів RGB на 64 паралелепіпеди, що не перетинаються

Зрозуміло, що формування палітри, тобто, у нашому випадку, розбиття простору кольорів RGB (на рис. 4.5 – великий куб) на паралелепіпеди, має виконуватися

достатньо швидко та забезпечувати низький КС. Тому реалізуємо його в **два** етапи:

1.1. Спочатку виконаємо **швидке початкове розбиття** простору кольорів RGB **на паралелепіпеди, що не перетинаються**.

1.2. Потім **поділимо отримані паралелепіпеди** так, **щоб мінімізувати КС.**

Початкове розбиття простору кольорів RGB на паралелепіпеди, що не перетинаються, виконаємо за наступним алгоритмом.

1.1.1. Розіб'ємо множину допустимих значень кольорів на паралелепіпеди максимального фіксованого розміру (як в [51]), що не перетинаються між собою (наприклад, для RGB – це множина кубів). Оскільки наперед невідомо, в яких паралелепіпедах містяться кольори пікселів відхилень обраного зображення, то їх множина має повністю покривати множину допустимих значень кольорової моделі, а кількість не може перевищувати максимально допустимої кількості кольорів палітри.

1.1.2. Визначимо кількість кольорів пікселів відхилень та межі їх знаходження у кожному паралелепіпеді. Звузимо межі кожного паралелепіпеда до меж знаходження кольорів пікселів відхилень у ньому.

1.1.3. Поєднаємо між собою суміжні паралелепіпеди, якщо їх сукупний розмір не перевищує максимального фіксованого.

1.1.4. Збережемо в палітрі координати та розмірності лише тих паралелепіпедів, що містять кольори пікселів відхилень.

Як показують експерименти, початкове розбиття простору кольорів RGB доцільно виконувати на 64 паралелепіпеди максимального фіксованого розміру 64x64x64 значення (див. рис. 4.5), оскільки більші початкові розміри в кілька разів сповільнюють виконання алгоритму, а менші – значно погіршують КС. Кольори пікселів відхилень яскравостей реальних зображень від прогнозованих предикторами значень містяться, як правило, лише в частині з 64 початкових паралелепіпедів, переважно в тих, що розташовуються біля країв простору RGB.

Визначимо кількість бітів, необхідних для зберігання рівномірного коду зміщень пікселів відхилень у кожному паралелепіпеді. Нехай діапазон значень ребра чергового паралелепіпеда до поділу по осі R знаходиться в межах [*minR; maxR*], по

осі G – в межах [*minG; maxG*], а по осі B – в межах [*minB; maxB*]. Тоді для рівномірного кодування зміщень піксела відхилень по кожній координаті в середині паралелепіпеда у двійковій системі числення достатньо $\lceil \log(maxR - minR + 1) \rceil$ + $\lceil \log(maxG - minG + 1) \rceil$ + $\lceil \log(maxB - minB + 1) \rceil$ бітів. Наприклад, для рівномірного кодування зміщень у згаданому паралелепіпеді розмірами 64x64x64 значення достатньо 18 бітів. Запис індекса в палітрі з 64 кольорів рівномірними кодами навіть без ентропійного кодування потребує 6 бітів, тому для зберігання перетвореного значення кольору піксела відхилення достатньо 24 біти, як і у просторі RGB. Тобто навіть у випадку максимальних розмірів всіх паралелепіпедів (для зображень з хаотичними кольорами пікселів) розмір файла після групового кодування перевищить розмір вхідного файла лише на опис цих паралелепіпедів. Наприклад, для опису паралелепіпедів, що не перевищують згаданого максимального, достатньо 33 біти: 24 біти для опису базового кольору паралелепіпеда в палітрі і 9 бітів для опису його розмірів – по три для зберігання кількості бітів рівномірних кодів максимальних зміщень по кожній осі, значення яких знаходяться в діапазоні [0; 6].

1.2. Підвищити КС групового кодування можна за рахунок ефективного розбиття отриманих паралелепіпедів. З одного боку, навіть безпосереднє розбиття довільного паралелепіпеда навпіл зменшує в отриманих паралелепіпедах діапазон значень по осі поділу вдвічі, а, отже, кількість бітів для запису зміщень пікселів відхилень в них – на один. Досягнути додаткового зменшення розмірів паралелепіпедів в результаті поділу можна завдяки врахуванню положень зміщень пікселів відхилень в них. Проілюструємо це на прикладі RG-проекції зміщень кольорів пікселів відхилень окремого умовного паралелепіпеда (рис. 4.6). Розбиття такого паралелепіпеда площиною, перпендикулярною осі R, посередині ребра (в проекції – штрих-пунктирна лінія l_1) створить два паралелепіпеди: перший – в діапазоні [0; 15] і другий – в діапазоні [16; 31] по цій осі. Тобто сумарного зменшення діапазону значень від такого поділу не відбудеться. Якщо ж розбити цей паралелепіпед аналогічною площиною по значенню R=7 (в проекції – пунктирна лінія l_2), то можемо створити лівий паралелепіпед в діапазоні [0; 5] та правий – в діапазоні [10; 31] по цій осі, що дозволить зменшити сумарний діапазон значень на 4

одиниці.

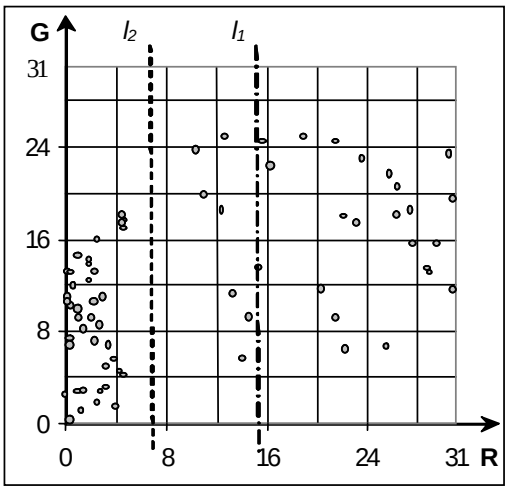


Рис. 4.6. RG-проекція зміщень кольорів пікселів відхилень окремого умовного паралелепіпеда

Позначимо через *countPoint* загальну кількість пікселів відхилень у черговому паралелепіпеді, через *countR(j)* – кількості зміщень, що мають значення *j* по осі R, через $\underline{i}$ – праву межу лівого, а через $\bar{i}$ – ліву межу правого з отриманих паралелепіпедів при поділі по значенню *i* осі R. Очевидно, що

$$countPoint = \sum_{j=minR}^{maxR} countR(j),\ \underline{i} = \max_{\substack{j=minR,\ i \\ countR(j)>0}} j,\ \bar{i} = \min_{\substack{j=\ i+1,\ maxR \\ countR(j)>0}} j.$$

Тоді зменшення довжини (кількості бітів) рівномірного коду зміщень пікселів відхилень паралелепіпеда внаслідок поділу по значенню *i* осі R становитиме

$$\nabla'_{R_i} = countPoint \times \lceil \log(maxR - minR + 1) \rceil - \left(\sum_{j=minR}^{\underline{i}} countR(j) \times \lceil \log(\underline{i} - minR + 1) \rceil + \sum_{j=\bar{i}}^{maxR} countR(j) \times \lceil \log(maxR - \bar{i} + 1) \rceil \right). \quad (4.10)$$

З іншого боку, розбиття довільного паралелепіпеда призводить до створення нового кольору в палітрі і поділу його пікселів відхилень між двома утвореними паралелепіпедами, тобто зменшує нерівномірність розподілу частот елементів, внаслідок чого збільшує ентропію джерела (згідно (1.3)) та загальну довжину коду

індексів пікселів відхилень. Тому збільшення дожини ентропійного коду індексів паралелепіпедів (згідно (2.1)) внаслідок поділу по значенню *i* осі R складе

$$\nabla''_{R_i} = countPoint \times \log(countPoint) - \\ -\left(\sum_{j=minR}^{i} countR(j) \times \log\left(\sum_{j=minR}^{i} countR(j) \right) + \sum_{j=i}^{maxR} countR(j) \times \log\left(\sum_{j=i}^{maxR} countR(j) \right) \right). \quad (4.11)$$

Зменшення довжини групового коду пікселів відхилень внаслідок поділу по значенню *i* осі R обчислимо з різниці зменшення рівномірного (4.10) та збільшення ентропійного (4.11) кодів:

$$\nabla_{R_i} = \nabla'_{R_i} - \nabla''_{R_i} = countPoint \times \left(\log(maxR - minR + 1) - \log(countPoint) \right) - \\ - \sum_{j=minR}^{i} countR(j) \times \left(\log(i - minR + 1) - \log\left(\sum_{j=minR}^{i} countR(j) \right) \right) - \quad (4.12) \\ - \sum_{j=i}^{maxR} countR(j) \times \left(\log(maxR - i + 1) - \log\left(\sum_{j=i}^{maxR} countR(j) \right) \right).$$

Максимальне зменшення довжини групового коду внаслідок розбиття паралелепіпеда по осі R визначимо з умови максимуму цієї величини для всіх можливих поділів по даній осі:

$$\nabla_R = \max_{i=minR,\ maxR-1} \nabla_{R_i}, \quad (4.13)$$

а внаслідок довільного поділу паралелепіпеда – з умови максимуму по всіх осях:

$$\nabla = \max\left(\nabla_R, \nabla_G, \nabla_B \right). \quad (4.14)$$

Фрагмент програми для визначення максимального зменшення довжини групового коду чергового паралелепіпеда наведено у восьмому розділі додатка Ж.

Максимальне зменшення КС досягається за умови максимального зменшення довжини групового коду. Тому **поділ паралелепіпедів** і, відповідно, розширення палітри **виконуються ітеративно**: під час кожної ітерації визначається максимальне значення ∇ серед всіх паралелепіпедів згідно (4.14) та виконується

поділ того паралелепіпеда і тією площиною, де цей максимум досягається. Оскільки на зберігання опису паралелепіпеда витрачається 33 біти, то ітеративний процес припиняється, коли максимальне зменшення довжини групового коду стане меншим від цього значення або розмір палітри досягне максимально можливого значення і не виявиться пари паралелепіпедів, збільшення довжини коду від поєднання яких буде меншим від максимального зменшення довжини групового коду (у випадку заповнення палітри та наявності таких пар необхідно перед розбиттям ітерації поєднати два паралелепіпеди з найменшим збільшенням довжини групового коду).

Розглянемо тепер практичні аспекти реалізації знаходження кращого розбиття паралелепіпеда площинами, перпендикулярними його осям, на прикладі осі R, що дозволяють істотно (до 5%) прискорити виконання обчислень:

1. Для всіх таких *i*, що *countR(i)*=0, $\nabla_{R_i} = \nabla_{R_{i-1}}$, тому проводити для них розрахунки зменшення довжини групового коду згідно (4.13) не потрібно.

2. Послідовно змінюючи *i*, перераховувати щоразу дві внутрішні суми у (4.12) не потрібно, адже відносно попередньої перша сума збільшується на *countR(i)*, а друга – зменшується на цю ж величину.

3. Під час обчислень двійкових логарифмів згідно (4.12), які, як правило, не реалізовані на рівні мови програмування, замість класичної формули $\log_2(x) = \ln(x)/\ln(2)$ доцільно використати $\log_2(x) = \ln(x) \times 1.4426950409$, що зменшить кількість викликів вбудованих функцій та складність виконання обчислень.

4. Для реалізації функції $\lceil \log(x) \rceil$ замість використання вбудованих функцій логарифмування та заокруглення до більшого числа доцільно здійснити лише реагування на порогові значення. Мовою С таке реагування може бути реалізоване, наприклад, так:

```
int countBit(UBYTE4 diapazon)
 {if (diapazon<=1) return 0;
  if (diapazon==2) return 1;
  if (diapazon<=4) return 2;
  if (diapazon<=8) return 3;
  ... } ;   .
```

5. Краще розбиття кожного зі створених паралелепіпедів не впливає на

значення ∇ для інших паралелепіпедів, тому після кожної ітерації алгоритму палітрування додатково розраховувати максимальне зменшення довжини групового коду необхідно лише для двох паралелепіпедів, що утворилися в результаті поділу.

Розглянемо **результати застосування** описаного алгоритму групового статистичного кодування для стиснення зображень набору АСТ. Тестування проводили за допомогою модифікованої базової програми, яка послідовно опрацьовує дані в такій послідовності: перехід до альтернативної різницевої кольорової моделі згідно алгоритму підрозділу 4.2; застосування предиктора *MedPredict* [73] та використання (у разі доцільності, як у попередньому підрозділі) методу коригування його значень [71; 72; 35]; перетворення значення пікселів відхилень з використанням палітри згідно описаного алгоритму; кодування для пікселів відхилень індексів відповідних паралелепіпедів префіксними кодами HUFF та зміщень – рівномірними кодами. Результати безпосереднього застосування алгоритму групового статистичного кодування з різними варіантами палітрування наведено в табл. 4.12, 4.13 та табл. 4.14.

Таблиця 4.12

Розміри групових (ентропійних + рівномірних) кодів файлів зображень набору АСТ після застосування різних варіантів модифікованого алгоритму палітрування, Кб

Варіант палітрування	№ файла								Середній КС, %
	1	2	3	4	5	6	7	8	
Без поділу паралелепіпедів	1726 (151+ +1575)	2948 (231+ +2717)	675 (98+ +577)	953 (138+ +815)	662 (96+ +566)	989 (142+ +847)	1196 (98+ +1098)	1000 (143+ +857)	84.28
З поділом паралелепіпедів	415 (226+ +189)	522 (308+ +216)	430 (233+ +197)	451 (329+ +122)	330 (230+ +100)	536 (349+ +197)	197 (122+ +75)	509 (352+ +157)	34.52

Таблиця 4.13

Час кодування (в т. ч. палітрування) файлів зображень набору ACT програмами для різних варіантів модифікованого алгоритму палітрування, с

Варіант палітрування	№ файла								Середній час
	1	2	3	4	5	6	7	8	
Без поділу паралелепіпедів	15.49 (0.66)	27.40 (1.04)	5.60 (0.28)	8.02 (0.44)	5.38 (0.27)	8.29 (0.44)	10.82 (0.44)	8.23 (0.44)	11.15 (0.50)
З поділом паралелепіпедів	24.83 (10.10)	60.03 (34.55)	13.24 (7.85)	16.20 (8.35)	12.69 (7.47)	15.54 (7.58)	32.35 (22.19)	18.40 (10.43)	24.16 (13.57)

Таблиця 4.14

Час декодування файлів зображень набору ACT у випадку застосування різних варіантів модифікованого алгоритму палітрування, с

Варіант палітрування	№ файла								Середній час
	1	2	3	4	5	6	7	8	
Без поділу паралелепіпедів	4.78	8.02	2.47	3.57	2.41	3.62	3.30	3.62	3.97
З поділом паралелепіпедів	3.68	5.82	2.47	3.41	2.31	3.52	2.36	3.62	3.40

Як свідчать дані цих таблиць, поділ окремих паралелепіпедів хоча й у середньому сповільнює кодування на 116 %, проте прискорює декодування на 14.4 % і, головне, зменшує КС на 49.76 %, причому покращення стиснення відбувається за рахунок кардинального зменшення довжини рівномірних кодів.

Описаний алгоритм групового статистичного кодування за допомогою палітри опрацьовує окремі піксели відхилень, а отже належить до класу контекстно-незалежних. Тому розглянемо також результати його застосування (табл. 4.15, 4.16, 4.17) після додаткового попереднього використання контекстно-залежного алгоритму LZPR (див. підрозділ 4.1) та алгоритму мінімізації розміру стиснутих блоків (див. підрозділ 3.1).

Таблиця 4.15

КС файлів зображень набору ACT після застосування різних варіантів модифікованих алгоритмів LZPR та палітрування, %

Варіант програми	№ файла								Середній КС
	1	2	3	4	5	6	7	8	
"Жадібний" розклад LZPR без палітрування	20.13	5.69	56.18	40.85	43.95	50.39	6.01	45.10	33.54
"Жадібний" розклад LZPR з палітруванням	17.37	5.66	55.53	38.77	42.65	46.40	6.08	43.97	32.05
Майже оптимальний розклад LZPR з палітруванням	17.28	5.22	55.53	38.42	42.52	46.40	5.67	43.89	31.87

Таблиця 4.16

Час кодування файлів зображень набору ACT програмами для різних варіантів модифікованих алгоритмів LZPR та палітрування, с

Варіант програми	№ файла								Середній час
	1	2	3	4	5	6	7	8	
"Жадібний" розклад LZPR без палітрування	15.49	30.76	8.95	18.95	11.20	18.29	12.79	16.15	16.57
"Жадібний" розклад LZPR з палітруванням	24.22	30.49	16.53	27.19	19.06	25.65	13.02	26.58	22.84
Майже оптимальний розклад LZPR з палітруванням	78.98	243.21	15.60	36.63	20.54	27.35	99.69	28.78	68.85

Таблиця 4.17

Час декодування файлів зображень набору ACT для різних варіантів модифікованих алгоритмів LZPR та палітрування, с

Варіант програми	№ файла								Середній час
	1	2	3	4	5	6	7	8	
"Жадібний" розклад LZPR без палітрування	3.41	4.28	2.80	3.62	2.53	3.90	1.70	3.79	3.25
"Жадібний" чи майже оптимальний розклади LZPR з палітруванням	3.08	4.23	2.47	3.40	2.36	3.51	1.70	3.52	3.03

Дані цих таблиць засвідчують, що в середньому по набору зображень ACT

використання палітри після контекстно-залежного алгоритму LZPR хоча й сповільнило кодування на 38 %, але прискорило декодування на 7 % та зменшило КС на 1.49 %. Застосовувати на практиці майже оптимального розкладу LZPR замість "жадібного" у цьому випадку недоцільно, оскільки таке використання хоча й зменшує у середньому КС лише на 0.18 % (в основному за рахунок синтезованих зображень), але сповільнює кодування у три рази. Найефективнішим застосування палітри виявилося в процесі стиснення синтезованих з перешкодами та фотореалістичних зображень. Використання ж даного алгоритму для кодування синтезованих зображень без перешкод хоча й може дещо прискорити декодування, але нерідко призводить до незначного погіршення КС (див. дані зображення 7 у табл. 4.15), оскільки вимагає зберігання параметрів палітри і застосовується лише до окремих літералів алгоритму LZPR. У середньому, як слідує з даних табл. 4.9, 4.11, 4.15 і 4.17, застосування палітри покращує КС аналогічно алгоритму коригування значень предиктора [71; 72; 35], але на відміну від нього прискорює декодування, і тому може широко застосовуватися у форматах графічних файлів.

4.4. Аналіз результатів сукупного використання модифікацій формату PNG. Застосування розробленого комплексу програм ModifyPNG

На завершення проаналізуємо результати сукупного використання досліджених модифікацій формату PNG для стиснення зображень наборів ACT та KTCI (табл. 4.18 – 4.20 та другий розділ додатка Е). Розширений аналіз результатів окремого застосування цих модифікацій та їх комбінацій наведено в [6]. Зображення цих наборів стискувалися базовою програмою (рядок 1) та програмою, що послідовно застосовує різницеві кольорові моделі з цілими коефіцієнтами і алгоритми коригування значень предиктора [71; 72; 35], LZPR, мінімізації розміру стиснутих блоків та палітрування (рядок 2). Для порівняння в цих же таблицях наведені результати тестування популярного архіватора RAR v. 3.00 (рядок 3), який використовує для зменшення міжелементної надлишковості той самий алгоритм LZ77 [78], та програми ERI v. 5.1 (рядок 4), яка застосовує алгоритм паралельного сортування блоків [24; 33] та забезпечує найкращий на сьогодні КС по набору

зображень ACT (за даними http://www.compression.ru/arctest/act/act-tif.htm).

Таблиця 4.18

КС файлів зображень набору ACT після застосування різних програм, %

Програма	№ файла								Середній КС
	1	2	3	4	5	6	7	8	
Базова	23.94	10.27	67.75	55.85	58.65	67.65	10.38	61.23	44.47
Базова з модифікаціями	17.37	5.66	55.53	38.77	42.65	46.40	6.08	43.97	32.05
RAR v. 3.0	23.94	5.63	61.51	44.84	47.98	50.91	6.08	51.00	36.49
ERI v. 5.1	16.52	5.19	60.08	38.33	43.69	46.05	5.33	44.15	32.42

Таблиця 4.19

Час кодування файлів зображень набору ACT різними програмами, с

Програма	№ файла								Середній час
	1	2	3	4	5	6	7	8	
Базова	10.76	15.27	4.78	9.83	5.99	7.90	6.10	8.73	8.67
Базова з модифікаціями	24.22	30.49	16.53	27.19	19.06	25.65	13.02	26.58	22.84
RAR v. 3.0	5.88	7.31	1.98	3.08	1.98	3.90	3.18	2.91	3.78
ERI v. 5.1	3.95	5.27	3.51	3.68	2.96	4.17	2.31	4.07	3.74

Таблиця 4.20

Час декодування файлів зображень набору ACT різними програмами, с

Програма	№ файла								Середній час
	1	2	3	4	5	6	7	8	
Базова	4.17	5.82	2.47	3.35	2.31	3.51	2.41	3.52	3.45
Базова з модифікаціями	3.08	4.23	2.47	3.40	2.36	3.51	1.70	3.52	3.03
RAR v. 3.0	0.55	2.91	0.44	0.71	0.43	0.66	1.21	0.66	0.95
ERI v. 5.1	4.56	5.05	3.79	4.07	3.08	4.51	1.93	4.34	3.92

Дані цих таблиць свідчать, що застосування досліджених модифікацій в середньому зменшує КС на понад 12 % та суттєво не впливає на час декодування, хоча й сповільнює кодування у 2.63 – 3.18 рази. Розроблена програма з зазначеними

модифікаціями на тестових наборах ACT та KTCI хоча й у середньому кодує повільніше від альтернативних програм у 6 – 8 разів та декодує довше від RAR v. 3.0 у 3.3 – 5 разів, але забезпечує швидшу декомпресію стосовно програми ERI v. 5.1 на 17 – 23 % та **дає змогу досягнути найменших на сьогодні середніх КС** в основному за рахунок фотореалістичних зображень з високим рівнем (понад 30 %, див. табл. 1.3 та 4.18) унікальних кольорів.

Для стиснення файлів зображень з застосуванням досліджених модифікацій нами розроблений комплекс програм ModifyPNG на базі програм з CD до [25]. У цьому комплексі, зокрема, міститься реалізація методу генерування та переходу до альтернативних різницевих кольорових моделей з цілими коефіцієнтами для кожного зображення, яка захищена авторським свідоцтвом [2]. Виконувані файли та вихідні тексти комплексу ModifyPNG розповсюджуються як безкоштовне сервісне програмне забезпечення і доступна для завантаження з Web-сторінки http:\\apserver.org.ua/peregl.php?=5. Цей комплекс програм можна застосовувати в операційних системах сімейств DOS та Widows за умов дотримання положень "Ліцензійної угоди" та наявності вільної оперативної чи дискової пам'яті з розміром, достатнім для розміщення чотирьох вхідних файлів зображень у форматі BMP.

Комплекс складається з двох виконуваних файлів, які завантажуються з командного рядка: кодера *MPngEnc.exe* та декодера *MPngDec.exe*.

Виконуваний файл *MPngEnc.exe* завантажується у форматі

MPngEnc [-I -F -M -P -R] inputFile.bmp outputFile.lsg.

Основні аргументи цієї програми такі: inputFile – назва довільного вхідного файла з розширенням та типу BMP для стиснення; *outputFile* – назва вихідного файла з розширенням LSG для збереження результатів стиснення. В процесі виконання програми створюється також файл *outputFile.pls*, у якому зберігаються рівномірні коди зміщень яскравостей по кожній компоненті у відповідних паралелепіпедах, індексованих палітрою, для незакодованих алгоритмом LZPR пікселів після застосування предикторів. Параметри програми *MPngEnc.exe* (вказуються необов'язково) використовуються для розширення її функціональності та задаються довільними комбінаціями з наступного переліку: -I – виводити додаткову

інформацію про хід процесу стиснення; -F – аналізувати лише перші елементи хеш-ланцюгів в процесі виконання алгоритму LZ77; -M – аналізувати всі елементи хеш-ланцюгів в процесі виконання алгоритму LZ77; -P – не використовувати предиктори; -R – не аналізувати ефективність замін алгоритму LZ77 у DEFLATE-блоках.

Виконуваний файл *MPngDec.exe* завантажується у форматі

MPngDec [-V] inputFile.lsg outputFile.bmp,

де необов'язковий параметр –V дає змогу вивести додаткову інформацію про хід процесу декодування. Основні аргументи цієї програми такі: *inputFile* – назва довільного вхідного файла з розширенням LSG для декодування. Успішне виконання програми можливе лише, коли у папці даного файла міститися також відповідний файл *inputFile.pls*; *outputFile* – назва вихідного файла з розширенням та типу BMP для збереження результатів декодування.

Комплекс програм ModifyPNG дозволяє досягнути низьких КС зображень, і тому може використовуватися як самостійна утиліта-архіватор чи, зважаючи на відкритість коду, у складі іншого службового та прикладного програмного забезпечення.

Як показали результати додаткових експериментів, досягнути ще менших КС на 0.3 – 1 % можливо за допомогою використання для алгоритму LZPR майже оптимального розкладу замість "жадібного" (див. табл. 4.15) та адаптації до контекстно-незалежного кодування алгоритму коригування значень предиктора [35, 71; 72], але такі модифікації сповільнюють кодування більш ніж в три рази.

4.5. Висновки до четвертого розділу

1. Застосування контекстно-залежного алгоритму LZPR замість алгоритму LZ77 в процесі стиснення кольорових зображень дає змогу суттєво зменшити їх КС (наприклад, по набору ACT – в середньому на 2.11 % за даними табл. 4.2) істотно не впливаючи при цьому на час декодування, за рахунок синхронного використання декількох ковзаючих вікон (для відхилень різних предикторів) замість одного та формування послідовності літералів з меншою ентропією перед застосуванням

контекстно-незалежного алгоритму. Це вказує на доцільність застосування подібних модифікацій у графічних форматах, які використовують ідеї словникових методів стиснення.

2. Підвищити КС зображень в трикомпонентних кольорових моделях можна не лише за рахунок декореляції даних окремих компонентів, а й за допомогою міжкомпонентної декореляції. Таку декореляцію слід виконувати так, щоб підсилити прояви властивостей зображення, що використовуються алгоритмами препроцесингу та безпосереднього стиснення обраного графічного формату. У випадку використання неадаптивних предикторів для стиснення зображень без втрат виконувати міжкомпонентну декореляцію доцільно за допомогою різницевих кольорових моделей, застосування яких дає змогу покращити КС фотореалістичних зображень в середньому на понад 7 %, а максимально – на понад 12 % (див., наприклад, дані зображення № 6 в табл. 4.6). Ефективність застосування різницевих кольорових моделей для таких зображень майже завжди підвищується зі зменшенням % унікальних кольорів. Формування різницевих кольорових моделей відображає процес пошуку кольорів, стосовно яких енергія приростів яскравостей є мінімальною. На практиці доцільно використовувати різницеві кольорові моделі з цілими коефіцієнтами, які формуються згідно запропонованого у пункті 4.2.3 алгоритму. Такі кольорові моделі хоча й не гарантують досягнення мінімальних КС для всіх зображень, проте забезпечують максимальну швидкість кодування і, головне, декодування, тому їх доцільно впровадити в наступні версії форматів, що використовують неадаптивні предиктори та не виконують міжкомпонентну декореляцію (PNG, JPEG-LS, BMF та інші), на рівні стандартів.

3. Формування ефективної палітри для групового статистичного кодування можливе лише за умови врахування довжини як ентропійних, так і рівномірних кодів. Групове статистичне кодування дає змогу досягнути кращих КС у порівнянні з результатами кодування HUFF (див. табл. 4.15) за рахунок зменшення надлишкової індексації кольорів. Застосування алгоритму групового статистичного кодування з допомогою палітри замість ентропійного кодування хоча й значно сповільнює кодування, але дає змогу суттєво зменшити час декодування (див.

табл. 4.16, 4.17) за рахунок введення рівномірних кодів, і тому може бути рекомендоване для комплексного використання у форматах стиснення зображень без втрат.

4. Комплексне застосування розроблених модифікацій формату PNG сумісно з алгоритмами коригування значень предиктора та мінімізації розміру стиснутих блоків дає змогу досягнути в середньому найменших на сьогодні КС без втрат в основному за рахунок фотореалістичних зображень з високим рівнем унікальних кольорів, зокрема по набору ACT (див. табл. 4.18), суттєво не впливаючи на час декодування, оскільки дані модифікації орієнтовані на підвищення ефективності різних способів кодування цього формату: різницеві кольорові моделі дають змогу зменшити ентропію після застосування предикторів, алгоритм LZPR вдосконалює механізм дії алгоритму LZ77, а палітрування використовується для покращення показників компресії HUFF. Саме це створює передумови для успішного сумісного застосування досліджених модифікацій в інших графічних форматах та архіваторах.

ВИСНОВКИ

В дисертації вирішена актуальна науково-практична задача підвищення ефективності стиснення без втрат у растрових графічних форматах (зокрема, і в форматі PNG), що використовують предиктори, словниковий алгоритм LZ77, контекстно-незалежне кодування та їх комбінування за допомогою вдосконалення і врахування взаємодії цих та застосування альтернативних чи нових методів і алгоритмів кодування. При цьому отримані наведені нижче основні наукові і практичні (в середньому по набору зображень АСТ) результати:

1. Вдосконалено метод пошуку однакових послідовностей потоку шляхом вибору найкоротших хеш-ланцюгів, що дало змогу, наприклад, прискорити формування розкладу алгоритму LZ77 для випадку аналізу всіх елементів цих ланцюгів у понад 13 разів.

2. Вперше запропоновано і реалізовано післяпроцесний метод зменшення розміру стиснутого блоку у форматі DEFLATE за допомогою: розрахунку розподілів частот для визначення розмірів альтернативних стиснутих блоків без їх попередньої генерації; вибору найкоротшого блоку з альтернативних; ітеративного відкидання неефективних замін у найкоротшому стиснутому блоці з наступним його формуванням. Реалізація цього методу хоча й сповільнює кодування на 10 %, але дозволяє зменшити КС переважної більшості синтезованих з шумами та фотореалістичних зображень на 2 – 6 %. В процесі реалізації даного методу вдосконалено метод точного розрахунку розмірів блоків динамічних кодів HUFF за допомогою врахування принципу формування цих кодів без їх безпосередньої генерації, що дало змогу прискорити виконання таких обчислень на понад 85 %.

3. Вдосконалено механізм формування "лінивого" / майже оптимального розкладів алгоритму словникового стиснення LZ77 шляхом використання результатів попереднього аналізу зображень для прогнозування довжин кодів елементів, що дало змогу зменшити КС на 0.05 % / 0.16 %, хоча й сповільнило кодування на 30 % / 8.3 %.

4. Проведено аналіз ефективності стиснення у випадку застосування

різноманітних предикторів, існуючих і розроблених ентропійних та емпіричних способів вибору предикторів для окремих рядків, різних способів кодування для різнотипних зображень за результатами якого виділено п'ять варіантів компресії, кожен з яких може виявитися оптимальним для чергового блоку рядків: без використання предикторів, з застосуванням *LeftPredict*, з використанням *RightPredict*, з застосуванням безпосереднього ентропійного способу вибору предикторів та з використанням ентропійного способу вибору предикторів після застосування коротких замін алгоритму LZ77.

5. Вперше розроблено метод попереднього аналізу зображень з розбиттям на мінімальні та однорідні блоки рядків з метою визначення для кожного з них оптимального варіанту компресії (з п'яти наведених у попередньому пункті) за допомогою методу динамічного програмування, реалізація якого у сукупності з методами та способами попередніх пунктів хоча й сповільнює кодування стосовно швидких варіантів стиснення у форматі PNG у понад 4.5 рази, але зменшує КС більш ніж на 3.96 % та досягає по цьому показнику найменших значень серед програмного забезпечення, що стискує зображення у форматі PNG.

6. Вдосконалено структуру розкладу алгоритму словникового стиснення LZ77 (алгоритм LZPR) шляхом використання декількох ковзаючих вікон та формування літералів з найменшою ентропією, що дало змогу зменшити КС на понад 2.1 %.

7. Вперше розроблено ряд методів для генерування різницевих кольорових моделей як з цілими, так і з дійсними коефіцієнтами, орієнтованих на зменшення довжини контекстно-незалежного коду у випадку використання предикторів. Застосування цих моделей дає змогу зменшити КС фотореалістичних зображень на понад 7 % у форматах без втрат, які використовують таке кодування і не виконують міжкомпонентну декореляцію.

8. Отримав подальший розвиток метод групового статистичного кодування за допомогою використання палітри в процесі стиснення трикомпонентних зображень без втрат, що дозволило зменшити КС на понад 1.4 %, хоча й сповільнило кодування на 38 %. Сумісна реалізація методу формування різницевих кольорових моделей з цілими коефіцієнтами, алгоритму коригування значень предиктора, алгоритму

LZPR, алгоритму мінімізації розміру стиснутих блоків та даного методу дає змогу зменшити КС стосовно швидких варіантів стиснення у форматі PNG на понад 12 % та досягає по цьому показнику найменших на сьогодні значень відносно інших споріднених програм, що створює передумови для їх успішного спільного застосування в графічних форматах та архіваторах.

Додаток А

Авторські свідоцтва на комп'ютерні програми, розроблені за результатами кандидатської дисертації (AnalysisForPNG та UsingDCM)

Копія авторського свідоцтва на комп'ютерну програму AnalysisForPNG міститься у першому примірнику дисертації, який знаходиться у НБУ ім. В. І. Вернацького

Копія авторського свідоцтва на комп'ютерну програму UsingDCM міститься у першому примірнику дисертації, який знаходиться у НБУ ім. В. І. Вернацького

Додаток Б

Акти впровадження наукових результатів кандидатської дисертації

(у Рівненській обласній клінічній лікарні, ТОВ "Інженерний центр "Імпульс", ВАТ "Алмаз СКС", ТзОВ фірмі "Віза")

Оригінал акта впровадження результатів кандидатської дисертації у Рівненській обласній клінічній лікарні міститься у першому примірнику дисертації, який знаходиться у НБУ ім. В. І. Вернацького

Оригінал акта впровадження результатів кандидатської дисертації у ТОВ "Інженерний центр "Імпульс" міститься у першому примірнику дисертації, який знаходиться у НБУ ім. В. І. Вернацького

Оригінал акта впровадження результатів кандидатської дисертації у ВАТ "Алмаз СОС" міститься у першому примірнику дисертації, який знаходиться у НБУ ім. В. І. Вернацького

Оригінал акта впровадження результатів кандидатської дисертації у ТзОВ фірмі "Віза" міститься у першому примірнику дисертації, який знаходиться у НБУ ім. В. І. Вернацького

Додаток В

Довідка про використання результатів кандидатської дисертації в РДГУ

Оригінал довідки про використання результатів кандидатської дисертації в РДГУ міститься у першому примірнику дисертації, який знаходиться у НБУ ім. В. І. Вернацького

Додаток Д

Застосування ARIC замість кодування HUFF для модифікацій формату PNG

Як відомо, арифметичне кодування ставить у відповідність кожному елементу інтервал з довжиною, пропорційною його ймовірності [37, с. 63]. При цьому з початкового інтервалу (найчастіше [0; 1)) обирають і надалі розглядають підінтервал, що відповідає першому елементу потоку (як, наприклад, на рис. Д.1). Цей підінтервал знову розбивають пропорційно частотам окремих елементів і з нього обирають підінтервал, що відповідає другому елементу потоку. Вибір підінтервалів триває аналогічно аж до закінчення елементів потоку. Таким чином, кінцева довжина обраного підінтервалу рівна добутку ймовірностей всіх елементів, а його початок залежить від порядку слідування цих елементів в потоці [24, с. 37]. Результатом ARIC є будь-яке число з останнього отриманого підінтервалу, яке, фактично, однозначно визначає всі елементи потоку. Зважаючи на скінченність розрядності чисел, для запису результуючого числа щоразу відслідковують перші значущі цифри меж інтервалів і записують їх у вихідний потік та виключають з подальшого розгляду, як тільки вони стають однакові.

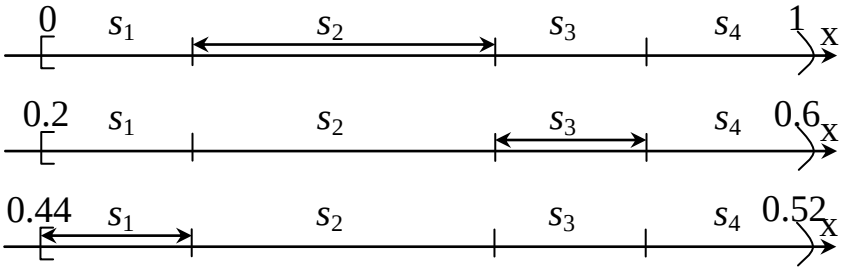


Рис. Д.1. Етапи арифметичного кодування трьох перших елементів послідовності літералів/довжин "2, 4, 1, 2, <3>"

Декодування наступного елемента, закодованого ARIC, виконують шляхом визначення відповідного чергового підінтервалу, якому належить зчитане число. Це кодування не потребує цілої кількості бітів для кодування кожного елемента (як при кодуванні HUFF), і тому середня довжина його коду прямує до ентропії [24, с. 17]. Середня ж довжина коду HUFF завжди перевищує ентропію і ця різниця пропорційна різниці між ймовірностями елементів, що поєднуються в процесі

генерації цих кодів. Доведемо це математично.

Твердження Д.1. Чим більше відрізняються між собою ймовірності елементів, що поєднуються під час виконання алгоритму HUFF, тим більшою стає різниця між середньою довжиною коду HUFF і ентропією джерела.

Доведення. Нехай внаслідок виконання алгоритму HUFF виконується поєднання двох елементів з частотами N та M. Не зменшуючи загальності, вважатимемо $N \geq M$. Алгоритм HUFF присвоює цим елементам коди довжиною 1, тобто загальна довжина коду для запису цих елементів складе $N+M$ бітів. Довжина ентропійного коду цих же елементів, згідно [24, с. 17], дорівнює $-N\log\left(\frac{N}{N+M}\right)-M\log\left(\frac{M}{N+M}\right)$. Обчислимо відхилення довжини коду HUFF від довжини ентропійного коду:

$$\begin{aligned}\Delta(N, M) &= N+M+N\log\left(\frac{N}{N+M}\right)+M\log\left(\frac{M}{N+M}\right)= \\ &= N\log\left(\frac{2N}{N+M}\right)+M\log\left(\frac{2M}{N+M}\right).\end{aligned} \qquad (Д.1)$$

Визначимо залежність цього відхилення від різниці частот N та M. Нехай $S=(N+M)/2$, $K=(N-M)/2$. Підставляючи ці змінні у (Д.1), отримаємо

$$\Delta(S, K) = (S+K)\log(S+K)+(S-K)\log(S-K)-2S\log(S). \qquad (Д.2)$$

$$\frac{\partial\Delta(S, K)}{\partial K} = \log\left(\frac{S+K}{S-K}\right) = \log\left(\frac{N}{M}\right) \geq 0. \qquad (Д.3)$$

Частинна похідна відхилення по різниці між частотами, згідно (Д.3), дорівнює нулю лише коли частоти елементів однакові. У всіх інших випадках вона перевищує нуль. Тобто різниця між довжиною коду HUFF і довжиною ентропійного коду не збільшується лише у випадку поєднань елементів з однаковими частотами (це також слідує з (Д.1) та (Д.2)). Як слідує з (Д.3), відхилення довжини коду HUFF від довжини ентропійного коду збільшується зі збільшенням різниці між поєднуваними частотами. Збільшення ж відхилень між загальними довжинами кодів свідчить про

збільшення відхилення між середньою довжиною коду HUFF і ентропією джерела.■

Отже, враховуючи той факт, що довжина коду HUFF завжди перевищує довжину ентропійного коду, дослідимо питання підвищення ефективності стиснення зображень у форматі PNG внаслідок заміни кодування HUFF арифметичним кодуванням.

Для виконання ARIC кодеру і, тим більше, декодеру мають бути відомі ймовірності окремих елементів. Сьогодні широко використовуються дві стратегії формування інтервалів елементів: статична та адаптивна. У випадку статичної стратегії частоти окремих елементів передаються декодеру у стиснутих даних явно [54]. При використанні адаптивної стратегії інтервали елементів формуються синхронно кодером та декодером в процесі опрацювання даних. Адаптивна стратегія забезпечує, як правило, кращі КС, оскільки не потребує зберігання у стиснутих даних частот окремих елементів, але суттєво сповільнює роботу декодера, бо вимагає перерахунку ймовірностей в процесі декодування. Формати ж графічних файлів мають забезпечувати, насамперед, швидке декодування, тому для модифікації формату PNG ми використали ARIC з статичною стратегією формування інтервалів.

Для **реалізації** ARIC ми модифікували та застосували range-кодер Е. Шелвіна [24, с. 361-363], оскільки він виконує зчитування/запис байтів а не бітів даних і за рахунок цього значно прискорює виконання алгоритму кодування/декодування [54]. Цей кодер для кожного елемента використовує інтервал з цілочисельною довжиною, пропорційною його ймовірності. З метою зберігання довжини інтервалу для кожного елемента у заголовку стиснутого блоку замість довжини коду HUFF нами записується **кількість бітів для зберігання відповідної довжини інтервалу** в загальному інтервалі [0; 32767), а після заголовка – **двійковий запис цієї довжини без першого біта** (який завжди рівний одиниці). Мовою С підпрограми цього модифікованого кодера, які використовуються для генерування та запису довжин інтервалів за частотами елементів записуються, наприклад, так:

```
// функція для визначення кількості бітів
// двійкового запису довжини інтервалу
int countBit(unsigned long diapazon)
```

```
 {if (diapazon==0) return 0;
  if (diapazon==1) return 1;
  if (diapazon<=3) return 2;
  if (diapazon<=7) return 3;
  ...
  if (diapazon<=16383) return 14;
  return 15; }
// масштабування сум частот до діапазону [0; 32767)
// для визначення довжин інтервалів елементів
// freq – масив частот та довжин інтервалів
// ehufsi – масив кількостей бітів двійкових записів довжин інтервалів
const uint TOP=1<<24; // нижня межа допустимої ширини інтервалу
const uint lenRange=1<<15; // сума масштабованих частот
sumFreq=0;
for (i=0; i<countFreq; ++i) sumFreq+=freq[i];
if (sumFreq==0) return; // частоти відсутні – інтервали не визначаємо
mnog=(double)lenRange/sumFreq;
maxFreq=0, indexMaxFreq=0; sumFreq=0;
if (mnog<1) // частоти потрібно зменшити
 {for (i=0; i<countFreq; ++i)
   if (freq[i]>0)
    {freq[i]*=mnog;
     if (!freq[i]) freq[i]=1;
     else
      if (freq[i]>maxFreq) // пошук максимальної частоти та її індекса
       {maxFreq=freq[i]; indexMaxFreq=i; }
     sumFreq+=freq[i]; }}
else // частоти потрібно збільшити
 {for (i=0; i<countFreq; ++i)
   if (freq[i]>0)
    {freq[i]*=mnog;
     if (freq[i]>maxFreq)
      {maxFreq=freq[i]; indexMaxFreq=i; }
     sumFreq+=freq[i]; }}
// враховуємо похибки заокруглень в максимальній частоті
freq[indexMaxFreq]+=lenRange-sumFreq
if (freq[indexMaxFreq]==lenRange)
 freq[indexMaxFreq]--; // для забезпечення можливості запису частоти
// підрахунок сум попередніх частот, кількостей бітів двійкового запису,
// масштабованих частот та генерація їх кодів без першого біта
sumFreq=0;
for (i=0 ; i<countFreq; ++i)
 if (freq[i]>0)
  {sumPprFreq[i]=sumFreq; // сума попередніх частот
   sumFreq+=freq[i];
   lenCode=countBit(freq[i]);
   ehufsi[i]=lenCode;
   for (j=0; j<lenCode-1; ++j)
    {if ((freq[i] & (1<<j)) != 0) // заносимо з двійкового запису лише одиниці
      bufFreq[pozBufFreq] |= (1 << pozBitFreq) ;
     ++pozBitFreq; // перехід до наступного біта
     if (pozBitFreq >= 8) // перехід до наступного байта
```

```
    {pozBufFreq++; bufFreq[pozBufFreq]=0; pozBitFreq=0; }}}
```

Після генерації інтервалів (масштабованих частот) елементів кодування чергового елемента зводиться до перенесення початку активного інтервалу і зменшення його ширини пропорційно відповідній частоті та масштабування ширини отриманого інтервалу до допустимих меж (функція *ShiftLow*() наведена у [24, с. 362]):

```
void ACEncoder::Encode (uint value)
 {low += sumPprFreq[value] * (range>>=15); // перенесення початку
  range*= freq[value]; // врахування частоти
  // масштабування ширини інтервалу
  while(range<TOP) ShiftLow(), range<<=8; }
```

Декодування ж чергового елемента полягає у визначенні індекса інтервалу, якому належить зчитаний код та у коригуванні параметрів цього інтервалу, як під час кодування:

```
int ACDecoder::Decode ()
 {int index = code / (range>>=15); // черговий зчитаний код
  int index1=0; // визначення індекса інтервалу
  while (sumPprFreq[index1]<=index) index1++;
  index=index1-1;
  code -= sumPprFreq[index]*range; // перенесення початку
  range *= freq[index]; // врахування частоти
  // масштабування ширини інтервалу
  while( range<TOP ) code=(code<<8) | buffer[pozBuffer++], range<<=8;
  return index; }
```

Для прискорення декодування шляхом уникнення циклу пошуку початку інтервала чергового елемента, після зчитування заголовка блоку стиснутих даних доцільно створити байтовий масив, у якому для кожного значення загального інтервалу зберегти номер елемента, що йому відповідає:

```
sumFreq=0;
for (i=0; i<countFreq; i++)
 if (huffbits[i]) // елемент в розподілі наявний
  { // зчитування частоти з доповненням першим бітом
   currentFreq=(1<<(huffbits[i]-1)) | decoder.GetBits(huffbits[i]-1);
   freq[i]=currentFreq;
   sumPprFreq[i]=sumFreq;
   // формування байтового масиву
   memset(inCode+sumFreq, i, currentFreq);
   sumFreq+=currentFreq; }
```

Тоді процедура декодування перепишеться у вигляді:

```
int ACDecoder::Decode ()
 {int index = code / (range>>=15); // черговий зчитаний код
  if (index>=sumPprFreq[256]) // оскільки обробляємо байти
```

```
 index=inCode[index]+256;
else index=inCode[index];
code -= sumPprFreq[index]*range;
range *= freq[index];
while( range<TOP ) code=(code<<8)|buffer[pozBuffer++], range<<=8;
return index; }
```

Застосування такого 32 Кб масиву дає змогу прискорити декодування в середньому на 46 %.

Розглянемо **результати застосування** описаного способу арифметичного кодування для компресії зображень набору ACT. Тестування проводили базовою програмою без внесення модифікацій, розглянутих у роботі, відмовляючись від застосування предикторів (*NonePredict*) та використовуючи предиктор Піфа (*PaethPredict*). Результати тестування наведено в табл. Д.1, Д.2, та Д.3. Як свідчать дані цих таблиць, використання ARIC для модифікації формату PNG дозволило покращити КС зображень набору ACT максимум на 0.38 %, а в середньому по набору – на 0.07 %. При цьому у випадку незастосування предикторів час кодування скоротився в середньому на 5.5 %, а декодування – на 20.5 %. Застосовуючи предиктори до тих самих зображень, час кодування внаслідок використання ARIC зменшився на 3.3 %; а декодування – на 10.1 %. Нижчі показники ефективності ARIC у випадку застосування предикторів пояснюються зменшенням кількості елементів, для яких виконується це кодування внаслідок збільшення кількостей однакових послідовностей, закодованих алгоритмом LZ77.

Таблиця Д.1

КС файлів зображень набору ACT після застосування різних предикторів та контекстно-незалежних алгоритмів, %

Предиктор	**Контекстно-незалежний алгоритм**	**№ файла**								**Середній КС**
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	
NonePredict	HUFF	23.89	6.85	91.29	70.42	84.53	77.54	7.17	83.09	55.60
	ARIC	23.51	6.85	91.16	70.42	84.53	77.54	7.17	83.09	55.53
PaethPredict	HUFF	23.80	9.77	67.75	55.68	58.65	67.65	9.97	61.14	44.30
	ARIC	23.80	9.80	67.62	55.68	58.52	67.39	9.97	61.06	44.23

Таблиця Д.2

Час кодування файлів зображень набору ACT у випадку застосування різних предикторів та контекстно-незалежних алгоритмів, с

Предиктор	Контекстно-незалежний алгоритм	№ файла								Середній час
		1	2	3	4	5	6	7	8	
NonePredict	HUFF	7.47	12.74	4.07	5.60	3.95	5.93	5.10	5.93	6.35
	ARIC	7.19	12.47	3.63	5.17	3.62	5.49	5.11	5.32	6.00
PaethPredict	HUFF	10.11	14.56	4.29	8.19	5.00	6.65	6.04	7.36	7.78
	ARIC	9.78	14.61	4.06	7.74	4.78	6.26	5.93	7.03	7.52

Таблиця Д.3

Час декодування файлів зображень набору ACT у випадку застосування різних предикторів та контекстно-незалежних алгоритмів, с

Предиктор	Контекстно-незалежний алгоритм	№ файла								Середній час
		1	2	3	4	5	6	7	8	
NonePredict	HUFF	3.47	4.18	2.69	3.24	2.52	3.46	1.65	3.68	3.11
	ARIC	2.85	3.90	1.92	2.47	1.76	2.63	1.54	2.69	2.47
PaethPredict	HUFF	4.17	5.82	2.47	3.35	2.31	3.51	2.41	3.52	3.45
	ARIC	3.79	5.60	2.20	2.86	1.97	3.08	2.30	2.97	3.10

Як показали результати подібного тестування, проведеного для програми, що реалізує описані у підрозділах 4.1 – 4.3 модифікації формату PNG, застосування ARIC замість HUFF у випадку використання алгоритму палітрування у середньому зменшує КС лише на 0.02 %, оскільки опрацьовує лише індекси паралелепіпедів незакодованих пікселів у палітрі.

Отже, альтернативою кодуванню HUFF у форматах графічних файлів є байт-орієнтоване арифметичне кодування зі статичною стратегією формування інтервалів елементів. Для здійснення такого кодування, враховуючи структуру ARIC, у стиснутих даних необхідно забезпечити відокремлене зберігання арифметичних кодів кожного розподілу. Для реалізації статичної стратегії формування інтервалів елементів ARIC у форматі PNG у заголовку кожного блоку стиснутих даних

доцільно замість довжин кодів HUFF зберігати кількості бітів для запису довжин інтервалів, а після заголовка – двійкові коди довжин без першого біта. Прискорити декодування арифметичних кодів зі статичним формуванням інтервалів в середньому на понад 45 % дозволяє використання допоміжного масиву, в якому для кожного цілочисельного значення загального інтервалу для всіх елементів зберігається номер елемента, який йому відповідає.

Додаток Е

Результати тестування програм на наборі зображень KTCI

Е.1. РЕЗУЛЬТАТИ ЗАСТОСУВАННЯ АЛГОРИТМУ ПОПЕРЕДНЬОГО АНАЛІЗУ ЗОБРАЖЕНЬ

Таблиця Е.1.1

КС файлів зображень набору KTCI у форматі PNG після застосування різних варіантів програм, %

Варіант програми	Заст. поперед. розкладу	№ файла							
		1	2	3	4	5	6	7	8
Базова	Ні	61.49	55.16	47.70	55.85	66.87	56.90	50.04	64.96
"Лінивий" розклад LZ77	Ні	58.46	50.82	42.50	52.56	65.05	52.30	47.01	64.01
	Так	58.46	50.82	42.50	52.47	64.96	52.21	46.92	63.92
Майже оптим. розклад LZ77	Ні	56.63	49.70	41.11	51.34	63.83	50.56	45.79	62.97
	Так	56.63	49.61	41.11	51.34	64.01	50.56	45.71	62.97
OptiPng, станд. перебір	Ні	61.23	52.39	45.88	54.03	65.65	53.86	47.96	64.35
OptiPng, макс. перебір	Ні	61.06	52.30	42.58	53.51	65.57	52.47	47.27	64.27

Продовж. табл. Е.1.1

Варіант програми	Заст. поперед. розкладу	№ файла							
		9	10	11	12	13	14	15	16
Базова	Ні	51.86	52.73	55.59	49.87	71.73	61.67	53.77	51.26
"Лінивий" розклад LZ77	Ні	46.57	48.83	50.65	44.41	68.69	58.20	49.96	45.01
	Так	46.49	48.66	50.56	44.41	68.69	58.20	49.87	44.93
Майже оптим. розклад LZ77	Ні	44.75	47.44	49.26	42.67	67.22	56.72	48.74	43.28
	Так	44.75	47.35	49.26	42.67	67.22	56.72	48.66	43.28
OptiPng, станд. перебір	Ні	49.44	50.30	52.82	48.22	70.25	60.62	51.95	47.09
OptiPng, макс. перебір	Ні	49.18	50.22	52.56	45.01	69.64	58.63	50.82	45.27

Продовж. табл. Е.1.1

Варіант програми	Заст. поперед. розкладу	№ файла								Середній КС
		17	18	19	20	21	22	23	24	
Базова	Ні	54.03	66.96	57.59	44.32	58.02	61.14	51.00	59.58	56.67
"Лінивий" розклад LZ77	Ні	50.30	64.44	54.21	41.37	52.73	58.46	46.66	57.42	52.94
	Так	50.22	64.44	54.12	41.28	52.73	58.37	46.57	57.42	52.88
Майже оптим. розклад LZ77	Ні	48.83	63.40	52.73	40.42	50.91	57.59	45.79	56.29	51.58
	Так	48.74	63.49	52.65	40.24	50.91	57.50	45.62	56.29	51.55
OptiPng, станд. перебір	Ні	52.47	65.05	55.94	42.41	56.03	59.50	47.27	58.80	54.73
OptiPng, макс. перебір	Ні	51.00	64.96	55.85	41.72	53.95	59.41	47.18	58.02	53.85

Таблиця Е.1.2

Час кодування файлів зображень набору KTCI у форматі PNG різними варіантами програм, с

Варіант програми	Заст. поперед. розкладу	№ файла							
		1	2	3	4	5	6	7	8
Базова	Ні	5.99	10.17	10.77	9.77	8.78	9.56	10.65	8.79
"Лінивий" розклад LZ77	Ні	31.52	36.25	39.05	36.03	32.35	33.50	38.33	32.57
	Так	36.97	43.88	46.67	43.17	38.45	39.88	46.79	37.90
Майже оптим. розклад LZ77	Ні	35.81	44.11	56.74	44.10	36.41	41.14	51.96	35.48
	Так	39.82	51.41	63.22	50.75	43.28	47.07	60.53	41.58
OptiPng, станд. перебір	Ні	23.89	32.68	44.05	28.12	21.48	32.07	41.20	23.62
OptiPng, макс. перебір	Ні	509.99	668.93	765.88	624.72	497.19	591.60	717.87	484.33

Продовж. табл. Е.1.2

Варіант програми	Заст. поперед. розкладу	№ файла							
		9	10	11	12	13	14	15	16
Базова	Ні	10.33	10.38	9.39	10.43	8.40	9.06	9.89	10.21
"Лінивий" розклад LZ77	Ні	36.86	36.69	32.85	35.54	30.42	31.20	36.69	34.82
	Так	42.79	45.04	39.11	42.02	35.32	36.31	43.94	40.65
Майже оптим. розклад LZ77	Ні	43.34	45.54	43.11	45.04	32.41	34.93	46.96	43.67
	Так	49.87	53.60	49.54	52.62	38.39	41.47	55.31	49.76
OptiPng, станд. перебір	Ні	34.60	34.99	36.09	37.29	19.93	22.95	34.00	31.52
OptiPng, макс. перебір	Ні	717.87	714.63	607.14	700.52	441.76	522.89	663.94	669.32

Продовж. табл. Е.1.2

Варіант програми	Заст. поперед. розкладу	№ файла								Середній час
		17	18	19	20	21	22	23	24	
Базова	Ні	10.10	8.29	9.83	9.28	9.88	9.00	10.71	9.39	9.54
"Лінивий" розклад LZ77	Ні	36.53	32.41	36.91	41.08	35.05	33.72	41.58	33.62	35.23
	Так	43.56	38.01	43.61	49.59	40.75	40.76	51.80	40.64	41.98
Майже оптим. розклад LZ77	Ні	44.82	35.38	44.05	57.56	40.64	38.72	53.66	41.69	43.22
	Так	52.24	41.58	50.36	66.51	46.36	45.87	63.22	47.97	50.09
OptiPng, станд. перебір	Ні	31.36	21.26	27.63	56.25	30.04	25.54	39.11	30.54	31.68
OptiPng, макс. перебір	Ні	679.32	499.55	648.07	841.41	645.76	574.14	787.42	584.29	631.61

Е.2. РЕЗУЛЬТАТИ СУКУПНОГО ВИКОРИСТАННЯ МОДИФІКАЦІЙ ФОРМАТУ PNG

Таблиця Е.2.1

КС файлів зображень набору KTCI після застосування різних програм, %

Програма	№ файла											
	1	2	3	4	5	6	7	8	9	10	11	12
Базова	61.49	55.16	47.70	55.85	66.87	56.90	50.04	64.96	51.86	52.73	55.59	49.87
Базова з модифікаціями	41.54	36.69	31.57	38.51	47.35	38.77	33.91	46.49	34.61	36.43	37.47	33.22
RAR v. 3.0	50.65	43.19	38.86	43.45	58.37	45.19	41.37	68.34	41.46	41.98	46.05	40.07
ERI v. 5.1	43.45	39.03	33.48	39.20	47.18	40.33	35.73	47.44	37.38	38.07	38.77	35.21

Продовж. табл. Е.2.1

Програма	№ файла												Середній КС
	13	14	15	16	17	18	19	20	21	22	23	24	
Базова	71.73	61.67	53.77	51.26	54.03	66.96	57.59	44.32	58.02	61.14	51.00	59.58	56.67
Базова з модифікаціями	49.52	43.10	37.21	33.39	36.69	47.35	39.81	31.92	39.29	43.63	36.08	43.37	39.08
RAR v. 3.0	67.91	49.35	43.71	40.42	43.80	54.21	48.66	35.99	47.09	48.57	41.02	51.26	47.12
ERI v. 5.1	50.22	43.54	37.55	36.17	38.16	47.01	41.28	34.69	41.54	44.06	36.43	42.41	40.35

Таблиця Е.2.2

Час кодування файлів зображень набору KTCI різними програмами, с

Програма	№ файла											
	1	2	3	4	5	6	7	8	9	10	11	12
Базова	5.99	10.17	10.77	9.77	8.78	9.56	10.65	8.79	10.33	10.38	9.39	10.43
Базова з модифікаціями	30.26	31.75	29.38	29.60	27.73	29.94	26.20	26.26	33.45	30.10	27.58	30.49
RAR v. 3.0	4.12	4.07	3.62	3.02	3.90	4.67	3.46	4.12	4.06	3.57	4.61	4.01
ERI v. 5.1	3.85	3.57	3.13	3.57	3.95	3.57	3.30	4.01	3.46	3.52	3.46	3.29

Продовж. табл. Е.2.2

Програма	№ файла												Середній час
	13	14	15	16	17	18	19	20	21	22	23	24	
Базова	8.40	9.06	9.89	10.21	10.10	8.29	9.83	9.28	9.88	9.00	10.71	9.39	9.54
Базова з модифікаціями	26.97	31.31	28.73	29.83	33.83	27.52	31.64	35.48	31.04	29.99	30.15	38.06	30.30
RAR v. 3.0	4.78	3.85	3.24	4.72	3.57	3.62	3.95	3.73	5.17	3.30	2.85	3.51	3.90
ERI v. 5.1	4.18	3.74	3.41	3.35	3.51	4.06	3.74	3.29	3.68	3.84	3.35	3.68	3.60

Таблиця Е.2.3

Час декодування файлів зображень набору KTCI різними програмами, с

Програма	№ файла											
	1	2	3	4	5	6	7	8	9	10	11	12
Базова	3.46	3.24	3.08	3.29	3.63	3.30	3.19	3.51	3.13	3.19	3.24	3.08
Базова з модифікаціями	3.90	3.46	3.24	3.46	3.63	3.57	3.30	3.63	3.35	3.46	3.35	3.30
RAR v. 3.0	1.10	0.60	0.61	0.66	0.60	0.65	0.60	0.60	0.61	0.66	0.87	0.66
ERI v. 5.1	4.28	3.95	3.57	4.12	4.45	4.06	3.73	4.45	4.01	4.01	3.95	3.79

Продовж. табл. Е.2.3

Програма	№ файла												Середній час
	13	14	15	16	17	18	19	20	21	22	23	24	
Базова	3.73	3.40	3.19	3.08	3.25	3.52	3.30	2.85	3.30	3.41	3.08	3.35	3.28
Базова з модифікаціями	3.62	3.51	3.41	3.29	3.41	3.63	3.47	2.20	3.41	3.57	3.35	2.52	3.38
RAR v. 3.0	0.55	0.66	0.66	0.66	0.66	0.71	0.66	0.50	0.66	0.66	0.66	0.60	0.66
ERI v. 5.1	4.67	4.23	3.84	3.84	4.01	4.50	4.23	3.73	4.18	4.28	3.79	4.12	4.07

Додаток Ж

Фрагменти реалізацій окремих алгоритмів мовою С

Ж.1. ФОРМУВАННЯ ТА ВИБІР НАЙКОРОТШОГО З АЛЬТЕРНАТИВНИХ СТИСНУТИХ БЛОКІВ

```
// крок 1 – розрахунок частот елементів альтернативних стиснутих
// блоків: розносимо частоти замін та відповідних їм літералів,
// що можуть виявитися неефективними
for (i=0; i<countAnalizZamina; i++)
 {len = lenZamina[i]; offset=offsetZamina[i];
  // визначаємо базові значення та додаткові біти
  LengthToCode (len, code, extra, value);
  DistanceToCode (offset, codeD, extraD, valueD) ;
  poz=pozImageZamina[i]; // початок заміни в даних
  nayavnoLenZamina[len]=true; // розподіл цієї довжини наявний
  // фіксуємо параметри заміни для аналізу альтернативних блоків:
  for (j=0; j<len; j++) // реєструємо літерали,
   freqProbaLength[len][imageData[poz+j]]++;
  freqProbaLength[len][code]++; // базу довжини,
  freqProbaDistance[len][codeD]++; // базу зміщення,
  plusBit[len]+=extra+extraD; } // додаткові біти заміни;
// крок 2 – вибір найкоротшого з альтернативних стиснутих блоків:
// встановлюємо ознаку наявності розподілу з усіма
nayavnoLenZamina[2]=true; // можливими замінами,
// встановлюємо індекс попереднього наявного альтернативного
pprNayavno=0; // стиснутого блоку,
// цикл по максимальних довжинах відкинутих замін
for (k=2; k<minLenZaminaLiteral; k++)
 if (nayavnoLenZamina[k]) // є заміни чергової довжини
  { // накопичуємо частоти літералів відкинутих замін
   for (i=0; i<=256; i++)
    {freqProbaLength[k][i]+=freqProbaLength[pprNayavno][i];
     // сумуємо частоти літералів розподілу з усіма
     // замінами та  частоти літералів відкинутих замін
     masAnalizLL[i]=freqCodeLengthTable[i]+freqProbaLength[k][i]; }
   // накопичуємо частоти базових значень довжин
   for (i=257; i<=285; i++) // відкинутих замін
    {freqProbaLength[k][i]+=freqProbaLength[pprNayavno][i];
     // обчислюємо різниці частот баз довжин розподілу
     // з усіма замінами та частот баз відкинутих замін
     masAnalizLL[i]=freqCodeLengthTable[i]-freqProbaLength[k][i]; }
   // підраховуємо довжину розподілу літералів/довжин замін
   // для чергового альтернативного стиснутого блоку
   lenCode=lenHuffmanCode(masAnalizLL, 286);
   // накопичуємо частоти базових значень зміщень відкинутих замін
   for (i=0; i<=29; i++)
    {freqProbaDistance[k][i]+=freqProbaDistance[pprNayavno][i];
```

```
      // обчислюємо різниці частот базових значень зміщень
      masAnalizD[i]=freqDistanceLengthTable[i]-freqProbaDistance[k][i]; }
    // додаємо довжину розподілу базових значень зміщень чергового
    // альтернативного стиснутого блоку
    lenCode+=lenHuffmanCode(masAnalizD, 30);
    // накопичуємо додаткові біти відкинутих замін
    plusBit[k]+=plusBit[pprNayavno];
    // відкидаємо додаткові біти з обчисленої довжини
    lenCode-=plusBit[k];
    if (lenCode<minLenCode) // черговий блок коротший
     {minLenCode=lenCode; // запам'ятовуємо цей розмір
      // запам'ятовуємо максимальну довжину
      // відкинутих замін найкоротшого з розглянутих
      indexMinBlock=k; } // альтернативних блоків
    // запам'ятовуємо індекс попереднього наявного
    pprNayavno=k; } // альтернативного блоку
// крок 3 – ітеративне зменшення розміру найкоротшого блоку
// визначення довжин кодів розподілу літералів/довжин
statEncoder.Reset();
for (i=0; i<=256; i++)
 statEncoder.frequencies[i]=freqCodeLengthTable[i]+
                                  freqProbaLength[lenTrebaZamina][i];
for (i=257; i<=285; i++)
 statEncoder.frequencies[i]=freqCodeLengthTable[i]-
                                   freqProbaLength[lenTrebaZamina][i];
statEncoder.BuildTable(PngMaxLengthCodeSize);
maxLenLength=statEncoder.ehufsi[256];
for (i=0; i<=285; i++)
 if (statEncoder.ehufsi[i]) lenProbaLength[i]=statEncoder.ehufsi[i];
 else lenProbaLength[i]=maxLenLength;
// визначення довжин кодів розподілу базових значень зміщень
statEncoder.Reset();
for (i=0; i<=29; i++)
 statEncoder.frequencies[i]=freqDistanceLengthTable[i]-
                                  freqProbaDistance[lenTrebaZamina][i];
statEncoder.frequencies[30]=1; // для генерування найбільшої довж. коду
statEncoder.BuildTable(PngMaxLengthCodeSize);
maxLenDistance=statEncoder.ehufsi[30];
if (maxLenDistance<3) maxLenDistance=3;
for (i=0; i<=29; i++)
 if (statEncoder.ehufsi[i]) lenProbaDistance[i]=statEncoder.ehufsi[i];
 else lenProbaDistance[i]=maxLenDistance;
// ітеративне відкидання неефективних замін
do
 {countZmineno=0;
  for (i=0; i<countAnalizZamina; i++)
   if (trebaZaminaBuffer[pozAnalizZamina[i]]) // заміна активна
    {len = block_buffer[pozAnalizZamina[i]] - 256;
     offset=block_buffer[pozAnalizZamina[i]+1];
     LengthToCode (len, code, extra, value);
     DistanceToCode (offset, codeD, extraD, valueD) ;
     poz=pozImageBuffer[pozAnalizZamina[i]];
```

```
     sizeZamina=lenProbaLength[code]+extra+
                      lenProbaDistance[codeD]+extraD;
     sizeLiteral=0;
     for (j=0; j<len && sizeLiteral<sizeZamina; j++)
      sizeLiteral+=lenProbaLength[imageData[poz+j]];
     if (sizeLiteral<sizeZamina)
      {// відміняємо заміну
       countZmineno++;
       trebaZaminaBuffer[pozAnalizZamina[i]]=0;
       for (j=0; j<len; j++)
        freqCodeLengthTable[imageData[poz+j]]++;
       freqCodeLengthTable[code]--;
       freqDistanceLengthTable[codeD]--; }}
  if (countZmineno) // якщо відкидалися заміни
   { // перераховуємо довжини кодів розподілів літералів/довжин
     // та зміщень, як на початку кроку 3
     ... }
  }while (countZmineno>5);
// ітеративне врахування ефективних замін
if (lenTrebaZamina>2)
 do
  {countZmineno=0;
   for (i=0; i<countAnalizZamina; i++)
     if (!trebaZaminaBuffer[pozAnalizZamina[i]]) // заміна неактивна
      {len = block_buffer[pozAnalizZamina[i]] - 256;
       offset=block_buffer[pozAnalizZamina[i]+1];
      LengthToCode (len, code, extra, value);
      DistanceToCode (offset, codeD, extraD, valueD) ;
      poz=pozImageBuffer[pozAnalizZamina[i]];
      sizeZamina=lenProbaLength[code]+extra+
                       lenProbaDistance[codeD]+extraD;
      sizeLiteral=0;
      for (j=0; j<len; j++)
       sizeLiteral+=lenProbaLength[imageData[poz+j]];
      if (sizeLiteral>sizeZamina)
       {// повертаємо заміну
        countZmineno++;
        trebaZaminaBuffer[pozAnalizZamina[i]]=1;
        for (j=0; j<len; j++)
         freqCodeLengthTable[imageData[poz+j]]--;
        freqCodeLengthTable[code]++;
        freqDistanceLengthTable[codeD]++; }}
   if (countZmineno) // якщо заміни враховувалися
    { // перераховуємо довжини кодів розподілів літералів/довжин
      // та зміщень, як на початку кроку 3
      ... }
  } while (countZmineno>5);
// зберігаємо частоти елементів розподілів
memcpy(length_table->frequencies, freqCodeLengthTable,
            PngMaxLengthCodes*sizeof(unsigned int));
memcpy(distance_table->frequencies, freqDistanceLengthTable,
            PngMaxDistanceCodes*sizeof(unsigned int));
```

Ж.2. ВИЗНАЧЕННЯ НОМЕРА ПРЕДИКТОРА, ЩО ПОРОДЖУЄ ДАНІ З НАЙМЕНШИМ КС ПІСЛЯ ІМІТАЦІЇ КОРОТКИХ ЗАМІН

```
for (i=0; i<=4; ++i) // цикл по предикторах
 {memset(freq, 0, sizeof(freq)); // частоти літералів/баз довжин
  memset(freqD, 0, sizeof(freqD)); // частоти баз зміщень
  memset(hash, 0, sizeof(hash)); // елементи хеш-таблиці
  plusBit=0; // додаткові біти зміщень
  j=0; // позиція обробки рядка після дії предиктора i
  while (j<row_width-2)
   if (hash[((buffers[i][j] & 15)<<8)+
             ((buffers[i][j+1] & 15)<<4)+(buffers[i][j+2] & 15)])
    {// подібні три байти вже були в рядку
     freq[257]++; // частота триелементної заміни
     offset=j-hash[((buffers[i][j] & 15)<<8)+
     ((buffers[i][j+1]&15)<<4)+(buffers[i][j+2]&15)]+1;
     // збільшуємо частоту бази зміщення
     freqD[codesDistance[offset]]++;
     // накопичуємо додаткові біти зміщення
     plusBit+=extrasDistance[codesDistance[offset]];
     // дані оброблених триелементних груп записуємо в хеш-таблицю
     hash[((buffers[i][j] & 15)<<8)+
      ((buffers[i][j+1] & 15)<<4)+(buffers[i][j+2] & 15)]=j+1;
     if (j+1<row_width-2)
      hash[((buffers[i][j+1] & 15)<<8)+
      ((buffers[i][j+2] & 15)<<4)+(buffers[i][j+3] & 15)]=j+2;
     if (j+2<row_width-2)
      hash[((buffers[i][j+2] & 15)<<8)+
      ((buffers[i][j+3] & 15)<<4)+(buffers[i][j+4] & 15)]=j+3;
     j+=3; }
   else // подібна група відсутня – заносимо літерал
    {freq[buffers[i][j]]++;
      hash[((buffers[i][j] & 15)<<8)+
       ((buffers[i][j+1] & 15)<<4)+(buffers[i][j+2] & 15)]=j+1;
      j++; }
   for (; j<row_width ; ++j) // останні позиції рядка
    freq[buffers[i][j]]++;
   len=lenEntropiCode(freq, 258)+
       lenEntropiCode(freqD, 30)+plusBit;
  if (len<minLen)
   {predict2=i; // запам'ятали номер предиктора
    minLen=len; }}
KC2=(double)minLen/(8*row_width)          .
```

Ж.3. РЕАЛІЗАЦІЯ ПОПІКСЕЛЬНОГО РОЗКЛАДУ LZ77 З ВИЗНАЧЕННЯМ ЕФЕКТИВНОСТІ СТИСНЕННЯ ОКРЕМИХ РЯДКІВ

```
// визначаємо ефективність стиснення рядків без дії предикторів
UBYTE2 minRowBlock=8192/(row_width+1)+1; // мін. к-ть рядків блоку
trebaPredictRow=new UBYTE1[image_row+1]; // для ознак ефективності
memset(trebaPredictRow, 0, sizeof(UBYTE1)*image_row);
// створюємо масив для кількостей елементів рядків довгих замін LZ77
UBYTE4 * countZaminaRow=new UBYTE4[image_row];
memset(countZaminaRow, 0, sizeof(UBYTE4)*image_row);
// переносимо дані зображення без зайвих бітів і перестановок
for (j=0; j<image_row; j++)
 memcpy(imageData+j*(row_width),(*current_image)[j], row_width);
// аналізуємо попіксельні однакові послідовності
currentPozImage=0;
countByteImage=row_width * image_row;
InitializeHashTable();
unsigned int length, offset, hashvalue;
while (currentPozImage < countByteImage)
 {LongestMatchPixel(length, offset); // шукаємо найдовшу однакову послідовність
  if (length<9) length=0; // враховуємо обмеження довжини
  if (length == 0)
   {hashvalue = HashValue (currentPozImage) ;
    FreePozHashEntry(currentPozImage);
    MoveHashEntry (currentPozImage, hashvalue) ;
    currentPozImage+=3; }
  else
   {if (offset!=row_width) // такі зміщення враховуються AbovePredict
     {countZaminaRow[currentPozImage/row_width]+=length;
      // враховуємо однакову послідовність і для словника попереднього рядка
      if ((int)((currentPozImage-offset)/row_width) <
          (int)(currentPozImage/row_width))
       countZaminaRow[(currentPozImage-offset)/row_width]+=length; }
    for (i = 0 ; i < length ; i+=3)
     { // поновлюємо хеш-ланцюги
      hashvalue = HashValue (currentPozImage) ;
      FreePozHashEntry(currentPozImage);
      MoveHashEntry (currentPozImage, hashvalue) ;
      currentPozImage+=3; }}}
// визначаємо рядки для стиснення без предикторів
UBYTE4 mega=row_width*0.75;
for (j=0; j<image_row; j++)
 if (countZaminaRow[j]<mega)
  trebaPredictRow[j]=1;
delete [] countZaminaRow; countZaminaRow=NULL;
// визначаємо рядки між рядками, що стискуються без предикторів
for (j=1; j<image_row-1; j++)
 if (trebaPredictRow[j-1]==trebaPredictRow[j+1] &&
     trebaPredictRow[j]!=trebaPredictRow[j-1] &&
     trebaPredictRow[j-1]==0)
  trebaPredictRow[j]=trebaPredictRow[j-1];
```

```
if (trebaPredictRow[1]==trebaPredictRow[2] &&
    trebaPredictRow[0]!=trebaPredictRow[1])
 trebaPredictRow[0]=trebaPredictRow[1];
if (trebaPredictRow[image_row-2]==trebaPredictRow[image_row-3] &&
    trebaPredictRow[image_row-1]!=trebaPredictRow[image_row-2])
 trebaPredictRow[image_row-1]=trebaPredictRow[image_row-2];
// ліквідуємо блоки рядків, коротші мінімуму
if (minRowBlock>2)
 {j=0; // початковий рядок зображення
  while (j<image_row) // поки не опрацьовані всі рядки
   {i=j+1; // індекс першого рядка наступного блоку
    while (i<image_row && trebaPredictRow[i]==trebaPredictRow[j]) i++;
    if (i<image_row && i-j<minRowBlock)
     {for (k=j; k<i; k++) // ліквідація короткого блоку
       trebaPredictRow[k]=trebaPredictRow[i];
      while (i<image_row && trebaPredictRow[i]==trebaPredictRow[j]) i++; }
    j=i; }}}
```

Ж.4. ОБЧИСЛЕННЯ ПРОГНОЗОВАНОЇ ДОВЖИНИ СТИСНУТОГО БЛОКУ У ФОРМАТІ DEFLATE З ВИКОНАННЯМ "ЖАДІБНОГО" РОЗКЛАДУ LZ77

```
int countBitPackBuffer(unsigned char * buffer, int lenBuffer, int startPoz)
 {int countBit=0,lenAnaliz=lenBuffer-2, lokalHashTablPredict[256];
  memcpy(lokalHashTablPredict,hashTablPredict,256*sizeof(int));
  unsigned int freqLen[286],freqOffset[30],baseKod,countBitKod,valueBitKod;
  memset(freqLen,0,286*sizeof(unsigned int));
  memset(freqOffset,0,30*sizeof(unsigned int));
  while (startPoz<lenAnaliz)
   {//шукаємо однакові послідовності
    int lenLZ=0, len, zmLZ, countSearch;
    if (lokalHashTablPredict[buffer[startPoz]]>=0) // є аналогічні початки
     {for (int zm=lokalHashTablPredict[buffer[startPoz]], countSearch=0, len=3;
           zm>=0 && countSearch<search_limit; zm=indexPredict[zm],
           countSearch++) // цикл по однакових початках
      {if ((buffer[zm+1]==buffer[startPoz+1]) &&
          (buffer[zm+2]==buffer[startPoz+2])) // наступні елементи однакові
       {if (buffer[zm+lenLZ]!=buffer[startPoz+lenLZ])
         continue;// збільшити довжину однакової послідовності з цієї позиції неможливо
        while (startPoz+len<lenBuffer) // визначення довжини однакової послідовності
         {if (buffer[zm+len]!=buffer[startPoz+len]) break;
          len++;
          if (len==258) break; }
        if (len>lenLZ)
         {lenLZ=len; zmLZ=startPoz-zm;
          if (lenLZ==258) break;
          if (startPoz+lenLZ==lenBuffer) break; }}}}
    if (lenLZ>0) // однакова послідовність знайдена
     {LengthToCode (lenLZ,baseKod,countBitKod,valueBitKod);
      countBit+=countBitKod;
      freqLen[baseKod]++;
      DistanceToCode (zmLZ,baseKod,countBitKod,valueBitKod);
      countBit+=countBitKod;
      freqOffset[baseKod]++; }
    else {freqLen[buffer[startPoz]]++; lenLZ=1; }
    corectHashPredict(lokalHashTablPredict, startPoz, lenLZ);
    startPoz+=lenLZ; }
  countBit+=lenEntropiCode(freqLen,286)+lenEntropiCode(freqOffset, 30);
  return countBit; }
```

Ж.5. РОЗБИТТЯ ЗОБРАЖЕННЯ НА БЛОКИ РЯДКІВ

```
// обчислення довжини коду поєднання блоків
UBYTE4 lenSumisnHuffmanCode(freqCodeLL masFreq1,
            freqCodeLL masFreq2, unsigned int countAllFreq=256)
 {freqCodeLL mas;
  for (UBYTE4 i=0; i<countAllFreq; i++)
   mas[i]=masFreq1[i]+masFreq2[i]; // обчислення частот поєднання блоків
  return lenHuffmanCode(mas, countAllFreq); }

// визначення частот після застосування предиктора з мін. ентропією
for (k=0; k<=4; k++) // цикл по предикторах
 {memset(freq, 0, sizeof(freq));
  for (unsigned int jj = 0 ; jj < row_width ; ++ jj)
   freq[predict_buffers[k][jj]]++; // накопичення частот k-го предиктора
  for (unsigned int jj=0, run=0; jj<256; jj++)
   if (freq[jj]>1)
    run+=freq[jj]*log(freq[jj]);
  run=(row_width*log(row_width)-run)/(log(2)*8*row_width);
  if (run<longestrun) // знайдено предиктор з меншою ентропією
   {longestrun=run;
    memcpy(freqMinBlock[row], freq, 256*sizeof(UBYTE4)); }}
  countRowMinBlock[row]=1;

// розбиття зображення на блоки рядків
countBlock=image_row;
// визначення приростів від поєднань суміжних блоків
for (j=0; j<countBlock-1; j++)
 pruristSumisn[j]=lenSumisnHuffmanCode(freqMinBlock[j], freqMinBlock[j+1])-
  sizeHuffmanCode(freqMinBlock[j])-sizeHuffmanCode(freqMinBlock[j+1]);
// ітеративне поєднання блоків
do
 {minPruristLen=1500;
  indexMPL=-1;
  for (j=0; j<countBlock-1; j++)
   if (pruristSumisn[j]<minPruristLen)
    {minPruristLen=pruristSumisn[j];
     indexMPL=j; }
  if (indexMPL>=0) // можливе поєднання
   { // поєднання обраних та зсув параметрів наступних блоків
    for (i=0; i<256; i++) // поєднання частот блоків
     freqMinBlock[indexMPL][i]+=freqMinBlock[indexMPL+1][i];
    memcpy(freqMinBlock[indexMPL+1], freqMinBlock[indexMPL+2],
              (countBlock-indexMPL-2)*sizeof(freqCodeLL));
    countRowMinBlock[indexMPL]+=countRowMinBlock[indexMPL+1];
    memcpy(countRowMinBlock+indexMPL+1, countRowMinBlock+
              indexMPL+2, (countBlock-indexMPL-1)*sizeof(UBYTE2));
    // розрахунок приростів від поєднань з утвореним блоком
    if (indexMPL>0) // попередній блок існує
     pruristSumisn[indexMPL-1]=
      lenSumisnHuffmanCode(freqMinBlock[indexMPL-1],
```

```
                                    freqMinBlock[indexMPL])-
      lenHuffmanCode(freqMinBlock[indexMPL-1])-
      lenHuffmanCode(freqMinBlock[indexMPL]);
    if (indexMPL<countBlock-2) // наступний блок існує
     {pruristSumisn[indexMPL]=
      lenSumisnHuffmanCode(freqMinBlock[indexMPL],
                                     freqMinBlock[indexMPL+1])-
      lenHuffmanCode(freqMinBlock[indexMPL])-
      lenHuffmanCode(freqMinBlock[indexMPL+1]);
      memcpy(pruristSumisn+indexMPL+1, pruristSumisn+indexMPL+2,
                (countBlock-indexMPL-2)*sizeof(UBYTE4)); }
    countBlockPredict--; }
  }while (indexMPL>=0); // поки доцільні поєднання
```

Ж.6. ВИЗНАЧЕННЯ ОПТИМАЛЬНИХ ВАРІАНТІВ КОМПРЕСІЇ ДЛЯ МІНІМАЛЬНИХ БЛОКІВ РЯДКІВ

```
// визначення розміру даних зображення після застосування предикторів
countByteImage=image_row*(row_width+1);
for (l=4; l>=0; l--) // цикл по варіантах компресії зображення
 {// формування даних після застосування чергового варіанта компресії
  switch (l)
   {case 0: // без застосування предикторів (NonePredict)
     for (j=0; j<image_row; j++) // цикл по рядках
      {imageData[j*(row_width+1)]=0; // ознака NonePredict перед рядком
       memcpy(imageData+j*(row_width+1)+1,(*image)[j], row_width); } break;
    case 1: // після застосування LeftPredict
     for (j=0; j<image_row; j++)
      {imageData[j*(row_width+1)]=1;
       memcpy(imageData+j*(row_width+1)+1,(*image)[j], 3);
       for (i=3; i<row_width; i++)
        imageData[j*(row_width+1)+i+1]=(*image)[j][i]-(*image)[j][i-3]; } break;
    case 2: // після застосування RightPredict
     imageData[0]=2;
     memcpy(imageData+1,(*image)[0], row_width);
     for (j=1; j<image_row; j++)
      {imageData[j*(row_width+1)]=2;
       for (i=0; i<row_width; i++)
        imageData[j*(row_width+1)+i+1]=(*image)[j][i]-(*image)[j-1][i]; } break;
    case 3: // після застосування безпосереднього ентропійного способу
     for (j=0; j<image_row; j++)
      {imageData[j*(row_width+1)]=nomerFilterRow[j][0];
       FormFilterRow(j, nomerFilterRow[j][0]);
       memcpy(imageData+j*(row_width+1)+1,filter_buffers[cfilter],row_width); }
     break;
    case 4: // після застосування ентропійного способу з короткими
     for (j=0; j<image_row; j++) // замінами алгоритму LZ77
      {imageData[j*(row_width+1)]=nomerFilterRow[j][1];
       FormFilterRow(j, nomerFilterRow[j][1]);
       memcpy(imageData+j*(row_width+1)+1,filter_buffers[cfilter],row_width); }
     break; }
  // перестановка компонентів згідно формату PNG
  for (j=0; j<image_row; j++)
   for (i=0; i<image_col;i++)
    {k=imageData[j*(row_width+1)+1+3*i];
     imageData[j*(row_width+1)+1+3*i]=imageData[j*(row_width+1)+1+3*i+2];
     imageData[j*(row_width+1)+1+3*i+2]=k; }
  // розрахунок прогнозованої довжини коду в межах мінімальних блоків
  currentPozImage=0; InitializeHashTable();
  for (j=0; j<countBlockFilter; j++) // цикл по мінімальних блоках
   {mega=nextPozStartBlockFilter[j]; // позиція початку наступного блоку
    memset(freqCodeLengthTable, 0, sizeof(freqCodeLengthTable));
    memset(freqDistanceLengthTable, 0, sizeof(freqDistanceLengthTable));
    plusBit=0; // додаткові біти коду блоку
    memset(nayavnoLenZamina, 0, sizeof(nayavnoLenZamina));
```

```
    memset(freqProbaLength, 0, sizeof(freqProbaLength));
    memset(freqProbaDistance, 0, sizeof(freqProbaDistance));
    memset(plusProbaBit, 0, sizeof(plusProbaBit));
    while (currentPozImage < mega) // цикл до даних чергового блоку
     {LongestMatch (length, offset) ;
      if (length == 0) // однакова послідовність відсутня
       {hashvalue = HashValue (currentPozImage) ;
        FreePozHashEntry(currentPozImage);
        MoveHashEntry (currentPozImage, hashvalue) ;
        freqCodeLengthTable[imageData[currentPozImage]]++;
        currentPozImage++; }
      else
       {LengthToCode (length, codeL, extraL, value);
        DistanceToCode (offset, codeD, extraD, value);
        freqCodeLengthTable[codeL]++;
        freqDistanceLengthTable[codeD]++;
        plusBit+=extraL+extraD;
        if (length<minLenZaminaLiteral)
         { // фіксування параметрів можливої неефективної заміни
          nayavnoLenZamina[length]=true; // заміни такої довжини наявні
          for (ii=0; ii<length; ii++)
           freqProbaLength[length][imageData[currentPozImage+ii]]++;
          freqProbaLength[length][codeL]++;
          freqProbaDistance[length][codeD]++;
          plusProbaBit[length]+=extraL+extraD; }
        for (ii = 0 ; ii < length ; ii++)
         {FreePozHashEntry(currentPozImage);
          hashvalue = HashValue (currentPozImage) ;
          MoveHashEntry (currentPozImage, hashvalue) ;
          currentPozImage++; }}}
        ...
        // визначення прогнозованої мінімальної довжини коду блоку,
        // використовуючи програму з Ж.1 без аналізу ефективності замін
        …
        bitBlockImage[j][l]=minSizeCode+plusBit; }}
const UBYTE2 plusBitPerexid[5][5]= // масив штрафів за зміну
 {{   0, 2000, 2000, 1800, 1800},       // варіанту компресії між
  {7000,    0, 1500, 1300, 1300},       // суміжними блоками
  {7000, 1500,    0, 1300, 1300},
  {7000, 1500, 1500,    0, 1300},
  {7000, 1500, 1500, 1300,    0}};
// прямий хід методу динамічного програмування  (3.12) для мінімізації
// загального прогнозованого розміру всього зображення
for (j=1; j<countBlockFilter; j++) // цикл по блоках
 for (i=0; i<=4; i++) // цикл по варіантах компресії чергового блоку
  {indexMinBit=0; minBit=bitBlockImage[j-1][0]+plusBitPerexid[0][i];
   for (m=1; m<=4; m++)//цикл по варіантах компресії попереднього блоку
    if (bitBlockImage[j-1][m]+plusBitPerexid[m][i]<minBit)
     {indexMinBit=m; minBit=bitBlockImage[j-1][m]+plusBitPerexid[m][i]; }
   // запам'ятовуємо індекс найвигіднішого варіанту компресії
   // попереднього блоку стосовно активного варіанту компресії
   // чергового блоку і обчислюємо загальну кількість бітів
```

```
   pprIndexBlockImage[j][i]=indexMinBit; bitBlockImage[j][i]+=minBit; }
// визначення прогнозованого мінімального розміру зображення (3.13)
indexMinBit=0; // та оптимального варіанту компресії останнього блоку (3.14)
minBit=bitBlockImage[countBlockFilter-1][0];
for (m=1; m<=4; m++) // цикл по інших варіантах компресії
 if (bitBlockImage[countBlockFilter-1][m]<minBit)
  {indexMinBit=m; minBit=bitBlockImage[countBlockFilter-1][m]; }
rozpodilBlockFilter[countBlockFilter-1]=indexMinBit;
// зворотній хід методу динамічного програмування (3.15)
// для визначення оптимальних варіантів компресії інших блоків
for (j=countBlockFilter-2; j>=0; j--) //цикл по блоках
 rozpodilBlockFilter[j]=pprIndexBlockImage[j+1][rozpodilBlockFilter[j+1]];
// встановлення номерів предикторів та частот розподілів
// згідно обраних варіантів компресії мінімальних блоків
l=0; // індекс першого рядка наступного блоку
for (j=0; j<countBlockFilter; j++)
 {k=l; // індекс першого рядка чергового блоку
  l=nextPozStartBlockFilter[j]/(row_width+1);
  n=rozpodilBlockFilter[j]; elementBlockFilter[j]=elementBlockImage[j][n];
  memcpy(freqLLBlockFilter[j],freqLLBlockImage[j][n], sizeof(freqCodeLL));
  memcpy(freqDBlockFilter[j],freqDBlockImage[j][n], sizeof(freqCodeD));
  if (n<3) // перевага єдиного фільтра над ентропійними
   for (i=k; i<l; i++) filterRow[i]=n;
  else
   for (i=k; i<l; i++) filterRow[i]=nomerFilterRow[i][n-3]; }
```

Ж.7. ФОРМУВАННЯ, ПЕРЕХІД ТА ПОВЕРНЕННЯ ВІД РІЗНИЦЕВОЇ КОЛЬОРОВОЇ МОДЕЛІ З ЦІЛИМИ КОЕФІЦІЄНТАМИ

```
// фрагмент кодера для формування та переходу до
// різницевої кольорової моделі з цілими коефіцієнтами

// розрахунок результатів дії LeftPredict
for (i=0; i<height; i++) // цикл по рядках зображення
 {for (j=0; i<3; j++)
   corect[i][j]=image[i][j]; // перенесення даних першого піксела рядка
  for (j=3; j<row_width; j++) // цикл по компонентах інших пікселів рядка
   corect[i][j]=image[i][j]-image[i][j-3]; }

// обчислення значення масиву A прогнозованих довжин кодів
for (m=0; m<=2; m++) // цикл по рядках масиву A
 for (n=0; n<=2; n++) // цикл по стовпцях масиву A
  {memset(freq, 0, sizeof(freq)); // обнулення масиву частот елементів
   // елемент діагональний – оцінюємо довжину коду компоненти
   if (m==n) // після дії предиктора
    {for (i=0; i<height; i++) // цикл по рядках масиву значень предиктора
      for (j=0; j<row_width; j+=3) // цикл по пікселах чергового рядка
       freq[corect[i][j+m]]++;
     a[m][n]=lenEntropiCode(freq); }
   else // якщо елемент недіагональний
    {if (n>m) // вище діагоналі – оцінюємо довжину коду різниці компонентів
      {for (i=0; i<height; i++) // після дії предиктора
        for (j=0; j<row_width; j+=3)
         freq[(byte)(corect[i][j+m]-corect[i][j+n])]++;
       a[m][n]=lenEntropiCode(freq); }
      else // нижче діагоналі – встановлюємо значення
       a[m][n]=a[n][m]; }} // симетричного елемента

// визначення елементів масиву A для формування різницевої кольорової моделі
index11=0, index12=0, index21=0, index22=0;
minusLen=0; // прогнозоване зменшення довжини ентропійного коду
for (c11=0; c11<=2; c11++)
 for (c12=0; c12<=2; c12++)
  if (c11!=c12 && a[c11][c12]<a[c11][c11]) // якщо після дії предиктора
   // довжина ентропійного коду різниці компонентів менше
   // довжини ентропійного коду компоненти
   {if (a[c11][c11]-a[c11][c12]>minusLen) // і це зменшення максимальне
    {minusLen=a[c11][c11]-a[c11][c12]; // то запам'ятовуємо цю позицію
     index11=c11; index12=c12; index21=0; index22=0; }
    for (c21=c11+1; c21<=2; c21++) // шукаємо додаткове зменшення
     for (c22=0; c22<=2; c22++)
      if (c21!=c22 && (c21!=c12 || c22!=c11) && a[c21][c22]<a[c21][c21])
       // довжина ентропійного коду іншої різниці компонентів менше
       // довжини ентропійного коду компоненти
       if (a[c11][c11]-a[c11][c12]+
           a[c21][c21]-a[c21][c22]>minusLen) // і це збільшення максимальне
        {minusLen=a[c11][c11]-a[c11][c12] + a[c21][c21]-a[c21][c22];
```

```
        index11=c11; index12=c12; index21=c21; index22=c22; }}
if (index11==index22) // потрібно змінити порядок віднімань
 {i=index11; index11=index21; index21=i;
  i=index12; index12=index22; index22=i; }

// застосування сформованої різницевої кольорової моделі
if (index11!=index12) // перша різниця визначена
 for (i=0; i<height; i++)
  for (j=0; j<row_width; j+=3)
   image[i][j+index11]-=image[i][j+index12];
if (index21!=index22) // друга різниця визначена
 for (i=0; i<image_row; i++)
  for (j=0; j<row_width; j+=3)
   image[i][j+index21]-=image[i][j+index22];

// фрагмент декодера для повернення від
// різницевої кольорової моделі з цілими коефіцієнтами

if (index21!=index22) // друга різниця визначена
 {for (i = 0; i < image_row; i++)
   for (j = 0; j<row_width; j+=3)
    image[i][j+index21]+=image[i][j+index22];
  index21=index22=0; } // для уникнення повторного повернення
if (index11!=index12) // перша різниця визначена
 {for (i = 0; i < image_row; i++)
   for (j = 0; j<row_width; j+=3)
    image[i][j+index11]+=image[i][j+index12];
  index11=index12=0; } // для уникнення повторного повернення
```

Ж.8. ВИЗНАЧЕННЯ МАКСИМАЛЬНОГО ЗМЕНШЕННЯ ДОВЖИНИ ГРУПОВОГО КОДУ ВНАСЛІДОК ПОДІЛУ ЧЕРГОВОГО ПАРАЛЕЛЕПІПЕДА ПІД ЧАС ФОРМУВАННЯ ПАЛІТРИ

```
if (maxR-minR==0 && maxG-minG==0 && maxB-minB==0)
 return 0; // паралелепіпед вже має мінімальний розмір
// розраховуємо кількість пікселів паралелепіпеда по значеннях осей
// (на перетині з площинами)
memset(countR, 0, 256*sizeof(UBYTE4));
memset(countG, 0, 256*sizeof(UBYTE4));
memset(countB, 0, 256*sizeof(UBYTE4));
p=firstPoint; // номер першого піксела паралелепіпеда
for (index=0; index<countPoint; index++)
 {countR[imageData[3*p]-minR]++;
  countG[imageData[3*p+1]-minG]++;
  countB[imageData[3*p+2]-minB]++;
  p=nextPoint[p]; }
// визначення максимального зменшення довжини групового коду внаслідок розбиття
// паралелепіпеда площинами, перпендикулярними осі R,
// які проходять через його цілочисельні координати
if (maxR>minR)
 {countLeft=0; countRight=countPoint;
  p=countPoint*(countBit(maxR-minR+1)-log2(countPoint));
  // цикл по правій межі лівого паралелепіпеда
  for (index=0; index<maxR-minR; index++)
   if (countR[index]>0)
   {countLeft+=countR[index]; countRight-=countR[index];
    index1=index+1;
    while (countR[index1]==0)
     index1++; // пошук лівої межі правого паралелепіпеда
    plus=p-(countLeft*(countBit(index+1)-log2(countLeft))+
          countRight*(countBit(maxR-minR-index1+1)-log2(countRight)));
    if (plus>0 && plusBit<plus)
     {plusBit=plus; tupMega=1; mega=minR+index; }}}
// опрацювання значень паралелепіпеда по осях G та B виконується аналогічно
...
// визначаємо площину, яка забезпечує максимальне зменшення довжини групового коду
switch(tupMega)
 {case 1:plusMaxR=mega; break;
  case 2:plusMaxG=mega; break;
  case 3:plusMaxB=mega; break; }
return plusBit; }
```

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. А. с. 35643 України. Комп'ютерна програма "AnalisysForPNG" / О. В. Шпортько – № 35819; заявл. 13.09.2010; опубл. 11.11.2010.
2. А. с. 35644 України. Комп'ютерна програма "UsingDCM" / О. В. Шпортько – № 35820; заявл. 13.09.2010; опубл. 11.11.2010.
3. Баврина А. Ю. Метод иерархической компрессии индексных изображений / А. Ю. Баврина, Н. И. Глумов, В. В. Сергеев, Е. И. Тимбай // Компьютерная оптика. – 2004. – № 26. – С. 125-129.
4. Бомба А. Я. Алгоритм оптимізації вибору фільтра для попереднього опрацювання зображень перед стисненням на основі методу "предиктор – коректор" / А. Я. Бомба, О. В. Шпортько // Вісник Національного університету "Львівська політехніка". – 2008. – № 621. – С. 46-54. – (Серія: Інформаційні системи та мережі).
5. Бомба А. Я. Ентропійні способи вибору предиктора для рядка пікселів у форматі PNG / А. Я. Бомба, О. В. Шпортько // Управляющие системы и машины. – 2010. – № 3. – С. 8-25.
6. Бомба А. Я. Комплексне застосування модифікацій формату PNG / А. Я. Бомба, О. В. Шпортько // Обчислювальні методи і системи перетворення інформації : Зб. праць наук. техн. конф. – Львів: ФМІ НАН України, 2010. – С. 151-154.
7. Бредихин Д. Ю. Сжатие графики без потерь качества [Електронний ресурс] / Д. Ю. Бредихин. – 2004. – Режим доступу: http://www.compression.ru/download/articles/i_lless/bredikhin_2004_lossless_image_compression_doc.rar. – Назва з екрана.
8. Гонсалес Р. Цифровая обработка изображений / Р. Гонсалес, Р. Вудс. – М.: Техносфера, 2005. – 1072 с.
9. Воробель Р. А. Ядра визначення контрасту елементів зображення / Р. А. Воробель // Відбір і обробка інформації. – 1997. – Вип. 11 (87). – С. 96-100.
10. Воробель Р. А. Визначення аналітичних функцій контрасту на основі трикутної норми Гамахера / Р. А. Воробель // Відбір і обробка інформації. – 2008. – Вип.

28 (104). – С. 103-109.

11. Воробель Р. А. Глобальні перетворення зображень з використанням контрасту їх елементів / Р. А. Воробель // Оброблення сигналів і зображень та розпізнання образів : Матеріали дев'ятої Всеукраїнської міжнародної конференції УкрОБРАЗ'2008. – К.: МННЦ ІТтС, 2008. – С. 155-158.
12. Дядик Д. Ф. Метод стиску зображень без втрат на основі контекстного моделювання в системах телекомунікацій : автореф. дис. на здобуття наук. ступеня канд. техн. наук : спец. 05.12.13 "Радіотехнічні пристрої та засоби телекомунікацій" / Д. Ф. Дядик ; Нац. аерокосмічний ун-т ім. М. Є. Жуковського "Харківський авіаційний інститут". – Х., 2008. – 22 с.
13. Іванов В. Г. Моделі, методи й інформаційні технології агрегативного кодування і стиску мультимедійних даних : автореф. дис. на здобуття наук. ступеня доктора техн. наук : спец. 05.13.06 "Інформаційні технології" / В. Г. Іванов ; Нац. технічний ун-т "Харківський політехнічний інститут". – Х., 2008. – 20 с.
14. Кадач А. В. Эффективные алгоритмы неискажающего сжатия текстовой информации / А. В. Кадач. – Дис. ... канд. физ.-мат. наук. – Ин-т систем информатики им. А.П. Ершова. – Новосибирск, 1997. – 200 с.
15. Кнут Д. Е. Искусство программирования для ЭВМ. Т. 3: Сортировка и поиск / Д. Е. Кнут. – 2-е изд. – М.: Вильямс, 2008. – 824 с.
16. Лидовский В. В. Теория информации : Учеб. пособ. / В. В. Лидовский. – М.: Компания Спутник+, 2004. – 111 с.
17. Мастрюков Д. Алгоритмы сжатия информации. Часть 1. Сжатие по Хаффмену / Д. Мастрюков // Монитор. – 1993. – № 7-8. – С. 14-20.
18. Мастрюков Д. Алгоритмы сжатия информации. Часть 2. Арифметическое кодирование / Д. Мастрюков // Монитор. – 1994. – № 1. – С. 20-23.
19. Мастрюков Д. Алгоритмы сжатия информации. Часть 3. Алгоритмы группы LZ / Д. Мастрюков // Монитор. – 1994. – № 2. – С. 10-13.
20. Мастрюков Д. Алгоритмы сжатия информации. Часть 4. Алгоритм LZW / Д. Мастрюков // Монитор. – 1994. – № 3. – С. 8-11.
21. Мастрюков Д. Алгоритмы сжатия информации. Часть 5. Алгоритмы сжатия в

драйверах устройств / Д. Мастрюков // Монитор. – 1994. – № 4. – С. 24-27.

22. Мастрюков Д. Алгоритмы сжатия информации. Часть 7. Сжатие графической информации / Д. Мастрюков // Монитор. – 1994. – № 6. – С. 12-17.
23. Методы компьютерной обработки изображений / [М. В. Гашников, Н. И. Глумов, Н. Ю. Ильясов и др.] ; Под ред. В. А. Сойфера. – 2-е изд., испр. – М.: ФИЗМАТЛИТ, 2003. – 784 с.
24. Методы сжатия данных. Устройство архиваторов, сжатие изображений и видео / Д. Ватолин, А. Ратушняк, М. Смирнов, В. Юкин. – М.: ДИАЛОГ-МИФИ, 2003. – 384 с.
25. Миано Дж. Форматы и алгоритмы сжатия изображений в действии : Учеб. пособ. / Дж. Миано. – М.: Триумф, 2003. – 336 с., ил. – (Серія: Практика программирования).
26. Минченков В. О. Цветовое преобразование для сжатия компьютерных и синтетических изображений без потерь / В. О. Минченком, А. В. Сергеев, А. М. Тюрликов // Информационно-управляющие системы. – 2008. – № 5. – С. 26-32.
27. Михальчич Е. В. Описание формата сжатия данных DEFLATE [Електронний ресурс] / Е. В. Михальчич. – 2004. – Режим доступу: http://evm.narod.ru/deflate.html. – Назва з екрана.
28. Мюррей Д. Энциклопедия форматов графических файлов: пер. с англ. / Д. Мюррей, Ван У. Райпер. – К.: BHV, 1997. – 672 с.
29. Пономаренко С. И. Пиксел и вектор: Принципы цифровой графики. Глава 17. Цифровые модели. Цифровая модель RGB [Електронний ресурс] / С. И. Пономаренко. – 2002. – Режим доступу: http://www.computerbooks.ru/books/Graphics/Book-Digital-Graphics/Glava%2017/Index2.htm. – Назва з екрана.
30. Привалов М. Выбор рационального набора признаков текстуры для сегментации ультразвуковых эхограмм [Електронний ресурс] / М. Привалов, А. Корсакова. – 2002. – Режим доступу: http://www.uran.donetsk.ua/~masters/2007/kita/gatilova/library/strykov/strykovo.htm. – Назва з екрана.
31. Прэтт Э. Цифровая обработка изображений: Пер. с англ. / Э. Прэтт. – М.: Мир,

1982. – Кн. 1, 312 с., ил.

32. Прэтт Э. Цифровая обработка изображений: Пер. с англ. / Э. Прэтт. – М.: Мир, 1982. – Кн. 2, 480 с., ил.
33. Ратушняк О. А. Алгоритмы сжатия изображений без потерь с помощью сортировки параллельных блоков / О. А. Ратушняк // Тезисы докладов конференции молодых ученых по математике, мат. моделированию и информатике. – Новосибирск, 2001. – С. 48-49.
34. Ратушняк О. А. Методы сжатия данных без потерь с помощью сортировки параллельных блоков / О. А. Ратушняк. – Дис. ... канд. физ.-мат. наук. – Красноярский государственный технический университет. – Красноярск, 2002. – 87 с.
35. Рыбаков Г. Дискретная математика: алгоритмы [Електронний ресурс] / Г. Рыбаков, А. Суслов. – 2006. – Режим доступу: http://www.rain.ifmo.ru/cat/view.php/theory/data-compression/jpeg-2006. – Назва з екрана.
36. Седов С. А. Индивидуальные видеосредства: телеантенны, телевизоры, видеомагнитофоны, видеокамеры, видеопроигрыватели, видеодиски : Справоч. пособие / С. А. Седов. – К.: Наукова думка, 1990. – 752 с.
37. Сэломон Д. Сжатие данных, изображений и звука / Д. Сэломон. – М.: Техносфера, 2006. – 368 с. – (Серія: Мир программирования: цифровая обработка сигналов).
38. Умняшкин С. В. Оптимизация кодирования цифровых изображений по методу JPEG / С. В. Умняшкин, М. В. Космач // Известия вузов. Электроника. – 2000. – № 4-5. – С. 139-141.
39. Фомин А. А. Основы сжатия информации : Метод. пос. / А. А. Фомин. – СПб: Изд. СПбГТУ, 1998. – 83 с.
40. Хэмминг Р. В. Теория кодирования и теория информации / Р. В. Хэмминг. – М.: Радио и связь, 1985. – 176 с.
41. Цифровая обработка изображений в информационных системах : Учеб. пос. / И. С. Грузман, В. С. Киричук, В. П. Косых и др. – Новосибирск: НГТУ, 2000. – 168 с.

42. Цифровое преобразование изображений : Учеб. пособ. для вузов / Р. Е. Быков, Р. Фрайер, К. В. Иванов, А. А. Манцветов; Под ред. проф. Р. Е. Быкова. – М.: Горячая линия–Телеком, 2003. – 228 с.
43. Шлезингер М. Десять лекций по статистическому и структурному распознаванию / М. Шлезингер, В. Главач. – К.: Наукова думка, 2004. – 548 с.
44. Шпортько О. В. Алгоритми оптимізації розкладу LZ77 та вибору розмірів блоків динамічних кодів Хафмана для стиснення даних у форматі DEFLATE / О. В. Шпортько // Волинський математичний вісник. Зб. наукових праць. − Рівне: РДГУ, 2008. – № 5 (14) – С. 297-311. – (Серія: Прикладна математика).
45. Шпортько О. В. Оптимізація використання статичних предикторів у процесі стиснення зображень без втрат / О. В. Шпортько // Відбір і обробка інформації. – 2008. – Вип. 28 (104). – С. 82-89.
46. Шпортько О. В. Стиснення RGB-зображень без втрат із використанням палітри / О. В. Шпортько // Системні дослідження та інформаційні технології. – 2010. – № 2. – С. 26-36.
47. Шпортько О. В. Вибір найкоротших хеш-ланцюгів у процесі пошуку однакових послідовностей / О. В. Шпортько // Відбір і обробка інформації. – 2008. – Вип. 29 (105). – С. 85-90.
48. Шпортько О. В. Стиснення RGB-зображень без втрат з використанням палітри : Тези доп. / О. В. Шпортько // Системний аналіз та інформаційні технології : Матеріали X Міжнародної науково-технічної конференції. – К.: НТУУ "КПІ", 2008. – С. 424.
49. Шпортько О. В. Використання альтернативних блоків стиснутих даних у форматі PNG / О. В. Шпортько // Комп'ютерні науки та інформаційні технології : Матеріали третьої Міжнародної конференції CSIT'2008. – Львів: Видавництво ПП "Вежа і Ко", 2008. – С. 149-153.
50. Шпортько О. В. Алгоритм оптимізації результатів розкладу LZ77 за рахунок мінімізації зміщень / О. В. Шпортько // Вісник Національного університету водного господарства та природокористування. – 2009. – Вип. 2 (46), Ч. 1. – С. 378-385.

51. Шпортько О. В. Використання палітри для групового статистичного кодування RGB-зображень без втрат / О. В. Шпортько // Відбір і обробка інформації. – 2009. – Вип. 30 (106). – С. 125-132.
52. Шпортько О. В. Оптимізація блоків стиснутих даних у графічному форматі PNG / О. В. Шпортько // Вісник Національного університету "Львівська політехніка". – 2009. – № 653. – С. 223-231. – (Серія: Інформаційні системи та мережі).
53. Шпортько О. В. Вибір найкоротшого з альтернативних стиснутих блоків динамічних кодів Хафмана у форматі PNG / О. В. Шпортько // Комп'ютинг. – 2009. – Т. 8, вип. 2. – С. 58-67.
54. Шпортько О. В. Використання арифметичного кодування у форматі PNG : Тези доп. / О. В. Шпортько // Системний аналіз та інформаційні технології : Матеріали XI Міжнародної науково-технічної конференції. – К.: УНК "ІПСА" НТУУ "КПІ", 2009. – С. 598.
55. Шпортько О. В. Використання різницевих кольорових моделей для стиснення RGB-зображень без втрат / О. В. Шпортько // Відбір і обробка інформації. – 2009. – Вип. 31 (107). – С. 90-97.
56. Шпортько О. В. Вибір найкоротших хеш-ланцюгів в процесі пошуку співпадаючих послідовностей у декількох словниках / О. В. Шпортько // Комп'ютерні науки та інформаційні технології : Матеріали 4-ї Міжнародної конференції CSIT'2009. – Львів: Видавництво ПП "Вежа і Ко", 2009. – С. 220-224.
57. Шпортько О. В. Застосування різницевих кольорових моделей та корегувань значень предикторів в процесі стиснення RGB-зображень без втрат : Тези доп. / О. В. Шпортько // Системний аналіз та інформаційні технології : Матеріали XII Міжнародної науково-технічної конференції. – К.: УНК "ІПСА" НТУУ "КПІ", 2010. – С. 506.
58. Шпортько О. В. Аналіз зображень перед стисненням у форматі PNG / О. В. Шпортько / О. В. Шпортько // Комп'ютинг. – 2010. – Т. 9, вип. 2. – С. 192-204.

59. Шульгин В. И. Основы теории передачи информации. Ч. I. Экономное кодирование : Учеб. пособ. / В. И. Шульгин. – Харьков: Нац. аэрокосм. ун-т «Харьк. авиац. ин-т», 2003. – 102 с.
60. An Overview of JPEG-2000 / M. Marcellin, M. Gormish, A. Bilgin, M. Boliek // Proceedings of the 2000 IEEE Data Compression Conference, Snowbird, Utah. – Mar. 2000. – P. 523-541.
61. Burrows M. A Block-sorting Lossless Data Compression Algorithm / M. Burrows, D. Wheeler // SRC Research Report 124, Digital Systems Research Center, Palo Alto. – May 1994. – 24 p.
62. Boutell T. PNG Specification. Version 1.0 / Boutell T., et. all // RFC 2083, Boutell. Com, inc. – Mar. 1997. – 102 p.
63. Deutsch P. DEFLATE Compressed Data Format Specification version 1.3 / P. Deutsch // RFC 1951, 1996, Alladin enterprises. – May 1996. – 15 p.
64. Deutsch P. ZLIB Compressed Data Format Specification version 3.3 / P. Deutsch, J-L. Gailly // RFC 1950, 1996, Network Working Group. – May 1996. – 10 p.
65. Grosbois R. New approach to JPEG 2000 compliant Region Of Interest coding / R. Grosbois, D. Santa-Cruz, T. Ebrahimi // Published in SPIE's 46th annual meeting, Applications of Digital Image Processing XXIV, Proc. of SPIE, San Diego, CA, USA. – Jul-Aug. 2001. – Vol. 4472. – P. 267-275.
66. Huffman D. A Method for the Construction of Minimum Redundancy Codes / D. Huffman // Proceedings of the IRE. – Sept. 1952. – Vol. 40(9). – P. 1098–1101.
67. Li X. Edge-directed prediction for lossless compression of natural images / X. Li, M. Orchard // IEEE Transactions on Image Processing. – June 2001. – Vol. 10(6). – P. 813-817.
68. Moffat A. Arithmetic Coding Revisited / A. Moffat, R. M. Neal, I. H. Witten // ACM Transactions on Information Systems. – July 1998. – Vol. 16(3). – P. 256-294.
69. Shannon C. E. A Mathematical Theory of Communication / C. E. Shannon // Bell System Technical Journal. – July, Oct. 1948. – Vol. 27. – P. 379-423, 623-656.
70. Storer J. A. Data compression via textual substitution / J. A. Storer, T. G. Szymanski // Journal of ACM. – Oct. 1982. – Vol. 29(4). – P. 928-951.

71. Weinberger M. J. From LOCO-I to the JPEG-LS Standard / M. J. Weinberger, G. Seroussi // Hewlett-Packard Laboratories, Palo Alto, HPL-1999-3. – Jan. 1999. – 19 p.
72. Weinberger M. J. The LOCO-I lossless image compression algorithm: Principles and standardization into JPEG-LS / M. J. Weinberger, G. Seroussi, G. Sapiro // IEEE Transactions on Image Processing. – Aug. 2000. – Vol. 9(8). – P. 1309-1324.
73. Weinberger M. J. LOCO-I: A Low Complexity, Context-Based, Lossless Image Compression Algorithm / M. J. Weinberger, Seroussi G., Sapiro G. // Proceeding of Data Compression Conferece. – IEEE Computer Society Press. – Mar. 1996. – P. 140-149.
74. Welch T. A Technique for High-Performance Data Compression / T. Welch // IEEE Computer. – June 1984. – Vol. 17, № 6. – P. 8-19.
75. Witten I. H. Arithmetic Coding for Data Compression / I. H. Witten, R. M. Neal, J. G. Cleary // Communications of the ACM. – June 1987. – Vol. 30(6). – P. 520-540.
76. Wu X. An algorithmic study on lossless image compression / X. Wu // Proceeding of Data Compression Conference'1996. – Mar. 1996. – P. 150-159.
77. Zeng W. An Overview of the Visual Optimization Tools in JPEG 2000 / W. Zeng, S. Daly, S. Lei // To appear in Special Issue on JPEG 2000, Signal Processing: Image Communication Journal. – Jan. 2002. – Vol. 17, № 1. – P. 85-104.
78. Ziv J. A universal algorithm for sequential data compression / J. Ziv, A. Lempel // IEEE Transactions on Information Theory. – May 1977. – Vol. 23(3). – P. 337-343.
79. Ziv J. Compression of individual sequences via variable-rate coding / J. Ziv, A. Lempel // IEEE Transactions on Information Theory. – Sept. 1978. – Vol. 24(5). – P. 530-536.