computing@computingonline.net
www.computingonline.net

ISSN 1727-6209
International Journal of Computing

ВИБІР НАЙКОРОТШОГО З АЛЬТЕРНАТИВНИХ СТИСНУТИХ БЛОКІВ ДИНАМІЧНИХ КОДІВ ХАФМАНА У ФОРМАТІ PNG

Олександр Шпортько

Кафедра інформатики та прикладної математики
Рівненський державний гуманітарний університет, Україна
33028, м. Рівне, вул. С. Бандери, 12
E-mail: chportko@ukr.net, chportko@yandex.ru

Резюме: *у статті пропонується алгоритм генерування альтернативних стиснутих блоків динамічних кодів Хафмана для кожного блоку даних, вибору найкоротшого стиснутого блоку з альтернативних та ітеративного зменшення його розміру для покращення компресії зображень у форматі PNG. Детально розглядаються способи оцінки розміру блоку кодів Хафмана за абсолютними частотами його елементів. Наведено фрагмент підпрограми мовою С, яка дозволяє точно визначати розмір стиснутого блоку за допомогою принципу генерації динамічних кодів Хафмана та окремого опрацювання малих частот елементів, витрачаючи для цього в середньому не більше часу, ніж для наближеної оцінки розміру з використанням ентропії. Як показують експерименти, реалізація запропонованого алгоритму дозволяє покращити показники стиснення переважної більшості зображень у форматі PNG на 2 – 6 %.*

Ключові слова: *безвтратне стиснення зображень, динамічні коди Хафмана, формат графічних файлів PNG.*

ВСТУП

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 [1]. На сьогодні дизайнери та розробники Web-сайтів найчастіше зберігають фотореалістичні зображення у форматі JPEG, а дискретно-тонові – у форматі PNG. Крім цього, формат PNG найчастіше використовується для зберігання зображень, де втрати неприпустимі (наприклад, для рентгенівських знімків). Саме тому цей формат підтримується більшістю сучасних програм для перегляду і створення зображень. Проблема підвищення ефективності стиснення зображень у форматі PNG є актуальною сьогодні і буде актуальною в найближчому майбутньому, оскільки навіть наближення до її розв'язання дозволяє зменшити розміри відповідних файлів, що, в свою чергу, дає змогу прискорити їх завантаження з мережі та підвищити ефективність використання дискового простору. В даній роботі детально описується спосіб покращення показників стиснення зображень у цьому форматі за рахунок генерування та порівняння розмірів альтернативних стиснутих блоків, розгляд яких започатковано в [2].

1. ПРИНЦИПИ СТИСНЕННЯ ЗОБРАЖЕНЬ У ФОРМАТІ PNG. ПОСТАНОВКА ЗАДАЧІ

Будь-яке стиснення даних можливе за рахунок зменшення надлишковостей. Чим більше видів надлишковостей виявляє і опрацьовує компресор і чим краще він ці надлишковості усуває – тим якісніше він зможе стиснути націлені на таку обробку дані. Формат PNG орієнтований на зменшення надлишковостей чотирьох типів:

- між сусідніми пікселами, що мають, як правило, близькі кольори;
- між однаковими фрагментами зображення;
- між однаковими змінами кольорів;
- між переважаючими кольорами пікселів зображення.

Пікселі зображення записуються у PNG-файли найчастіше по рядках зверху вниз, а у кожному рядку – послідовно зліва направо (як символи у текстах), формуючи тим самим вхідний потік. Перед кодуванням ці пікселі можуть бути попередньо оброблені предикторами, принцип дії яких розглядається наприкінці цього розділу. У PNG-файлах, що

використовуються на сьогодні, стиснуті дані зберігаються у відокремлених блоках згідно формату словникового стиснення DEFLATE [1, 3]. Відповідно до цього формату, у стиснутих блоках містяться результати застосування до вхідного потоку алгоритму LZH [3], згідно з яким результати контекстно-залежного словникового алгоритму LZ77 [4] стискаються контекстно-незалежними кодами Хафмана.

Описуючи словникові алгоритми, фіксовану кількість попередніх закодованих неподільних елементів (літералів) вхідного потоку називають словником, а наступних незакодованих – буфером. Алгоритм LZ77 базується на заміні у вихідному потоці послідовності чергових літералів буфера посиланням на аналогічну послідовність літералів словника у вигляді пари чисел *<довжина; зміщення від закінчення словника>*. У випадку відсутності аналогічної послідовності літералів у словнику, перший літерал буфера переноситься у вихідний потік без змін. Після цього закодовані літерали переносяться з початку буфера в кінець словника і кодування продовжується аналогічно аж до закінчення літералів вхідного потоку. Наприклад, потік кодів біт пікселів *36 38 35 35 36 38 35 36 36 38 35 38 36 38 35 35 28* в закодованому вигляді записується як *36 38 35 35 <3; 4> 36 <3; 4> 38 <4; 12> 28*. Алгоритм LZ77 використовується у форматі PNG насамперед для зменшення надлишковостей між однаковими фрагментами зображення.

Під час декодування кодів алгоритму LZ77 окремі літерали копіюються у вихідний потік без змін. Пари ж *<довжина; зміщення>* декодуються шляхом послідовного копіювання з кінця вихідного потоку за вказаним зміщенням в кінець вихідного потоку необхідної кількості літералів. Природно, що алгоритм декодування має розрізняти окремі літерали та групи *<довжина; зміщення>*. Згідно з алгоритмом LZH, у форматі DEFLATE з цією метою довжини замін та окремі літерали кодуються разом числами в межах [0; 285]. При цьому числа з діапазону [0; 255] відповідають кодам окремих літералів, 256 позначає закінчення блоку, а числа з діапазону [257; 285] вказують на базові значення довжин. Після базових значень довжин міститься визначена форматом кількість біт, що разом з базовим значенням однозначно визначає довжину заміни. Зміщення зберігається після відповідної довжини аналогічно – у вигляді базового значення та додаткових біт. Базове значення зміщення знаходиться в межах [0; 29]. У форматі DEFLATE максимальне значення довжини закодованої послідовності може сягати 258, а зміщення – 32768.

Ідея використання контекстно-незалежних кодів Хафмана, що застосовуються для кодування окремих літералів і базових значень довжин та базових значень зміщень після виконання алгоритму LZ77, полягає у заміні чисел з більшою частотою (тут і надалі – абсолютною) кодами меншої кількості біт, ніж для чисел з меншою частотою. Коди Хафмана для літералів/довжин та зміщень визначаються для кожного блоку стиснутих даних, як правило, окремо (і називаються у цьому випадку динамічними), що сприяє покращенню стиснення. Автономне кодування Хафмана в форматі PNG насамперед зменшує надлишковості між переважаючими яскравостями пікселів зображення.

Згідно теореми Шеннона, елемент s_i з ймовірністю появи $p(s_i)$ найвигідніше кодувати $-\log p(s_i)$ бітами (тут і надалі логарифм береться за основою 2). Тоді середня довжина коду елемента після застосування контекстно-незалежного алгоритму має наближатися до *ентропії джерела* [3]:

$$H = -\sum_i p(s_i) \times \log\left(p(s_i)\right). \qquad (1)$$

Ентропія джерела зменшується при збільшенні нерівномірності розподілу ймовірностей між елементами. Середня довжина коду Хафмана співпадає з ентропією джерела лише тоді, коли для всіх елементів s_i довжини їх оптимальних кодів $-\log p(s_i)$ цілі. Чим більше $-\log p(s_i)$ відхиляються від цілих чисел, тим більшою стає різниця між середньою довжиною коду Хафмана і ентропією джерела, але ця різниця не перевищує одного біта.

Зменшити ентропію джерела при обробці зображень у форматі PNG намагаються за допомогою предикторів. *Предиктор* – це функція, що прагне, використовуючи значення відомих суміжних елементів, спрогнозувати (змоделювати) значення чергового елемента. Якщо піксел зображення характеризується декількома компонентами (наприклад, R, G, B), то предиктор кожної компоненти прогнозує значення згідно відповідних компонент сусідніх пікселів. В процесі використання цієї технології обчислюють і надалі кодують відхилення чергової компоненти від прогнозованого предиктором значення. Тому, у загальному випадку, процес застосування предикторів до кожної компоненти піксела у вузлі *(i, j)* можна записати формулою

$$\Delta_{ij} = C_{ij} - predict_{ij}, \qquad (2)$$

де C_{ij} – значення компоненти до застосування

предиктора, Δ_{ij} – значення компоненти після застосування предиктора, $predict_{ij}$ – значення предиктора, обчисленого для обраної компоненти.

Оскільки сусідні пікселі зображення мають, як правило, близькі кольори і тому близькі значення відповідних компонент, то часто значення предиктора буде співпадати зі значенням чергового елемента, найчастіше – буде близьким до цього значення і рідко – значно відрізнятиметься від нього. Тобто більшість значень Δ_{ij} будуть близькими до 0. Такий перерозподіл частот значень (а, отже, і ймовірностей) значно підвищує нерівномірність розподілу [5] і тому зменшує ентропію джерела (згідно (1)), а, отже, і довжину закодованої послідовності. Таким чином, предиктори використовуються, насамперед, для зменшення надлишковостей між сусідніми пікселами зображення, що мають близькі кольори. Крім цього, застосування алгоритму LZ77 та кодування Хафмана до результатів дії предикторів дозволяє зменшити надлишковості між однаковими змінами кольорів.

У стиснутих блоках формату PNG даним кожного рядка передує окремий байт, що визначає предиктор, який застосовується до компонент всіх його пікселів. На сьогодні форматом передбачено п'ять можливих значень цього байта [1], що визначає чотири різні предиктори: 0 – дані рядка не обробляються предикторами; 1 – предиктор рівний значенню відповідної компоненти зліва; 2 – предиктор рівний значенню компоненти зверху; 3 – предиктор рівний середньому арифметичному значень зліва та зверху; 4 – предиктор Піфа, що прогнозує значення у напрямку найменшого приросту. Опис предикторів формату PNG, міститься в [1], класифікація цих та інших предикторів наведена в [5].

На сьогоднішній день у спеціалізованому програмному забезпеченні для покращення стиснення зображень у форматі PNG найчастіше використовують метод перебору різних варіантів предикторів, розмірів блоків стиснутих даних та стратегій стиснення. **Метою** ж цієї **статті** є опис алгоритму мінімізації розміру стиснутих блоків динамічних кодів Хафмана за допомогою аналізу ефективності замін алгоритму LZ77.

2. АЛГОРИТМ ГЕНЕРУВАННЯ АЛЬТЕРНАТИВНИХ СТИСНУТИХ БЛОКІВ, ВИБОРУ НАЙКОРОТШОГО БЛОКУ З АЛЬТЕРНАТИВНИХ ТА ІТЕРАТИВНОГО ЗМЕНШЕННЯ ЙОГО РОЗМІРУ

Очевидно, що заміни LZ77 слід вважати ефективними, якщо вони записуються не більшою кількістю біт, ніж окремі літерали, які вони замінюють. Оскільки окремі літерали і базові значення довжин та зміщення у форматі DEFLATE записуються кодами Хафмана, то заміна j довжини len_j, що виконується починаючи з літерала s_k за зміщення $offset_j$, ефективна лише тоді, коли

$$\sum_{i=0}^{len_j-1} l_{s_{k+i}} \geq l_{len_j} + d_{len_j} + \lambda_{offset_j} + \delta_{offset_j}, \quad (3)$$

де l_m – довжина коду Хафмана літерала/довжини заміни m, d_m – кількість додаткових біт для запису довжини заміни m, λ_m – довжина коду Хафмана зміщення m, δ_m – кількість додаткових біт для запису зміщення m. Але короткі заміни виявляються, як правило, ефективними для стиснутих блоків, в яких інші заміни такої 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 Хафмана. З іншого боку, врахування сукупності коротких замін може суттєво зменшити частоти окремих літералів і тому збільшити довжини їх кодів Хафмана. Крім цього, на довжини (загальні кількості біт) стиснутих блоків суттєво впливають зміщення між однаковими фрагментами вхідного блоку даних, адже більші зміщення кодуються більшою кількістю додаткових біт.

У форматі DEFLATE мінімальна довжина кодів Хафмана рівна 1, максимальна – 15, максимальна кількість додаткових біт довжини складає 5, а зміщення – 13. Тому, згідно (3), для будь-яких розподілів частот ефективними будуть заміни довжиною від 48 літералів. На практиці зображення, в яких окреме значення яскравості 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 будемо вважати заміни від 24 елементів. Ефективність коротших замін залежить не лише від літералів, які вони замінюють, а й загалом від

розподілу стиснутого блоку. Для чергового блоку даних характер розподілу його літералів/довжин і, тим більше, зміщень наперед визначити неможливо, тому **для кожного вхідного блоку даних ми пропонуємо такий алгоритм мінімізації розміру відповідного стиснутого блоку:**

1) **створити альтернативні стиснуті блоки 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 використовуються на сьогодні для стиснення зображень у форматі PNG, або ж враховують всі заміни, або ж відкидають найкоротші заміни для всіх блоків. Ми ж, по суті, пропонуємо визначати мінімальний розмір врахованих замін для кожного блоку даних окремо так, щоб мінімізувати відповідні стиснуті блоки. 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 біт, загальну довжину альтернативного блоку з максимальною довжиною неврахованих замін j для кожного блоку даних будемо розраховувати за формулою:

$$L_j = \sum_{i=0}^{255} n_{ij} l_{ij} + l_{256,j} + \sum_{i=257}^{285} n_{ij}\left(l_{ij} + d_i\right) + \\ + \sum_{i=0}^{29} \eta_{ij}\left(\lambda_{ij} + \delta_i\right), \quad (4)$$

де n_i – частота елемента чи базової довжини i, η_i – частота базового зміщення i. Як видно з цієї формули, для обчислення довжини альтернативного стиснутого блоку не потрібно зберігати сам блок. Достатньо знати лише частоти елементів розподілів альтернативного стиснутого блоку та згенерувати коди Хафмана згідно цих розподілів. На практиці достатньо зберігати частоти розподілів 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 та враховувати відповідні літерали.

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

$$indexMinBlock = \min\left\{k \mid L_k = \min_{j=2,\,24} L_j\right\}. \quad (5)$$

Прискорити визначення найкоротшого з альтернативних стиснутих блоків можна шляхом перебору максимальних довжин відкинутих замін не до 24, а лише до 9, 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 (4) можна безпосередньо, після генерації відповідних кодів Хафмана. Прискорити ж виконання цих розрахунків можна за допомогою окремого зберігання додаткових біт замін (оскільки вони не залежать від розподілів частот) та обчислення розміру блоків кодів Хафмана літералів/довжин та зміщень лише за абсолютними частотами їх елементів, як це описано в наступному розділі. Наведемо фрагмент програми мовою С, що реалізує два розглянуті кроки алгоритму:

```
// розносимо частоти замін та відповідних їм
// літералів, що можуть виявитися неоптимальними
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 наявний
 // фіксуємо параметри заміни для аналізу відкидань:
 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
// встановлюємо ознаку наявності розподілу з усіма
nayavnoLenZamina[2]=true; // можливими замінами
// встановлюємо індекс попереднього наявного
pprNayavno=0; // альтернативного 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/замін
// для чергового альтернативного стиснутого блоку
sizeCode=countBitHufPackFreq(masAnalizLL, 286);
// накопичуємо частоти базових значень зміщень
for (i=0; i<=29; i++) // відкинутих замін
 {freqProbaDistance[k][i]+=
   freqProbaDistance[pprNayavno][i];
  // обчислюємо різниці частот баз зміщень
  masAnalizD[i]=freqDistanceLengthTable[i]-
   freqProbaDistance[k][i]; }
// додаємо довжину розподілу замін чергового
// альтернативного стиснутого блоку
sizeCode+=countBitHufPackFreq(masAnalizD, 30);
// накопичуємо додаткові біти відкинутих замін
plusBit[k]+=plusBit[pprNayavno];
// відкидаємо додаткові біти з обчисленої довжини
sizeCode-=plusBit[k];
if (sizeCode<minSizeCode) //черговий блок коротший
 {minSizeCode=sizeCode;//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
```

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

- згенерувати частоти розподілів найкоротшого з альтернативних стиснутих блоків, заміщуючи для замін, що не враховуються, в частотах розподілів з усіма замінами базові значення довжин та зміщень відповідними літералами;
- на основі сформованих частот розподілів розрахувати довжини кодів Хафмана літералів та базових значень довжин і зміщень;
- присвоїти елементам розподілів, що не використовуються, довжину коду Хафмана одиничного елемента, тобто максимальну довжину, оскільки такі елементи можуть з'явитися під час подальшого зменшення розміру найкоротшого з альтернативних блоків.

Зменшити довжину обраного стиснутого блоку можна за рахунок відкидання неефективних врахованих та врахування ефективних відкинутих замін згідно (3). Але врахування ефективних замін зменшує частоти окремих елементів і, відповідно, може збільшити після перерахунку довжини їх кодів Хафмана. Тому серед замін, неефективних з попередніми кодами Хафмана, 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 Хафмана, тому серед замін, ефективних з попередніми кодами Хафмана, можуть виявитися неефективні з перерахованими такими кодами. Ось чому для обраного найкоротшого блоку з альтернативних у випадку, коли він не враховує всі заміни, доцільно після аналізу ефективності замін, коротших 24, перерахувати довжини кодів Хафмана елементів розподілів та ітеративно проаналізувати ефективність цих замін ще раз. Додаткові ітерації для перевірки ефективності замін недоцільні, оскільки вони суттєво не впливають на довжину обраного стиснутого блоку та значно сповільнюють виконання алгоритму.

3. ОБЧИСЛЕННЯ РОЗМІРУ БЛОКУ КОДІВ ХАФМАНА ЗА АБСОЛЮТНИМИ ЧАСТОТАМИ ЙОГО ЕЛЕМЕНТІВ

У форматі DEFLATE для кожного стиснутого блоку генерується два блоки кодів Хафмана – для літералів/довжин та для зміщень. Визначення розміру таких блоків дозволяє обрати найкоротший стиснутий блок з альтернативних.

Наближено обчислити розмір таких блоків можна за допомогою ентропії, до якої наближається середня довжина кодів Хафмана [3]. Справді, нехай кожен з елементів S_i зустрічається N_i разів в послідовності довжиною $N = \sum_i N_i$. Тоді $p(s_i) = N_i / N$ і загальна довжина закодованих даних, враховуючи (1), наближається до значення

$$N \times H = N \log(N) - \sum_i N_i \log(N_i). \quad (6)$$

Мовою С підпрограма для такого наближеного обчислення розміру блоку кодів Хафмана має вигляд:

```
// masFreq – масив частот елементів
// countAllFreq – загальна кількість частот в масиві
size=0; n=0;
for (i=0; i<countAllFreq; i++)
 {n+=freq[i]; // сумуємо частоти
  if (freq[i]>1) // для коректності обчислень
   size+=freq[i]*log(freq[i]); }
if (n) size=(n*log(n)-size)/log(2);
else size=0;
```

Для точного обчислення розміру блоку кодів Хафмана скористаємося ітеративним принципом їх генерації [3]: серед всіх частот шукаються дві найменші і надалі розглядається їх сума (породжена вершина); після поєднання всіх частот в одну і формування відповідного дерева поєднань в зворотному порядку утворюються коди елементів за допомогою дописування до коду вершини одному з поєднаних елементів одиниці, а іншому – нуля (коренева вершина не кодується, якщо дерево містить хоча б два елементи). Тобто **внаслідок поєднання двох найменших частот довжина блоку кодів Хафмана збільшується на суму цих частот**. Цей принцип дозволяє ітеративно обчислювати розмір 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, знаходити в масиві частот щоразу дві найменші недоцільно, адже це значно сповільнить обчислення. Тому відсортуємо додатні частоти за спаданням і будемо поєднувати дві останніх частоти. Крім цього, сума двох чергових частот не може бути меншою суми двох попередніх поєднаних частот, оскільки щоразу обираються два найменших значення. Отже, чергова сума поєднання може бути вставлена перед попередньою сумую частот, що значно скорочує область пошуку позиції її вставки у відсортований масив. Наведемо фрагмент програми мовою С, що реалізує обчислення довжини блоку кодів Хафмана з використанням описаних підходів і бінарного пошуку позиції вставки частот у відсортований масив (тут ділення на 2 замінено побітовим зсувом):

```
// сортуємо частоти масиву за спаданням
for (i=0; i<countAllFreq; i++)
 if (masFreq[i]>0) // елемент в розподілі наявний
  {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, >0
if (countFreq==0) return 0; // якщо частоти відсутні
if (countFreq==1) // один елемент кодується бітом
 return masFreq[0];
countBit=0;
j=countFreq-2; // позиція для вставки поєднаних частот
while (countFreq>2)
 {// обчислюємо суму двох найменших частот
  element=masFreq[countFreq-1]+masFreq[countFreq-2];
  countBit+=element; // збільшуємо розмір блоку
  // шукаємо позицію для прямого включення суми
  while (j>0 && masFreq[j-1]<element) j--; // частот
  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]; }
```

Прискорити розрахунок довжини блоку кодів Хафмана 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.

Значно ж прискорити розрахунок довжини таких блоків дозволяє врахування особливостей розподілу їх частот: як свідчать експерименти, біля 67 % частот малі (до 15) і часто повторюються. Вставка таких частот у відсортований масив і подальше опрацювання займає багато часу, хоча можлива їх окрема ефективніша обробка. Розглянемо, наприклад, розподіл, в якому частота 1 повторюється 9 разів. Відразу можна визначити, що в процесі обробки буде поєднано 4 пари частот зі значенням 1 і внаслідок цих поєднань розмір блоку кодів збільшиться на 8 та буде створено 4 частоти зі значенням 2. Дев'ята частота зі значенням 1 поєднається з більшою частотою. 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**. Така оптимізація дозволяє зменшити кількість елементів у відсортованому масиві частот на 30-40 %. Використання ж двох описаних модифікацій дозволяє в середньому точно розраховувати довжину блоку кодів Хафмана не повільніше наближеної оцінки розподілу з використанням ентропії. Фрагмент програми мовою С, що реалізує всі описані модифікації, має вигляд:

```
// сортуємо в масиві частоти, більші 30, за спаданням;
// для менших частот підраховуємо їх кількість
for (i=0; i<countAllFreq; i++)
 if (masFreq[i]>0) // елемент в розподілі наявний
```

```
  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, >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 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
       // переходимо до більшої частоти
       goto nextElement; }}
  //вставляємо поєднанні частоти в кінець масиву
  for (j=30; j>=i; j--)
   if (masCountMinFreq[j]>0)
    for (k=0; k<masCountMinFreq[j]; k++)
     masFreq[countFreq++]=j; }
if (countFreq==1) // один елемент кодується бітом
 return masFreq[0];
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
  // враховуємо поєднання в довжині блоку
  countBit+=masFreq[bottom];
  // якщо сумарна частота більша 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; }
```

4. РЕЗУЛЬТАТИ ЕКСПЕРИМЕНТІВ

На завершення розглянемо результати застосування описаного алгоритму використання альтернативних стиснутих блоків для компресії восьми різнотипних 24-бітних зображень стандартного набору файлів ACT у форматі PNG (завантажити їх TIFF-версії можна, наприклад, з http://compression.ru/arctest/act/act-files.html).

Тестування проводилося за допомогою програми з CD до [1], у яку були внесені такі модифікації:

- забезпечена можливість виходу зі словника в буфер під час кодування повторів;
- реалізований вибір предиктора для рядка пікселів зображення на основі співставлення ефективності стиснення зображення без попередньої обробки з оцінкою нерівномірності розподілу після дії кожного предиктора;
- запроваджений аналіз альтернативних стиснутих блоків і кількостей додаткових біт при записі зміщень;
- розмір блоків даних збільшений до 64 Кб та відкинуті допоміжні текстові блоки.

Результати тестування наведено в табл. 1, 2, де показником компресії файлів обрано коефіцієнт стиснення, виражений в bpp, тобто у кількості біт, що в середньому витрачаються для кодування одного піксела зображення. Дані тестування програми без застосування розглянутого алгоритму наведено в третьому, а з застосуванням – у четвертому стовпці. Крім цього, для порівняння ефективності стиснення у другому стовпці таблиць вказано результати тестування програми Microsoft Photo Editor 2000, яка не застосовує предиктори, а в п'ятому та шостому – результати популярної серед Web-дизайнерів програми OptiPng (http://www.optipng.sourceforge.net), яка генерує короткі PNG-файли за результатами відповідно стандартного та максимального перебору предикторів, розмірів стиснутих блоків та стратегій стиснення.

Таблиця 1. Коефіцієнти стиснення (bpp) файлів зображень набору ACT у форматі PNG після застосування різних варіантів програм

Bmp-файл	Photo Editor 2000	Miano без алгор.	Miano з алгор.	OptiPng стан-дарт	OptiPng макс.
Clegg	9.06	5.69	5.00	5.37	5.32
Frymire	1.78	1.64	1.64	1.63	1.63
Lena	22.85	15.92	14.45	14.48	14.48
Monarch	18.36	13.16	12.61	12.51	12.49
Peppers	21.97	13.98	12.92	12.95	12.92
Sail	20.73	15.88	15.88	15.97	15.80
Serrano	1.84	1.72	1.72	1.70	1.70
Tulips	21.61	14.65	13.86	13.84	13.82
Середній	**14.77**	**10.33**	**9.76**	**9.81**	**9.77**
Сукупний	**10.89**	**7.70**	**7.29**	**7.35**	**7.32**

Таблиця 2. Час стиснення (c) файлів зображень набору ACT у форматі PNG різними варіантами програм на комп'ютері з частотою 300 МГц

Bmp-файл	Photo Editor 2000	Miano без алгор.	Miano з алгор.	OptiPng стан-дарт	OptiPng макс.
Clegg	3	14	15	72	996
Frymire	4	18	18	78	684
Lena	2	6	7	14	289
Monarch	3	12	13	33	733
Peppers	2	7	8	16	408
Sail	3	11	11	21	486
Serrano	2	8	8	23	336
Tulips	3	10	11	26	562
Разом	**22**	**86**	**91**	**283**	**4494**

Як свідчать результати тестування, застосування описаного алгоритму покращило коефіцієнт стиснення на 2 – 6% для 63% зображень, які, як правило, є неперервно-тоновими, та не вплинуло на компресію решти файлів, хоча й сповільнило виконання програми в середньому на 7%. Крім цього, реалізація розглянутого алгоритму дозволила наблизитися до коефіцієнта стиснення програми перебору, а для деяких зображень і перевершити його, витрачаючи для компресії в 1.9 – 66 разів менше часу.

5. ВИСНОВКИ

1. Для підвищення ефективності стиснення даних у форматах, що послідовно використовують алгоритми декількох методів, слід враховувати взаємний вплив цих алгоритмів.
2. Підвищити швидкість розрахунку розміру блоку кодів Хафмана за частотами елементів можна не лише з допомогою швидких алгоритмів сортування та використання списків, а й шляхом окремого попереднього опрацювання малих частот.
3. Розглянутий алгоритм дозволяє суттєво покращити коефіцієнт стиснення більшості зображень за рахунок вибору для кожного блоку даних найкоротшого з альтернативних стиснутих блоків динамічних кодів Хафмана та ітеративного зменшення його розміру.
4. Описаний алгоритм не вимагає модифікації декодера чи програм перегляду зображень і за рахунок зменшення розмірів файлів лише прискорює їх роботу. Саме тому він може бути ефективно використаний для збереження даних у стандартах, що використовують формат словникового стиснення DEFLATE.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

[1] Миано Дж. Форматы и алгоритмы сжатия изображений в действии: учеб. пособ. / Дж. Миано. – М. : Триумф, 2003. – С. 249-318. – (Практика программирования).

[2] 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.

[3] Методы сжатия данных. Устройство архиваторов, сжатие изображений и видео / Д. Ватолин, А. Ратушняк, М. Смирнов, В. Юкин. – М. : ДИАЛОГ-МИФИ, 2003. – С. 17-106.

[4] Ziv J., Lempel A. A universal algorithm for sequential data compression / J. Ziv, A. Lempel // IEEE Transactions on Information Theory. – May 1977. – Vol. 23(3). – P. 337-343.

[5] Бредихин Д. Ю. Сжатие графики без потерь качества [Електронний ресурс] / Д. Ю. Бредихин. – 2004. – http: //www.compression.ru/download/articles/i_less/bredikhin_2004_lossless_image_compression_doc.rar

***Олександр Шпортько**, аспірант кафедри інформатики та прикладної математики Рівненського державного гуманітарного університету. Закінчив Рівненський державний педагогічний інститут у 1997 році за спеціальністю "Прикладна математика".*

Наукові інтереси: стиснення даних без втрат, теорія інформації, теоретичні основи програмування, проектування та розробка баз даних.