УДК 004.043

А.Я. Бомба, О.В. Шпортько
Рівненський державний гуманітарний університет,
кафедра інформатики та прикладної математики

АЛГОРИТМ ОПТИМІЗАЦІЇ ВИБОРУ ФІЛЬТРА ДЛЯ ПОПЕРЕДНЬОГО ОПРАЦЮВАННЯ ЗОБРАЖЕНЬ ПЕРЕД СТИСНЕННЯМ НА ОСНОВІ МЕТОДУ "ПРЕДИКТОР – КОРЕКТОР"

Проаналізовано особливості фільтрування для попереднього опрацювання зображень перед стисненням, обґрунтовано можливість та описано алгоритм оптимізації вибору фільтра на основі методу "предиктор–коректор". Наведено фрагменти програм для реалізації запропонованого алгоритму та результати його застосування для стиснення зображень у форматі PNG.

The features of filtration at the previous images processing before the compression are considered. The possibility is proved and the algorithm of the optimization of choice of filter is described on the basis of method of "is prognostication – is proof-reader". The fragments of the programs for the realization of the offered algorithm and results of its application for the compression of images in the format of PNG are presented.

Вступ

Сьогодн існує декілька основних методів та безліч алгоритмів стиснення даних [4]. Однак проблема стиснення зображень у зв'язку зі стрімким розвитком комунікаційних технологій та підвищенням інформаційних потреб суспільства не втрачає своєї актуальності. Більшість відомих алгоритмів, що дають змогу стиснути зображення у сотні разів (JPEG, фрактальних та вейвлет-перетворень), призводять до незначних втрат якості зображень. Такі втрати непомітні для відеоданих, але впливають на якість статичних зображень. Алгоритми стиснення зображень без втрат хоча й стискають зображення значно слабше, зате не погіршують їхньої якості. Тому для стиснення відеоданих найчастіше використовують алгоритми з втратами, а для статичних зображень – без втрат. Ці алгоритми модифікуються та вдосконалюються паралельно, покращуючи з кожним роком свої характеристики. При цьому оптимізуються не лише алгоритми безпосереднього стиснення, а й розвиваються алгоритми попередньої обробки (препресингу), що дають змогу підвищити надлишковість даних.

Застосування фільтрування для попереднього опрацювання зображень. Аналіз останніх досліджень. Постановка задачі

Більшість сучасних архіваторів та форматів графічних файлів для стиснення зображень без втрат використовують як один з основних етапів словниковий метод [4]. Існує два різновиди алгоритмів словникового методу [1, 2]. Перший з них бере початок від алгоритму LZ77 і ґрунтується на заміні послідовності чергових елементів потоку посиланням на аналогічну закодовану послідовність елементів у вигляді пари чисел *<довжина; зміщення від закінчення закодованої частини потоку>*, якщо така послідовність зустрічалася раніше. Другий бере початок

від алгоритму LZ78 і ґрунтується на заміні послідовності чергових елементів потоку індексом аналогічної послідовності у словнику, що формується під час виконання алгоритму. Оскільки словникові алгоритми використовують опрацьовану раніше послідовність, то вони належать до класу контекстно-залежних. Методи цього типу покликані усувати залежності між різними послідовностями елементів. Для покращання стиснення даних результати дії словникових алгоритмів, як правило, стискуються одним з контекстно-незалежних алгоритмів (арифметичним чи Хафмана). Ідея використання цих алгоритмів полягає у заміні елементів з більшою частотою послідовностями меншої кількості біт, ніж для елементів з меншою частотою. При цьому середня довжина коду елемента після застосування контекстно-незалежного методу, за теоремою Шеннона, має наближатися до величини

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

де $p(s_i)$ – ймовірність появи елемента s_i. Цю величину також називають *ентропією джерела* [4]. Ентропія джерела зменшується при збільшенні нерівномірності розподілу ймовірностей між елементами.

Зменшити ентропію джерела під час опрацювання зображень найчастіше намагаються за допомогою алгоритмів попередньої обробки. Один з таких алгоритмів отримав назву фільтрування. Ми розглянемо його функціонування на прикладі формату PNG [5]. *Фільтр* – це функція, що намагається, використовуючи значення відомих суміжних елементів, спрогнозувати (змоделювати) значення чергового елемента. Якщо піксел зображення характеризується декількома компонентами (наприклад, R, G, B), то фільтр кожної компоненти прогнозує значення згідно з відповідними компонентами сусідніх пікселів. З використанням цієї технології обчислюють і надалі кодують відхилення чергової компоненти від прогнозованого фільтром значення. Тому у загальному випадку процес фільтрування кожної компоненти піксела у вузлі *(i, j)* можна записати формулою

$$\Delta_{ij} = F_{ij} - filtr_{ij},$$

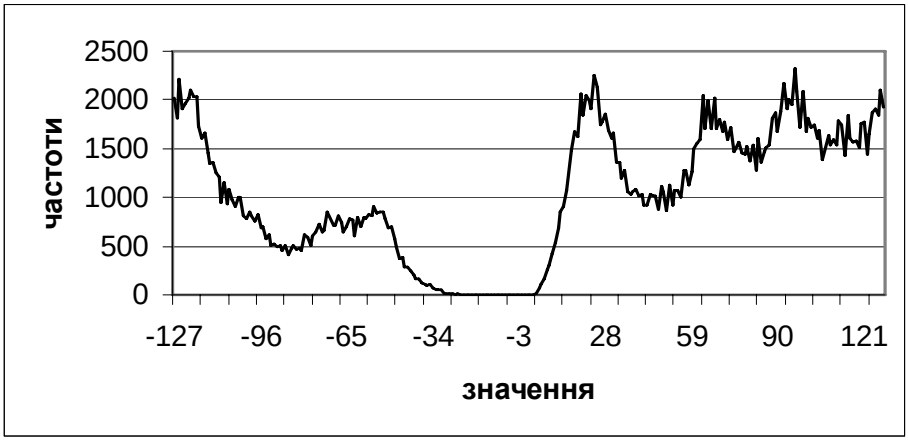
де F_{ij} – значення компоненти до фільтрування, Δ_{ij} – значення компоненти після фільтрування, $filtr_{ij}$ – значення фільтра, що прикладається до обраної компоненти.

Як зазначено в [3], "Відхилення між значеннями сусідніх елементів в зображеннях найчастіше зумовлені двома причинами: "сильними" коливаннями, що обумовлюються зображеними об'єктами – трендом, і слабкими фоновими коливаннями – шумом. Тому можливі два протилежні типи моделей:

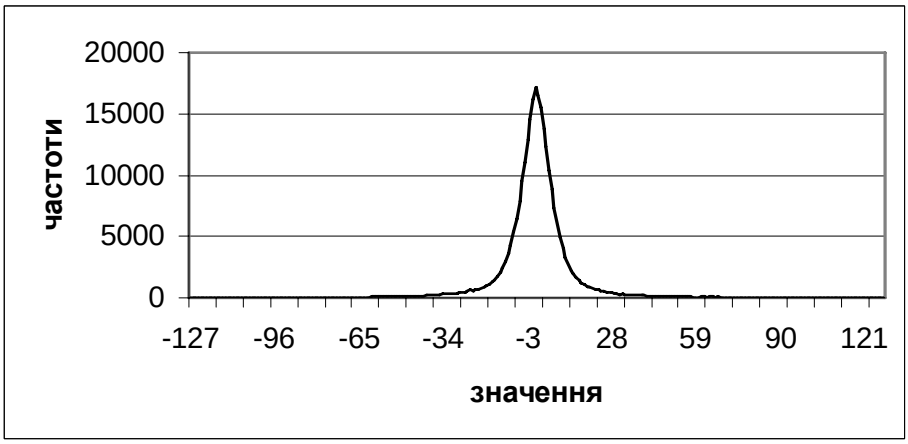
- внесок шуму незначний порівняно з внеском тренду;
- внесок тренду незначний порівняно з внеском шуму.

У першому випадку розумно спрогнозувати значення чергового елемента, виходячи з тенденції, що склалася, у другому – як середнє арифметичне обраних попередніх елементів".

Оскільки сусідні пікселі зображення мають, як правило, близькі кольори і отже, близькі значення відповідних елементів, то часто значення фільтра збігатиметься 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 [3] і тому зменшує ентропію джерела (згідно з (1)), а, отже, і довжину закодованої послідовності. Приклад перерозподілу частот значень компонент при застосуванні фільтра наведено на рис. 1.



а



б

Рис. 1. Розподіл частот значень зеленої компоненти зображення Lena.bmp до застосування фільтра (а) та після застосування Left-фільтра

Фільтрування виконується побайтно над аналогічними компонентами суміжних пікселів. Саме тому воно неефективне у випадках, коли глибина кольору менша за вісім біт. Процес фільтрування, як правило, здійснюється за рядками згори донизу, зліва направо за правилами цілочислової арифметики і має бути оберненим. Тому використовувати при цьому значення компонент справа та знизу відносно активної компоненти неможливо. Отже, позначимо компоненти сусідніх пікселів для компоненти X за схемою:

LeftAbove	Above
Left	X

У форматі PNG даним кожного рядка передує окремий байт, що визначає фільтр компонент всіх його пікселів. Сьогодні форматом передбачено п'ять можливих значень цього байта [5], що визначає чотири різні фільтри: 0 – дані рядка не фільтруються, 1 – субфільтр (sub filter); 2 – суперфільтр (up filter); 3 – усереднюючий фільтр (average filter); 4 – фільтр Піфа (Paeth filter). Мовою С ці фільтри, а також фільтр Med (median edge detection) можна реалізувати так:

```
char SubFilter(char Left, char Above, char LeftAbove)
 {return Left;
 }
char UpFilter(char Left, char Above, char LeftAbove)
```

```
 {return Above;
 }
char AverageFilter(char Left, char Above, char LeftAbove)
 {return (Left + Above) / 2;
 }
char MedFilter(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;
 }
char PaethFilter(char Left, char Above, char UpperLeft)
 {int pp = Left + Above – UpperLeft;
 int pa, pb, pc;
 pa = abs(pp – Left);
 pb = abs(pp – Above);
 pc = abs(pp – UpperLeft);
 if (pa <= pb && pa <= pc)
  return Left;
 else
  if (pb <= pc)
   return Above;
  else
   return UpperLeft;
 }
```

Фільтр *LeftFilter* прогнозує, що значення чергового елемента дорівнює значенню зліва, фільтр *AboveFilter* – дорівнює значенню згори, фільтр *AverageFilter* – середньому арифметичному цих значень. Ці три фільтри описують шумову модель і належать до лінійних статичних фільтрів. Наступні два фільтри описують трендову модель і належать до нелінійних статичних фільтрів.

Фільтр Піфа *PaethFilter* 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 фільтр використовується при стисненні найчастіше (зокрема, в архіваторі WinRAR).

Фільтр *MedFilter* намагається адаптуватися до локальних горизонтальних та вертикальних ребер. Значення *Left* найчастіше повертається у випадку виявлення горизонтального, а *Above* – у випадку виявлення вертикального ребра. Якщо ребро не виявлено, то повертається значення площини над точкою *X*, що проходить у тривимірному просторі через точки *Left*, *Above* та *LeftAbove*. Цей фільтр використовується у форматі стиснення JPEG-LS. Опис інших фільтрів можна віднайти у [3]. Проблема вибору фільтрів рядків для стиснення конкретних зображень залишалася невирішеною до цього часу [5, с. 317]. У цій статті пропонуються і описуються два нові способи її вирішення.

Алгоритм оптимізації

Розглянемо питання оптимізації вибору фільтра для **кожного** рядка з метою досягнення **загальних** оптимальних результатів стиснення.

Спочатку проаналізуємо результати застосування описаних фільтрів на стандартному наборі ACT з восьми файлів 24-бітних зображень [6] за допомогою модифікованої програми з компакт-диску до [5]. Результати тестування наведено в табл. 1 (коефіцієнт стиснення (КС) – це процент зменшення початкового розміру файла). Для порівняння в цій же таблиці наведено результати збереження у форматі PNG за допомогою програми MS Photo Editor 2000.

Таблиця 1

Розміри файлів (Kb) набору ACT при застосуванні фільтрів та їх комбінацій

Формат \ Файл	1	2	3	4	5	6	7	8	Разом	КС, %
BMP-формат	2101	3622	769	1153	769	1153	1464	1153	12184	0,00
PNG-формат (Photo Editor)	793	269	732	882	704	996	112	1038	5526	54,65
PNG-формат без фільтр.	504	**266**	704	817	652	899	**106**	962	4910	59,70
Sub фільтр	504	355	546	669	459	798	144	766	4241	65,19
Up фільтр	**488**	369	518	660	485	810	**150**	729	4209	65,45
Average фільтр	1200	538	**513**	**631**	**452**	**764**	224	**707**	5029	58,72
Paeth фільтр	505	373	523	647	**452**	783	152	**708**	4143	66,00
Комбінування 1	505	366	525	729	536	845	153	908	4567	62,52
Комбінування 2	505	372	697	664	462	802	153	735	4390	63,97
Комбінування 3	503	371	512	637	451	776	153	706	4109	66,28
Комбінування 4	486	266	512	632	448	767	109	705	3925	67,79
WinRar 3.00 (для порівняння)	503	204	473	517	369	587	89	588	3330	72,67

Як видно з табл. 1, не існує універсального фільтра, застосування якого сприяло б найкращому стисненню всіх типів зображень. Навіть для сусідніх рядків зображень оптимальними можуть виявитися різні фільтри.

Стандартом PNG рекомендується вибирати найкращий фільтр для кожного рядка, визначаючи мінімальну суму значень знакових байт його пікселів при застосуванні кожного з наявних фільтрів до нефільтрованих даних. Дійсно, зменшення загальної суми знакових байт пікселів рядка, як правило, свідчить про зменшення їх абсолютних значень, а, отже, сприяє покращанню показників стиснення, але чи є цей критерій визначальним? Результати застосування такого способу вибору оптимального фільтра кожного рядка наведено у табл. 1 в рядку *Комбінування 1*.

Інший спосіб визначення оптимального фільтра для кожного рядка полягає у підрахунку кількостей п'яти однакових значень байт, що йдуть підряд [5]. Дійсно, повторення значень сприяє покращанню показників стиснення, але цей спосіб не враховує можливості повторень груп різних байт, що також покращує стиснення. Результати застосування такого способу вибору оптимального фільтра кожного рядка наведено в табл. 1 в рядку *Комбінування 2*.

На наш погляд, для кожного рядка треба обирати той фільтр, застосування якого породжує мінімальну суму **модулів** значень знакових байт (з діапазону [-128;127]). Зменшення загальної суми модулів значень рядка після застосування фільтра свідчить про збільшення нерівномірності розподілу частот навколо нуля (див. рис. 1) та покращує результати застосування контекстно-незалежного алгоритму. Результати застосування такого способу вибору оптимального фільтра кожного рядка наведено у табл. 1 в рядку *Комбінування 3*. Як видно з таблиці, такий спосіб вибору оптимального фільтра покращує коефіцієнт стиснення стосовно способу стандарту на 3,76%, а стосовно фільтра Піфа – на 0,28%.

Крім цього, нами пропонується спосіб оптимізації вибору фільтра рядка на основі попереднього прогнозування кількості біт, що знадобляться для стиснення фільтрованих даних. Серед всіх фільтрів обиратимемо той, що генерує фільтровані дані з найменшою прогнозованою кількістю біт після стиснення. Цей спосіб не вимагає попереднього стиснення фільтрованих даних. Він лише попередньо оцінює кількість біт, що знадобиться для стиснення при використанні кожного з фільтрів, тому практично тут використано метод *предиктор – коректор*.

У форматі PNG для стиснення даних після фільтрування послідовно застосовується контекстно-залежний словниковий алгоритм LZ77 [1] і контекстно-незалежний алгоритм стиснення кодами Хафмана [4]. Особливості стиснення даних у цьому форматі описано в [5, 6]. Врахуємо також, що при застосуванні фільтрів можуть з'явитися нові чи зникнути існуючі збіжні послідовності, що відповідно підвищить або знизить ефективність подальшого застосування алгоритму LZ77.

Наведемо фрагменти алгоритму мовою С, що реалізують запропонований спосіб оптимізації вибору фільтра для рядка пікселів. Оцінювати фільтровані дані кожного рядка будемо з використанням згаданих алгоритмів стиснення. Для цього використаємо буфер з двох попередніх рядків, оскільки буфер меншого розміру спотворює результати прогнозування, а більшого – вимагає додаткового часу на опрацювання й істотно не впливає на якість стиснення. Поновлювати буфер двох попередніх рядків будемо після визначення фільтра чергового рядка:

```
// посуваємо фільтр попереднього рядка
memcpy(buffers, buffers + row_width, row_width);
// записуємо результати дії оптимального фільтра current_filter в буфер
memcpy(buffers + row_width, filter_buffers[current_filter], row_width);
```

Частоти використання елементів/довжин та зміщень зберігатимемо у відповідних масивах. Для оцінки кількості біт, що будуть використані для зберігання даних згідно з накопиченими частотами, скористаємося ітеративним принципом генерації кодів Хафмана: серед всіх символів знаходимо два з найменшою частотою, і розмір їхніх кодів збільшуємо на одиницю, після чого поєднуємо віднайдені частоти, тобто кількість використаних біт зростає на суму їхніх частот:

```
unsigned int countBitHuffmanPackFreq (unsigned int *masFreq, unsigned int countAllFreq)
 // masFreq – масив накопичених частот
 // countAllFreq – кількість елементів в масиві накопичених частот
 {unsigned int countFreq = 0, i, j, k,element, countBit;
  // формує відсортовану за спаданням послідовність ненульових частот
// методом прямого ввімкнення
  for (i = 0; i < countAllFreq; i++)
   if(masFreq[i] > 0)
    {j = 0;
     element = masFreq[i];
     while(j < countFreq && masFreq[j] >= element)
      j++; // шукаємо позицію для прямого ввімкнення
     for (k = countFreq; k > j; k--)
      masFreq[k] = masFreq[k – 1]; // розсуваємо існуючі елементи
     masFreq[j] = element; // вставляємо віднайдений ненульовий елемент
     countFreq++;
    }
  if (countFreq == 0)
   return 0;
  if (countFreq == 1)
   return masFreq[0];
  countBit=0;
  while (countFreq > 2) // поки кількість ненульових частот перевищує 2
   {j=0;
    element = masFreq[countFreq – 1] + masFreq[countFreq – 2]; // поєднуємо частоти
    countBit += element; // к-ть використаних біт збільшуємо на суму частот
    while(j < countFreq – 2 && masFreq[j] >= element)
     j++; // шукаємо позицію для прямого ввімкнення поєднаної частоти
    for (k = countFreq – 2; k > j; k--)
     masFreq[k]=masFreq[k – 1];
    masFreq[j]=element;
    countFreq--;
   }
  // два символи з найбільшими частотами кодуються бітом
  return countBit + masFreq[0] + masFreq[1];
 }
```

Функція оцінки кількості біт для кодування кожного результату фільтрування має вигляд:

```
int countBitPackBuffer(unsigned char *buffer, unsigned int lenBuffer, unsigned int startPoz)
 // startPoz – початкова позиція буфера для оцінки к-ті біт кодування
 // lenBuffer – загальна довжина буфера з врахуванням попередніх рядків
 {int countBit = 0;
  unsigned int freqLen[286], freqOffset[30], baseKod, countBitKod, valueBitKod;
  memset(freqLen, 0, 286*sizeof(unsigned int)); // обнулюємо масиви частот
  memset(freqOffset, 0, 30*sizeof(unsigned int));
  while (startPoz < lenBuffer – 2) // поки не оцінено весь буфер
   {// шукаємо збіжні послідовності
    int lenLZ = 0, zmLZ, zm = startPoz – 1, len;
    while (zm >= 0)
     {// згідно методу LZ77, довжина збігу не може бути меншою за три
       if ((buffer[zm] == buffer[startPoz]) && (buffer[zm + 1] == buffer[startPoz + 1]) &&
  (buffer[zm + 2] == buffer[startPoz + 2]))
       {len = 3; // визначаємо довжину збігу
        while (startPoz + len < lenBuffer)
         {if (buffer[zm + len] != buffer[startPoz + len])
           break;
          len++;
          if (len == 258) // максимальна довжина збігу
           break;
         }
        if (len > lenLZ)
         {lenLZ = len;
          zmLZ = startPoz – zm;
          if (lenLZ >= 258)
           {lenLZ = 258;
            break;
           }
         }
       }
      zm-= 1;
     }
    if (lenLZ > 2) // вдалося віднайти збіжну послідовність
     {// визначаємо кількість додаткових біт та обробляємо код довжини
      LengthToCode (lenLZ, baseKod, countBitKod, valueBitKod);
      countBit+= countBitKod;
      freqLen[baseKod]++;
      // визначаємо кількість додаткових біт та обробляємо код зміщення
      DistanceToCode(zmLZ, baseKod, countBitKod, valueBitKod);
      countBit+= countBitKod;
      freqOffset[baseKod]++;
      startPoz+= lenLZ; // переміщуємося за знайдену послідовність
     }
    else // збіжна послідовність відсутня – кодуємо символ
     {freqLen[buffer[startPoz]]++;
      startPoz++;
     }
   }
  countBit+= countBitHuffmanPackFreq(freqLen, 286);
```

```
 countBit+= countBitHuffmanPackFreq(freqOffset, 30);
 return countBit;
}
```

Використовуючи цю функцію, вибір фільтра для кожного рядка здійснюємо так:

```
current_filter = 0; // визначаємо прогноз кодування буфера без кодування
memcpy (buffers + 2*row_width, filter_buffers[0], row_width);
// обчислюємо вартість кодування рядка без фільтрування
long longestrun = countBitPackBuffer(buffers, 3*row_width, 2*row_width);
// порівнюємо з результати дій всіх фільтрів
for (ii = 1; ii < FilterBufferCount; ++ii)
 {memcpy(buffers + 2*row_width, filter_buffers[ii], row_width);
  long run = countBitPackBuffer(buffers, 3*row_width, 2*row_width);
  if (run < longestrun)
   {current_filter = ii;
    longestrun = run;
   }
 }
```

Результати застосування описаного способу оптимізації вибору фільтра кожного рядка на основі методу *предиктор – коректор* наведено в табл. 1 в рядку *Комбінування 4*. Як видно з таблиці, такий спосіб вибору фільтра дає кращі результати від відомих раніше способів на 3,8 % (465 Кб).

Розглянемо тепер проблему витрат часу при різних способах вибору оптимального фільтра. В табл. 2 наведено час стиснення файлів набору АСТ без фільтрування та при різних способах оптимізації вибору фільтра.

Таблиця 2

Час стиснення (в секундах) файлів набору АСТ при різних варіантах фільтрування

Формат\Файл	1	2	3	4	5	6	7	8	Разом
Photo Editor (без фільтрування)	3	4	2	3	2	3	2	3	22
Базова програма (без фільтрування)	17	36	7	11	7	12	14	11	115
Комбінування 1	22	34	10	16	10	14	14	14	134
Комбінування 2, 3	22	34	10	16	10	14	14	16	136
Комбінування 4	241	318	67	142	66	85	60	121	1100
Вдосконалене комбінування 4	44	58	29	40	28	44	25	46	314

Як видно з табл. 2, описаний вище алгоритм вибору оптимального фільтра сповільнює процес стиснення у 8,5 раза стосовно базового алгоритму. Тому для прискорення виконання алгоритму було виконано такі модифікації:

- впроваджено хеш-функцію для прискорення пошуку збіжних послідовностей та черги вказівок на відповідні елементи обробленої послідовності;
- реалізовано створення черг згідно з хеш-функцією для буфера двох попередніх рядків. Такі черги генеруються лише один раз після модифікації буфера і використовуються для аналізу ефективності кожного фільтра;
- встановлено максимальну кількість елементів кожної черги для значень хеш-функції на рівні 128 вказівок;
- замінено пряме включення двійковим при аналізі частот кодів/довжин та зміщень.

Такі вдосконалення дали змогу скоротити витрати часу більше ніж в 3 рази, хоча й після їх застосування розроблений алгоритм працює значно повільніше від базового (див. останній рядок з табл. 2).

Висновки

З наведених результатів дослідження ефективності окремих фільтрів та способів оптимізації вибору фільтрів для кожного рядка і часу на їх виконання доходимо таких висновків:

1. Застосування фільтрів підвищує у середньому коефіцієнт стиснення не менше, ніж на 5%, що виправдовує їх використання;
2. Використання нелінійних трендових фільтрів загалом ефективніше від лінійних шумових на 0,5%, хоча й вимагає виконання більшої кількості обчислень;
3. Статичні фільтри, що використовуються для попереднього опрацювання зображень перед стисненням, не повинні негативно впливати на властивості потоку даних, що використовуються основним алгоритмом стиснення (наприклад, для словникових методів фільтри не повинні зменшувати кількість збіжних послідовностей);
4. Запропонований алгоритм оптимізації вибору фільтрів підвищує коефіцієнти стиснення всіх файлів набору ACT в середньому на 3,8%, хоча й працює в 2,5 рази повільніше від відомих раніше алгоритмів;
5. Розроблений алгоритм не вимагає модифікацій декодера та програм перегляду зображень і за рахунок зменшення розмірів файлів лише прискорює їх роботу. Саме тому він може бути використаний для оптимізації фільтрування графічних файлів у форматах типу PNG.

1. Ziv J. Lempel A. A universal algorithm for sequential data compression // IEEE Transactions on Information Theory. May 1977. Vol. 23(3). – P. 337–343. 2. Ziv J. Lempel A. Compression of individual sequences via variable-rate coding // IEEE Transactions on Information Theory. Sept. 1978. Vol. 24(5). – P. 530–536. 3. Бредихин Д.Ю. Сжатие графики без потерь качества. – 2004. http://www.compression.ru/download/articles/ i_lless/bredikhin_2004_lossless_image_compression_doc.rar. 4. Методы сжатия даннях. Устройство архиваторов, сжатие зображений и видео / Д. Ватолин, А. Ратушняк, М. Смирнов, В. Юкин. – М.: ДИАЛОГ-МИФИ, 2003. – С. 75–118. 5. Миано Дж. Форматы и алгоритмы сжатия зображений в действии: Учеб. пособ. – М.: Триумф, 2003. – С. 249–318. 6. Шпортько О.В. Алгоритми оптимізації розкладу LZ77 та вибору розмірів блоків динамічних кодів Хафмана для стиснення даних у форматі DEFLATE // Економіка та інформаційні технології. Зб. наукових праць кафедри економічної кібернетики.– Рівне: РДГУ, 2006. – Вип. 4 – С. 104–108.