МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ

ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

Кафедра прикладної математики

**КУРСОВА РОБОТА**

з дисципліни «Програмування»

на тему:

СТИСКАННЯ ЗА ХАФМАНОМ. ДЕРЕВО ХАФМАНА.

Виконав:

студент І курсу групи КМ-84,

спеціальність 113

Прикладна математика

КУЛИК К. А.

Керівник:

ЛЮБАШЕНКО Н.Д.

Національна оцінка:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Кількість балів: \_\_\_\_\_\_

Оцінка ECTS:\_\_\_\_\_\_\_\_

Київ – 2019

ЗМІСТ

[ВСТУП 3](#_Toc5389841)

[Постановка задачі 4](#_Toc5389842)

[Вибір методу розв’язання 5](#_Toc5389843)

[Алгоритм обраного методу 6](#_Toc5389844)

[Опис програми 7](#_Toc5389846)

[Результати. Контрольний приклад 10](#_Toc5389847)

[ВИСНОВКИ 14](#_Toc5389849)

[Література 15](#_Toc5389850)

[Додаток А. Фрагмент тексту програми на мові С 16](#_Toc5389851)

# ВСТУП

У будь-який час дуже важливою характеристикою інформації є її розмір. Особливо зараз, коли електронні дані надходять с багатьох джерел, було б корисно вміти «економити» простір на носіях. Але як же можна стиснути двійкові дані, в яких кожний байт, кожний біт має значення, і просто так «відкинути» певний шматок даних, не знаючи, який зміст вони несуть, неможливо.

Виникає закономірне питання: чи існують алгоритми стиснення без втрат?

Відповідь: існують. Серед них почесне місце одного з «першовідкривачів» належить алгоритму Девіда Хаффмана. Це простий адаптивний жадібний алгоритм оптимального префіксного кодування алфавіту з мінімальною надмірністю. На відміну від алгоритму Шеннона — Фано, алгоритм Хаффмана залишається завжди оптимальним і для вторинних алфавітів з більш ніж двома символами.

Отож, нашим об’єктом дослідження будуть дані, представлені у вигляді масиву двійкових символів (файлу), а предметом дослідження буде кодування та декодування за допомогою алгоритму Хаффмана.

# Постановка задачі

На вході: таблиця частот символів.

На виході: дерево кодування Хаффмана (Н-дерево).

Зазвичай практичного сенсу від побудови дерева немає, тому більш складна задача полягає у подальшому використанні цього дерева, наприклад, для кодування тексту. Тоді вона може бути поставлена наступним чином:

1. Кодування:

На вході: певний текст, частоти символів для якого необхідно буде порахувати.

На виході: закодований текст та дерево.

1. Декодування

На вході: закодований текст та дерево.

На виході: декодований текст.

1. Візуалізація

На вході: певний текст, частоти символів для якого необхідно буде порахувати.

На виході: лише дерево, але вже не в оптимальному форматі, а у зручному для унаочнення, наприклад, у вигляді графічного зображення.

Дерево Хаффмана (також Н-дерево) являється простим двійковим деревом, де ребрам відповідають символи 0 та 1.

Для того, щоб знайти символ за його кодом, необхідно, перебираючи всі цифри коду, переходити від кореня до нащадка, при цьому вибирається лівий нащадок (ребро «0») або правий нащадок (ребро «1»), якщо наступна цифра у коді – 0 або 1 відповідно.

У даному дереві символ «p» матиме код «101».

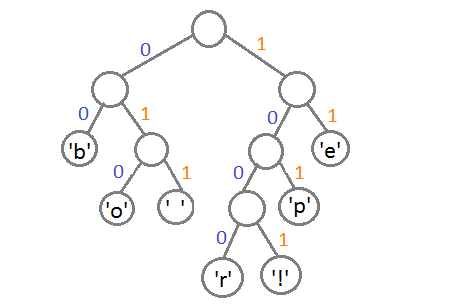


Рис. 1 – Дерево Хаффмана

# Вибір методу розв’язання

Ідея алгоритму така: знаючи ймовірності появи символів у повідомленні, можна описати процедуру побудови кодів змінної довжини, що складаються з цілої кількості бітів. Символам з більшою ймовірністю ставляться у відповідність коротші коди. Коди Хаффмана володіють властивістю префіксності (тобто, жодне кодове слово не є префіксом іншого), що дозволяє однозначно їх декодувати.

Класичний алгоритм Хаффмана на вході отримує таблицю частот, з якими зустрічаються символи у повідомленні (або ж цю таблицю можна обрахувати самостійно). Далі будується дерево кодування Хаффмана (Н-дерево).

Найпростішим варіантом реалізації буде визначення частот символів «на етапі компіляції», на основі відомих частот літер різних мов. Але, специфічні файли можуть складатися лише з декількох символів, і тоді алгоритм буде неефективним. Отже, доведеться рахувати частоти символів за допомогою проходу по всім символам.

Для того, щоб уникнути цього проходу, можна використовувати адаптивний алгоритм. Його суть полягає в тому, що при появі нового символу дерево буде «перераховуватися» на основі нової частоти. Але, оскільки саме кодування символів потребує вже побудованого оптимального дерева, цей підхід буде використовувати ті ж самі два проходи, і практичної користі від нього не буде ніякої.

Оскільки при обробці файлів можна собі дозволити зчитати неперервний потік символів двічі, оптимальним буде простий алгоритм Хаффмана з підрахунком частот. Запишемо порядок дій, необхідних для побудови Н-дерева частот:

1. Символи вхідного алфавіту утворюють список вільних вузлів. Кожен лист має вагу, яка може бути рівною або ймовірності, або кількості входжень символу у стиснене повідомлення.
2. Вибираються два вільних вузли дерева з найменшими вагами.
3. Створюється їхній батьківський вузол з вагою, рівною їх сумарній вазі.
4. Вузол-батько додається в список вільних вузлів, а два його нащадки видаляються з цього списку.
5. Одній дузі, котра виходить з вузла батька, ставиться у відповідність біт 1, інший — біт 0.
6. Кроки, починаючи з другого, повторюються доти, поки в списку вільних вузлів не залишиться тільки один вільний вузол. Він і буде вважатися коренем дерева.

# Алгоритм обраного методу

# C:\Users\Kostya\Desktop\main_8c_a0ddf1224851353fc92bfbff6f499fa97_cgraph.png

Рис. 2 – Алгоритм роботи програми.

Рис. 3 – UML­-діаграма викликів.

**Кінець**

**Запис файлу**

**Обробка меню**

**Читання файлу**

Декодування

Кодування

**Декодування**

**Кодування**

**Рекурсивне розпакування дерева**

**Рекурсивне пакування дерева**

**Який режим вибрано**

**Початок**

# Опис програми

1. Інтерфейс користувача

Оскільки єдина задача програми – кодувати та декодувати файли, причому ніяких суттєвих налаштувань бути не може (програма сама будує найоптимальніше дерево), то користувацьке меню має складатися лише з трьох команд – «кодувати», «декодувати» та «генерувати дерево».

Остання команда не проводить фактичного стиснення даних, а лише аналізує їх та будує дерево, виводячи його у форматі бібліотеки graphviz та візуалізуючи у форматі png-зображення.

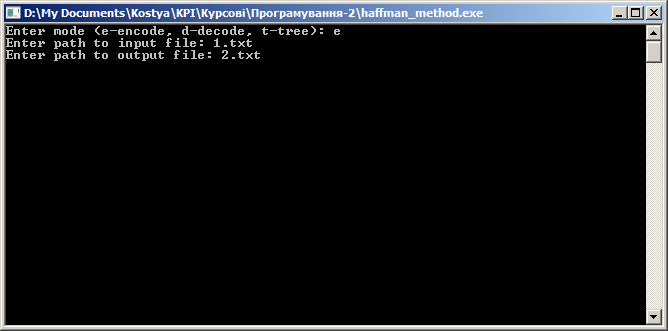
* Для автоматизації роботи можливе задання параметрів у командному рядку, при цьому допустимі три команди:

1. -encode INPUT OUTPUT
2. -decode INPUT OUTPUT
3. -treeN/treeS INPUT OUTPUT

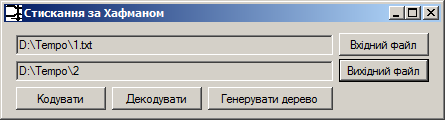
Тут можна відповідною командою вибрати режим зображення вершин –

за допомогою чисел або за допомогою ASCII – символів відповідно.

* Можливе також введення безпосередньо під час роботи програми:



* Для спрощення роботи було розроблене проста графічна програма-меню на мові C#, яка за допомогою вбудованих діалогів Windows Forms пропонує користувачу вибрати необхідні файли, а потім викликає безпосередню головну програму для стиснення та передає їй вибрані шляхи до файлів за допомогою параметрів командного рядка.



1. Інформація для програміста

|  |  |
| --- | --- |
| obj\_s | Структура для представлення вершини дерева.   * **freq** - частота появи символу, вага вершини. * **ind** - індекс відповідного символу або вузла дерева. * **next** - номер наступної вершини двозв'язного списку. * **prev** - номер попередньої вершини двозв'язного списку. |
| tree\_o | Структура для представлення вузла бінарного дерева.   * **zero** - індекс елемента, що лежить на лівій вітці і кодується нулем. * **one** - індекс елемента, що лежить на правій вітці і кодується одиницею. |
| main | Головна функція, реалізує взаємодію з користувачем та введення-виведення. |
| draw\_tree | Функція, що генерує дерево Хаффмана в форматі graphviz.   * **tree** - вказівник на структуру дерева. * **tl** - кількість вузлів дерева. * **freq** - вказівник на масив частот вершин. * **treem** - вид дерева. |
| encode | Функція, що будує H-дерево та кодує файл.   * **s** - вказівник на буфер з вихідним файлом. * **s\_s** - довжина вихідного файлу. * **r** - вказівник на буфер з результативним файлом. * **r\_s** - довжина результативного файлу. * **dt** – параметр необхідності візуалізації. * **treem** – вид дерева. |
| encode\_tree | Функція, що рекурсивно пакує дерево у файл.   * **buf** - вказівник на буфер розміру letter\_size для коду поточного символу. * **buf\_p** - позиція в буфері, параметр рекурсії. * **code** - результативний масив згенерованих кодів для подальшого кодування файлу. * **r** - вказівник на результативний масив, що представляє вміст закодованого файлу. * **r\_p** - вказівник у масиві r, що вказує на поточну позицію нового символу. * **tree** - вказівник на структуру дерева для пакування. * **ind** - номер вершини в дереві, параметр рекурсії. |
| decode | Функція, що зчитує Н-дерево та декодує файл.  **s** - вказівник на буфер з вихідним файлом.  **r** - вказівник на буфер з результативним файлом. |
| decode\_tree | Функція, що рекурсивно розпаковує дерево з файлу.  **s** - вказівник на буфер з вихідним файлом.  **s\_p** - вказівник на позицію в буфері, параметр рекурсії.  **tree** - вказівник на результативне дерево.  **tree\_p** - індекс вузла у дереві, параметр рекурсії.  **r\_s** - вказівник на номер останнього доданого вузла. |

Для покращення сприйняття вихідного коду програми він подається з пояснюючими коментарями, наприклад,

int c;

uchar obj=0; // Буфер для номера вершини/вузла

if(READ\_BIT(s,\*s\_p)==0) // Якщо далі - вершина

{

(\*s\_p)++;

for(c=0;c<8;c++)

{ WRITE\_BIT(&obj,c,READ\_BIT(s,\*s\_p)); s\_p[0]++; } // Зчитаємо її номер

tree[tree\_p].zero=obj; // Та помістимо її в ліву вітку поточного вузла

}

Для більшого розуміння структури програми код був оформлений з використанням коментарів формату бібліотеки doxygen, яка обробляє вихідний файл та генерує інтерактивну веб-сторінку, на якій доступна навігація по всім складовим програми (функції, об’єкти, визначення структур, макровизначення), детальний опис кожного елементу та UML-діаграми викликів.

Зокрема, рисунок 3 – UML-діаграма викликів був отриманий за допомогою бібліотеки doxygen.

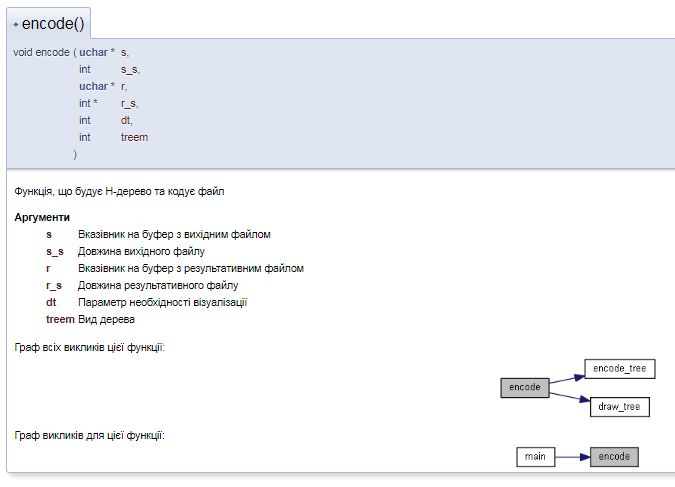
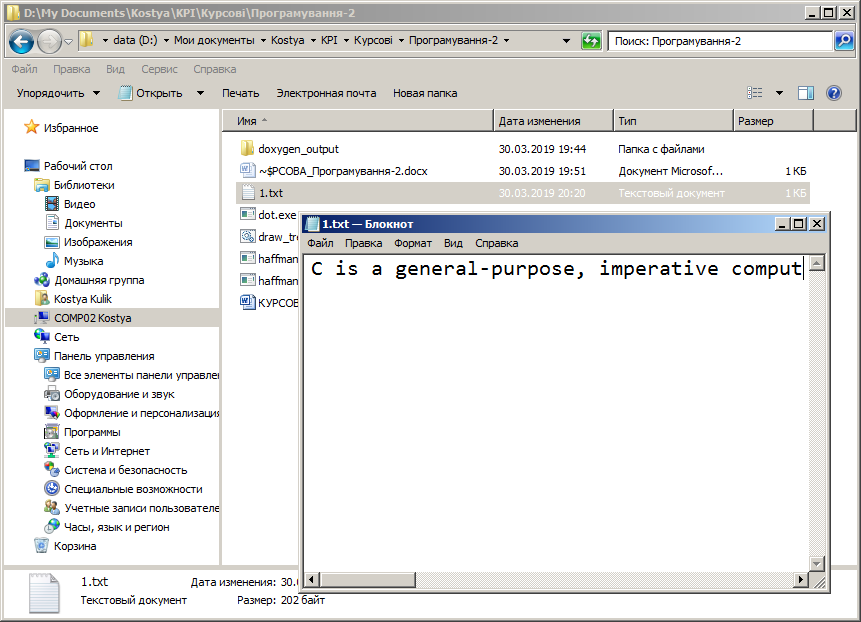


Рис. 4 – Згенерована документація.

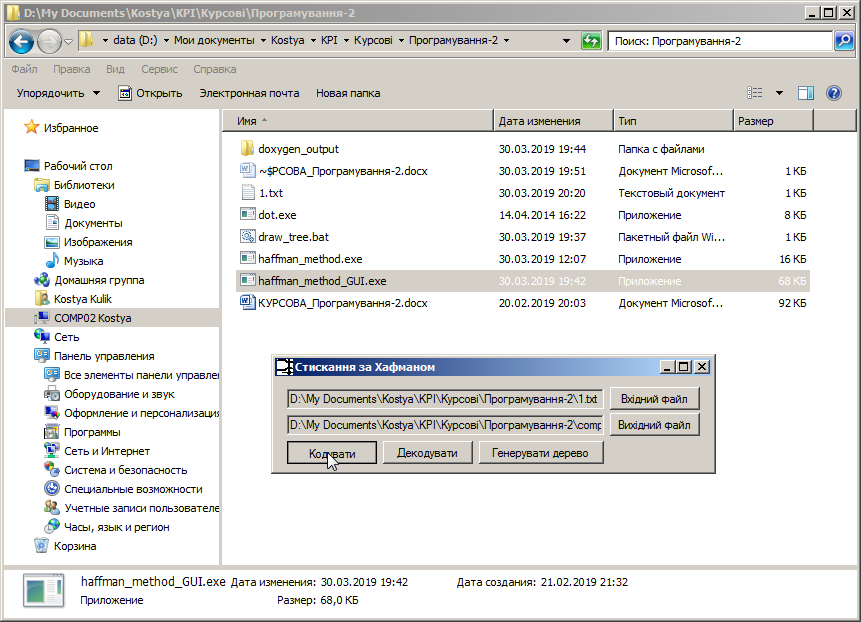
# Результати. Контрольний приклад

В якості контрольного прикладу надамо результати кодування та декодування простого текстового файлу.

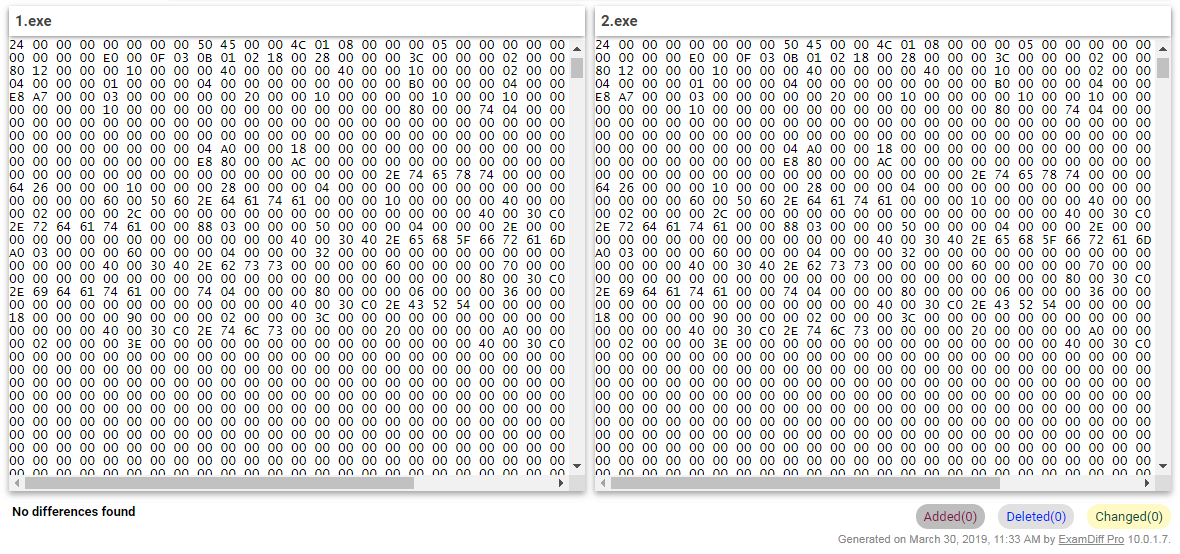
1. Наберемо та збережемо необхідний текст у файлі Text\_about\_C.txt



1. Викликаємо програму haffman\_method\_GUI.exe, за допомогою кнопки «Вхідний файл» виберемо Text\_about\_C.txt, у якості вихідного файлу оберемо compressed.bin та натиснемо кнопку «Кодувати».



1. У робочій теці з’явиться стиснений двійковий файл compressed.bin. Можна перевірити, що він не являється текстовим та не схожий зовнішньо на початковий файл.
2. Декодувавши отриманий файл compressed.bin, отримаємо початковий файл з точністю до біту.
3. Для перевірки на великих файлах спробуємо «пропустити» через програму який-небудь двійковий файл, наприклад, exe-файл Windows-програми.



Порівняння початкового та кінцевого файлів показує повну відповідність, тому, очевидно, результативний файл-програма запускається та нормально функціонує.

У таблиці нижче наведені вмісти тестових файлів у шістнадцятковому форматі. Можна помітити, що для цього файлу з великим показником надмірності алгоритм виявився ефективним, зменшивши розмір файлу з 202-х до 170-ти байтів, тобто, на 16%.

|  |  |
| --- | --- |
| C is a general-purpose, imperative computer programming language, supporting structured programming, lexical variable scope and recursion, while a static type system prevents many unintended operations. | 4320697320612067656E6572616C2D707572706F73652C20696D706572617469766520636F6D70757465722070726F6772616D6D696E67206C616E67756167652C20737570706F7274696E6720737472756374757265642070726F6772616D6D696E672C206C65786963616C207661726961626C652073636F706520616E6420726563757273696F6E2C207768696C6520612073746174696320747970652073797374656D2070726576656E7473206D616E7920756E696E74656E646564206F7065726174696F6E732E |
| КжYјзf3aГќ3vvћ"Сtт,М>C1Ms!џ6ќN@Ѓ$dВla«cєу€`ј’бБмBV'кдw‰·±З¦ИЦ)·XШ/!єs‰иJ›!Ўо‡^HпocЋ%MЇ#Ai&0ТXЗҐЌ@†`ЋЕ5'ЃЋјi—“M¤“@CC“  Ћ5Г@8Gx, dгёќRTЂSPjB8=€H”5ё | 000000CAE659BCE710663361C39D0D3376769E22D174F22CCC3E43314D0773219F369D4E4006812464C26C140F61AB63BAF38860BC92E1C1EC42560118270EEA01E47789B7B1C712A6C8D629B758D82F21BA7389E84A9B21A1EE98875E0D48EF136F638E254DAF2341692630D258C7A58D134086608E04C53527818EBC691297934DA493409843431093090B8E0435C3403847782C052064E314B89D52548053506A42383D88489435B8 |
| C is a general-purpose, imperative computer programming language, supporting structured programming, lexical variable scope and recursion, while a static type system prevents many unintended operations. | 4320697320612067656E6572616C2D707572706F73652C20696D706572617469766520636F6D70757465722070726F6772616D6D696E67206C616E67756167652C20737570706F7274696E6720737472756374757265642070726F6772616D6D696E672C206C65786963616C207661726961626C652073636F706520616E6420726563757273696F6E2C207768696C6520612073746174696320747970652073797374656D2070726576656E7473206D616E7920756E696E74656E646564206F7065726174696F6E732E |

У якості наочної ілюстрації дерева Хаффмана подамо згенероване алгоритмом зображення для розглянутого вище тексту.

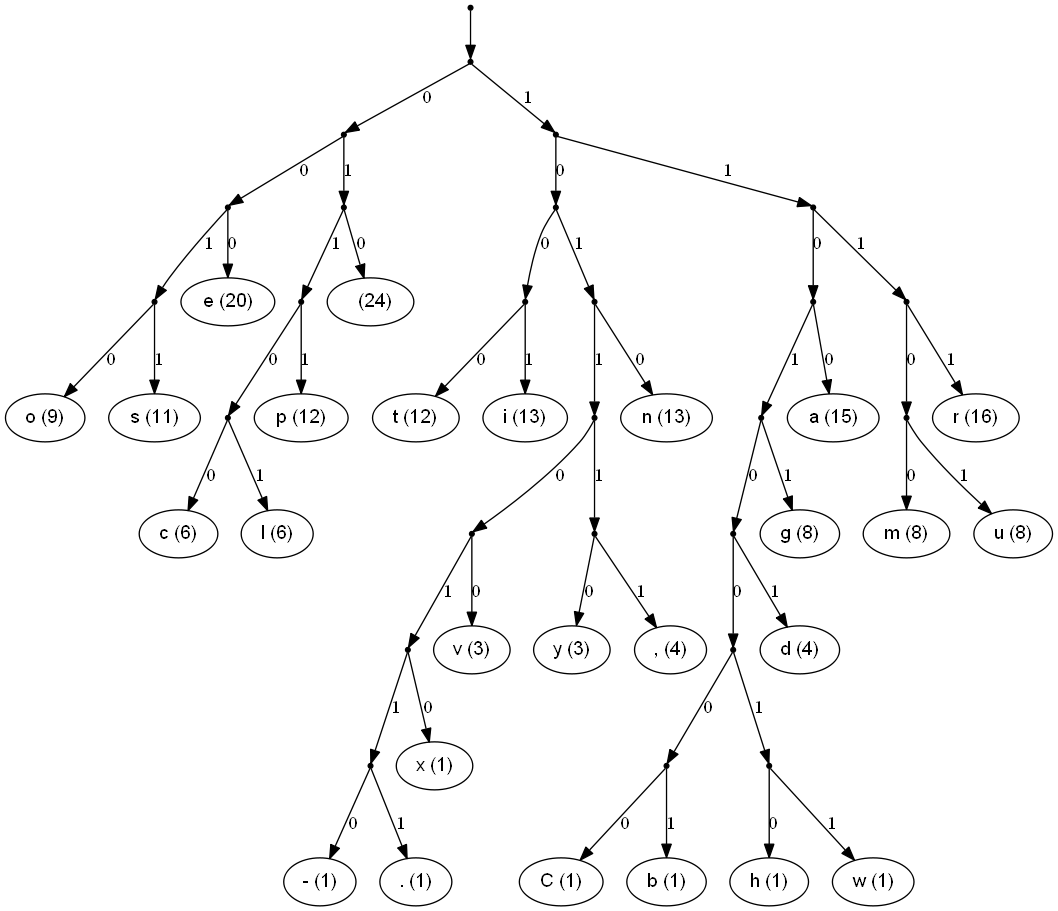
Можна помітити, що найбільш часто вживані символи (пробіл та латинська літера e) отримали найкоротші коди (010 та 000 відповідно), а найбільш рідкі (C, b, h, w, крапка, дефіс) – найдовші коди (по 8 двійкових розрядів).

Таке представлення дозволяє зменшити надлишковість текстових даних незалежно від взаємного розміщення символів, що в деяких випадках дозволить досягти навіть більшого стиснення, ніж у алгоритмів «зі словниками».

У випадку двійкових даних найчастіше надлишковість буде достатньо малою, тому високої ефективності алгоритм не дасть, проте у деяких випадках (наприклад, при стисненні растрових bitmap-зображень з однокольоровим фоном) можна досягти непоганих результатів. Наприклад, спробуємо стиснути зображення розміру 640х480 п., повністю заповнене зеленим кольором.

Можна помітити, що навіть для форматів зі стисненням алгоритм дозволяє зменшити розмір файлу принаймні вдвічі. Але, звісно, дуже сильно поступається сучасному високоефективному алгоритму архівації.

|  |  |  |  |
| --- | --- | --- | --- |
|  | BMP | JPG | PNG |
| Оригінальний файл | 900 КБ | 5.3 КБ | 2.24 КБ |
| Алгоритм Хаффмана | 337 КБ | 2.59 КБ | 1.06 КБ |
| WinRAR | 597 Б | 566 Б | 268 Б |



Таблиця 1 – Порівняння ефективності

Рис. 5 – Приклад дерева Хаффмана для простого текстового файлу з високою надмірністю.

.

# ВИСНОВКИ

Поставлену мету було досягнено, оскільки був реалізований бажаний алгоритм Хаффмана. Результати тестів показали, що алгоритм працює коректно, і декодований файл збігається з вихідним із точністю до байтів.

Для простих текстових файлів алгоритм показує непогані результати, особливо, якщо у таких файлах використані лише декілька символів з усього ASCII-діапазону. Наприклад, файл розміром 1 МБ лише з двох символів стискається до 256 МБ + 8 байтів для заголовку (запакованого дерева), тобто, у 8 разів, оскільки для запису кожного символу використовується вже не один байт ASCII, а лише один біт.

Для бінарних файлів без надмірності алгоритм показує погані результати, оскільки в двійкових даних майже немає надмірності, а, отже, всі «символи» використовуються майже з однаковою частотою. Але, в деяких окремих випадках все одно можна домогтися стиснення у 2-3 рази.

Програму можна використовувати як для наочної демонстрації методу Хаффмана, так і для реального стиснення великих файлів з високою надмірністю.

Серед шляхів подальшого покращення конкретного алгоритму можна назвати реалізацію адаптивної версії алгоритму (див. розділ «Вибір методу розв’язання»), реалізацію алгоритму зі змінним розміром «символу» (наприклад, двобайтові символи UNICODE) та введення механізму масштабування коефіцієнтів для уникнення переповнень лічильників.

Якщо ж заглиблюватися в область стиснення даних, то можна було б додатково реалізувати алгоритми арифметичного кодування та перетворення Барроуза-Вілера.

# Література

1. Бакнелл, Д. М. Фундаментальные алгоритмы и структуры данных в Delphi / Д. М. Бакнелл. – Питер: ДиаСофтЮП, 2003. – 351 с.
2. Димитренко Е.В., Швецов А.Ю. ИСПОЛЬЗОВАНИЕ БИНАРНЫХ ДЕРЕВЬЕВ В МЕТОДЕ КОДИРОВАНИЯ ХАФФМАНА // Научное сообщество студентов XXI столетия. ТЕХНИЧЕСКИЕ НАУКИ: сб. ст. по мат. XXXI междунар. студ. науч.-практ. конф. № 4(30). – 266 с.

# Додаток А. Фрагмент тексту програми на мові С

void encode\_tree(uchar\* buf, int buf\_p, uchar code[alphabet\_size][letter\_size+2], uchar\* r, int\* r\_p, tree\_o\* tree, int ind)

{

int c;

buf[buf\_p]='0'; *// Додаємо до коду нуль*

if(tree[ind].zero<(int)alphabet\_size+1) *// Якщо у лівій вітці лежить вершина*

{

for(c=0;c<=buf\_p;c++) code[tree[ind].zero][c]=buf[c]; *// Збережемо її код*

code[tree[ind].zero][c]='**\0**';

WRITE\_BIT(r, \*r\_p, 0); r\_p[0]++; *// Запишемо у файл нуль - ознаку вершини*

for(c=0;c<8;c++)

{ WRITE\_BIT(r, \*r\_p, READ\_BIT(&tree[ind].zero, c)); r\_p[0]++; } *// Запишемо у файл індекс вершини*

}

else

{

WRITE\_BIT(r, \*r\_p, 1); r\_p[0]++; *// Якщо ж це - вузол дерева*

encode\_tree(buf, buf\_p+1, code, r, r\_p, tree, *// Рекурсивно запакуємо його*

tree[ind].zero-(alphabet\_size+1));

}

buf[buf\_p]='1'; *// Додаємо до коду одиницю*

if(tree[ind].one<(int)alphabet\_size+1) *// Якщо у правій вітці лежить вершина*

{

for(c=0;c<=buf\_p;c++) code[tree[ind].one][c]=buf[c]; *// Збережемо її код*

code[tree[ind].one][c]='**\0**';

WRITE\_BIT(r, \*r\_p, 0); r\_p[0]++; *// Запишемо у файл нуль - ознаку вершини*

for(c=0;c<8;c++)

{ WRITE\_BIT(r, \*r\_p, READ\_BIT(&tree[ind].one, c)); r\_p[0]++; } *// Запишемо у файл індекс вершини*

}

else

{

WRITE\_BIT(r, \*r\_p, 1); r\_p[0]++; *// Якщо ж це - вузол дерева*

encode\_tree(buf, buf\_p+1, code, r, r\_p, tree, *// Збережемо його*

tree[ind].one-(alphabet\_size+1));

}

}

void encode(uchar\* s, int s\_s, uchar\* r, int\* r\_s, int dt, int treem)

{

int c, cc, mp1, mp2, cf\_ptr, tree\_l=-1;

obj\_s obj[alphabet\_size+2]; *// Масив вершин*

int dt\_pfreq[alphabet\_size]; *// Масив частот для візуалізації*

tree\_o tree[alphabet\_size]; *// Масив вузлів дерева*

uchar code\_buf[alphabet\_size]; *// Масив-буфер для функції encode\_tree*

uchar code[alphabet\_size][letter\_size+2]; *// Масив кодів*

memset(obj, 0, sizeof(obj\_s)\*(alphabet\_size+2)); *// Ініціалізуємо масив вершин нулями*

obj[0].freq=-1; *// Частота першого та останного символів-*

obj[alphabet\_size+1].freq=-1; *// заглушок буде максимальною*

obj\_s\* \_obj=&(obj[1]); *// Додатковий вказівник для зручності індексації*

for(c=0;c<s\_s;c++) \_obj[s[c]].freq++; *// Рахуємо частоти вихідних символів*

if(dt) for(c=0;c<(int)alphabet\_size;c++) dt\_pfreq[c]=\_obj[c].freq;

for(c=-1;c<(int)alphabet\_size;c=cc)

{

\_obj[c].ind=c; *// Ініціалізуємо індекси вершин*

for(cc=c+1;cc<(int)alphabet\_size+1;cc++)

if(\_obj[cc].freq>0) { \_obj[c].next=cc; \_obj[cc].prev=c; **break**; } *// та вказівники двозв'язного списку*

}

while(\_obj[obj[0].next].next>0) *// Поки не закінчаться вершини*

{

mp2=obj[0].next;

mp1=\_obj[mp2].next;

cf\_ptr = \_obj[mp1].next; *// Визначимо вершини з мінімальною та наступною*

if(\_obj[mp1].freq>\_obj[mp2].freq) { c=mp2; mp2=mp1; mp1=c; } *// мінімальною частотами серед двох перших*

while(cf\_ptr>0) *// Рухаючись двозв'язним списком*

{

if(\_obj[cf\_ptr].freq<\_obj[mp1].freq) { mp2=mp1; mp1=cf\_ptr; } *// визначимо вершини з мінімальною та наступною*

else if(\_obj[cf\_ptr].freq<\_obj[mp2].freq) mp2=cf\_ptr; *// мінімальною частотами серед усіх*

cf\_ptr=\_obj[cf\_ptr].next;

}

\_obj[mp1].freq+=\_obj[mp2].freq; *// Об'єднаємо mp1 та mp2 в mp1 - додамо частоти*

\_obj[\_obj[mp2].prev].next=\_obj[mp2].next; *// Видалимо mp2 зі списку*

\_obj[\_obj[mp2].next].prev=\_obj[mp2].prev;

tree[++tree\_l].zero=\_obj[mp1].ind; *// Створимо новий вузол дерева з mp1 на лівій*

tree[tree\_l].one=\_obj[mp2].ind; *// вітці та mp2 на правій*

\_obj[mp1].ind=alphabet\_size+1+tree\_l; *// Налаштуємо індекс на індекс вузла*

}

cf\_ptr=32; *// Пропустимо у файлі 4 байти*

encode\_tree(code\_buf, 0, code, r, &cf\_ptr, tree, tree\_l); *// Рекурсивно запакуємо заголовок-дерево у файл*

if(dt) { draw\_tree(tree, tree\_l, (int\*)&dt\_pfreq, treem); return; } *// За потреби викличемо функцію візуалізації*

for(c=0;c<s\_s;c++) *// Для кожного символу у вихідному файлі*

for(cc=0;code[s[c]][cc]!='**\0**';cc++) *// Закодуємо його за допомогою коду з таблиці*

{ WRITE\_BIT(r, cf\_ptr, code[s[c]][cc]=='1'); cf\_ptr++; }

PACK\_INT(s\_s,r); *// Запишему розмір файлу у зарезервоване місце*

\*r\_s=cf\_ptr;

}

void decode\_tree(uchar\* s, int\* s\_p, tree\_o\* tree, int tree\_p, int\* tree\_l)

{

int c;

uchar obj=0; *// Буфер для номера вершини/вузла*

if(READ\_BIT(s,\*s\_p)==0) *// Якщо далі - вершина*

{

(\*s\_p)++;

for(c=0;c<8;c++)

{ WRITE\_BIT(&obj,c,READ\_BIT(s,\*s\_p)); s\_p[0]++; } *// Зчитаємо її номер*

tree[tree\_p].zero=obj; *// Та помістимо її в ліву вітку поточного вузла*

}

else *// Якщо це - вузол*

{

(\*s\_p)++;

tree[tree\_p].zero=(alphabet\_size+1)+(++(\*tree\_l)); *// Додамо його в ліву вітку поточного вузла*

decode\_tree(s, s\_p, tree, \*tree\_l, tree\_l); *// та рекурсивно розпакуємо*

}

if(READ\_BIT(s,\*s\_p)==0) *// Якщо далі - вершина*

{

(\*s\_p)++;

for(c=0;c<8;c++)

{ WRITE\_BIT(&obj,c,READ\_BIT(s,\*s\_p)); s\_p[0]++; } *// Зчитаємо її номер*

tree[tree\_p].one=obj; *// Та помістимо її в праву вітку поточного вузла*

}

else *// Якщо це - вузол*

{

(\*s\_p)++;

tree[tree\_p].one=(alphabet\_size+1)+(++(\*tree\_l)); *// Додамо його в праву вітку поточного вузла*

decode\_tree(s, s\_p, tree, \*tree\_l, tree\_l); *// та рекурсивно розпакуємо*

}

}

void decode(uchar\* s, uchar\* r)

{

int s\_p=32,c,c\_s,tree\_l=0;

tree\_o tree[alphabet\_size+2]; *// Масив вузлів дерева*

UNPACK\_INT(s,c\_s); *// Розпаковуємо довжину результативного файлу*

decode\_tree(s, &s\_p, tree, 0, &tree\_l); *// Рекурсивно розпаковуємо дерево*

for(c=0;c<c\_s;c++) *// Для кожного символу у вихідному файлі*

{

tree\_l = 0;

do *// Розшифруємо код*

{

if(tree\_l>=(int)(alphabet\_size+1)) tree\_l-=(alphabet\_size+1);

if(READ\_BIT(s,s\_p)==0) tree\_l=tree[tree\_l].zero; *// Якщо в коді нуль - переходимо по лівій вітці*

else tree\_l=tree[tree\_l].one; *// Якщо одиниця, то по правій*

s\_p++;

} while(tree\_l>=(int)(alphabet\_size+1)); *// Поки не досягнемо вершини*

r[c]=(uchar)tree\_l; *// Запишемо отриманий символ у результативний файл*

}

}