МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Тернопільський національний економічний університет
Навчально-науковий інститут інноваційних освітніх технологій
Кафедра комп'ютерної інженерії

БУРДА Олександр Володимирович

Алгоритми арифметичного кодування з низькою надлишковістю для стиснення цифрової інформації / Algorithms of Arithmetic Coding with Low Redundancy for Digital Information Compression

спеціальність: 123 - Комп'ютерна інженерія
магістерська програма - Комп'ютерна інженерія

Магістерська робота

Виконав студент групи КІзм-21
О. В. Бурда

Науковий керівник:
к.ф.-м.н., доцент, М. М. Касянчук

Магістерську роботу допущено
до захисту:

"31" 01 2018 р.

Завідувач кафедри
О. М. Березький

ТЕРНОПІЛЬ - 2018

Тернопільський національний економічний університет
Навчально-науковий інститут інноваційних освітніх технологій
Факультет комп'ютерних інформаційних технологій
Кафедра комп'ютерної інженерії
Освітній ступінь «магістр»
спеціальність: 123 - Комп'ютерна інженерія
магістерська програма - Комп'ютерна інженерія

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ О.М. Березький
"21" 11 2016 р.

ЗАВДАННЯ
НА МАГІСТЕРСЬКУ РОБОТУ СТУДЕНТУ

Бурді Олександру Володимировичу
(прізвище, ім'я, по батькові)

1. Тема магістерської роботи «Алгоритми арифметичного кодування з низькою надлишковістю для стиснення цифрової інформації / Algorithms of Arithmetic Coding with Low Redundancy for Digital Information Compression» керівник роботи к.ф.-м.н., доц. М.М. Касянчук
затверджені наказом по університету від 17 листопада 2016 р. № 669.

2. Строк подання студентом роботи «15» січня 2018 року

3. Вихідні дані до магістерської роботи

Об'єкт дослідження – процес кодування даних на основі алгоритмів арифметичного кодування.

Предмет дослідження – алгоритми арифметичного кодування інформації.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

- аналіз методів та засобів стиснення цифрової інформації;
- огляд та аналіз сучасних алгоритмів арифметичного кодування з низькою надлишковістю;
- розробка алгоритму арифметичного кодування на основі перетворення вхідного потоку в число з плаваючою точкою;
- обгрунтування вибору інструментальних засобів для реалізації алгоритму;
- програмна реалізація запропонованого алгоритму арифметичного кодування;
- порівняльний аналіз алгоритмів арифметичного кодування;
- експериментальне дослідження ефективності стиску..

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень):

- мета роботи, об'єкт та задачі дослідження
- схема арифметичного кодування;
- межі кодування символів;
- блок-схема розробленого алгоритму арифметичного кодування;
- головне вікно програми для арифметичного кодування;
- визначення середнього числа біт;
- кроки кодування адаптивним алгоритмом ABBACD;
- кроки декодування адаптивним алгоритмом;
- відповідність кодових слів довжинам ділянок;
- порівняння арифметичного кодування та Хаффмана.

6. Консультанти розділів магістерської роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв
Антиплагіат	Мельник Г.М., доцент		
Нормо-контроль	Гураль І. В., викладач		

7. Дата видачі завдання «21» листопада 2016 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів магістерської роботи	Строк виконання етапів магістерської роботи	Приміт
1	Аналіз алгоритмів стиснення інформації та суті арифметичного кодування	18.11.2016 – 1.01.2017	Виконано
2	Алгоритми арифметичного кодування з низькою надлишковістю	2.01.2017 – 31.05.2017	Виконано
3	Програмна реалізація та порівняльний аналіз запропонованого алгоритму арифметичного кодування	1.06.2017 – 22.12.2017	Виконано
4	Нормоконтроль, попередній захист	15.01.2018 – 26.01.2018	Виконано
5	Захист	2.02.2018	

Студент _____ (підпис) Бурда О.В.

Керівник магістерської роботи _____ (підпис) к.ф.-м.н., доцент М.М. Касянчук

ЗМІСТ

ВСТУП

Актуальність роботи. Сучасне суспільство використовує цифровий вид подання інформації в багатьох сферах життєдіяльності [1]. Великий обсяг інформації вимагає великої протяжності та пропускної здатності каналів передачі даних [2]. На даний момент розвитку інформаційної інфраструктури існуючі канали не завжди справляються з необхідним трафіком. Отже, завдання стиснення даних є актуальною в багатьох додатках обробки і передачі інформації [3-6].

Стисненням блоку даних (компресією, архівуванням) називається такий його опис, при якому створюється стиснутий блок містить менше бітів, ніж вихідний, але по ньому можливе однозначне відновлення кожного біта вихідного блоку [7]. Зворотний процес відновлення за описом називається декомпресією або розархівуванням. Відповідно, крім цього, використовуються ще і такі пари термінів: компресія/декомпресія, кодування/декодування, упакування/розпакування [8].

Більшість методів компресії різних типів цифрової інформації часто використовують на певних стадіях алгоритми стиснення без втрат. Це таке кодування, при якому ентропія стиснутих даних збігається з ентропією вихідного джерела і по стисненим даним можна повністю відновити вихідну інформацію [9]. Можна сказати, що компресія без втрат є екстремальним випадком стиснення, при якому ентропія даних залишається незмінною [10].

В основі всіх методів стиснення лежить проста ідея: якщо представляти елементи, які часто використовуються, короткими кодами, а ті, що рідко використовуються - довгими, то для зберігання блоку даних буде потрібен менший обсяг пам'яті, ніж якби всі елементи представлялися кодами однакової довжини [11].

Ефективність стиснення враховує ступінь стиснення (відношення довжини нестиснутих даних до довжини відповідних їм стиснутих) і швидкість стиснення та відновлення [12]. Часто користуються оберненою до ступеня стиснення величиною - коефіцієнтом стиснення, який визначається як відношення довжини стиснутих даних до довжини відповідних їм нетиснутих [13].

Отже, дана робота присвячена розробці нового алгоритму та його порівняльному аналізу з існуючими методами арифметичного кодування інформації і порівнянні їх коефіцієнтів стиснення.

Мета роботи. Метою даної роботи є розробка алгоритму арифметичного кодування інформації з низькою надлишковістю для стиснення цифрової інформації та порівняння його параметрів з існуючими алгоритмами.

Досягнення поставленої мети включало розв’язання таких взаємопов’язаних **завдань**:

- аналіз методів та засобів стиснення цифрової інформації;
- огляд та аналіз сучасних алгоритмів арифметичного кодування з низькою надлишковістю;
- розробка алгоритму арифметичного кодування на основі перетворення вхідного потоку в число з плаваючою точкою;
- обгрунтування вибору інструментальних засобів для реалізації алгоритму;
- програмна реалізація запропонованого алгоритму арифметичного кодування;
- порівняльний аналіз алгоритмів арифметичного кодування;
- експериментальне дослідження ефективності стиску.

Об'єктом дослідження є арифметичне кодування інформації.

Предметом дослідження є процес кодування даних на основі алгоритмів арифметичного кодування.

Методи дослідження. В основу наукових досліджень покладено методи на основі теорії інформації, теорії цифрових автоматів, теорії кодування, теорії графів, комп'ютерного моделювання алгоритмів арифметичного кодування.

Наукова новизна одержаних результатів визначається наступним:

– удосконалено формалізований опис арифметичних кодів, що дозволило обгрунтувати вибір програмного забезпечення для реалізації алгоритмів;

– побудовано математичну модель алгоритму арифметичного кодування на основі перетворення вхідного потоку в число з плаваючою точкою;

– розроблено адаптивний алгоритми арифметичного кодування, що дозволило реалізувати кодування довжин повторень у вхідному потоці інформації.

Практична цінність одержаних результатів полягає в тому, що:

– обґрунтовано підхід та розроблено алгоритм арифметичного кодування, що дозволило застосувати відповідне програмне забезпечення для реалізації запропонованого алгоритму;

– реалізовано програмне забезпечення системи арифметичного кодування на основі середовища розробки Borland Delphi7.

Публікації та апробація ДР. За результатами наукових досліджень, проведених у магістерській роботі, підготовлено тези доповіді «Алгоритми оптимального арифметичного кодування для стиснення цифрової інформації» обсягом 1 сторінка на Всеукраїнській школі-семінарі «Сучасні комп'ютерні інформаційні технології» (ACIT-2017) [14].

1 АНАЛІЗ АЛГОРИТМІВ СТИСНЕННЯ ІНФОРМАЦІЇ ТА СУТІ АРИФМЕТИЧНОГО КОДУВАННЯ

1.1 Алгоритми стиснення цифрової інформації

Стисненням кінцевого обсягу цифрової інформації (блоку) називається такий його опис, при якому створений стиснутий блок містить менше бітів, ніж вихідний, але по ньому однозначно можна відновити кожен біт вихідного блоку. Так зване стиснення з втратами (lossy compression) - це два різні процеси: виділення збереженої частини з блоків інформації - створення моделі, яка залежить від мети стиснення і по можливості поліпшує ступінь подальшого стискання, - і власне стиснення (lossless compression) [15].

Існуючі алгоритми стиснення даних можна розділити на два великі класи - з втратами і без втрат. Алгоритми з втратами зазвичай застосовуються для стиснення зображень і аудіо. Ці алгоритми дозволяють досягти високих ступенів стиснення завдяки вибірковій втраті якості. Однак, за визначенням, відновити початкові дані з стисненого результату неможливо [16].

Алгоритми стиснення без втрат застосовуються для зменшення розміру даних і працюють таким чином, що можливо відновити дані в точності такими, які вони були до стиснення. Вони застосовуються в комунікаціях, архіваторах і деяких алгоритмах стиску аудіо- та графічної інформації [17-18].

Основний принцип алгоритмів стиснення базується на тому, що в будь-якому файлі, що містить невипадкові дані, інформація частково повторюється. Використовуючи статистичні математичні моделі, можна визначити ймовірність повторення певної комбінації символів. Після цього можна створити коди, що позначають обрані фрази, і присвоїти фразам, які найчастіше повторюються, найкоротші коди. Для цього використовуються різні техніки, наприклад ентропійне кодування, кодування повторів і стиснення за допомогою словника. З їх допомогою 8-бітний символ або цілий рядок, можуть бути

замінені лише кількома бітами, усуваючи таким чином зайву інформацію (рисунок 1.1).

Рисунок 1.1 – Блок-схема стиснення та відновлення даних

З незапам'ятних часів, коли 8-бітових процесорів було більше, ніж 16-бітних, а в UNICODE не було необхідності, символом найчастіше називали байт, а байтом - послідовність восьми бітів. Послідовність двох байтів комп'ютерний світ називає словом (Word), а чотирьох - подвійним словом (Dword). Однак дуже багато процесорів влаштовані так, що в кожному адресному осередку пам'яті лежить 16-, 24- або 32-бітний байт, а зовсім не 8-бітний, як у всіх IBM-сумісних комп’ютерних системах. Якщо стиснутий файл довший вихідного - то це є перекодування, але не стиснення за визначенням, всупереч роботі багатьох стискаючих програм (упакувальники, компресори, архіватори), в тому числі RAR32 для DOS і OS/2. Якщо стиснення з втратами не містить функцій дійсного стиснення, то це видалення інформації, несуттєвою для заданої мети, в рамках прийнятої моделі. Втім, визначень понять «стиснення», а, особливо, «код» існує стільки, що виникає дуже багато непорозумінь.

Тепер, з такими чіткими визначеннями, здається досить очевидним наступне [19]:

1) кожен з методів стиснення створений або для стиснення блоків бітів,

або блоків байтів, або блоків слів, оскільки це дуже різні об'єкти. Можна створити чіткі математичні визначення для цих трьох базових видів цифрової інформації, але ясно і без формул, що стискаючи заданий блок, метод стиснення враховує три аспекти: ймовірності бітів різні і залежать від їх значення, ймовірності появи байтів різні, імовірності різних слів неоднакові;

2) заданий блок стиснеться краще, якщо відома його структура:

а) це блок слів з *N*-бітних байтів;

б) це блок *N*-бітних байтів;

в) це блок бітів.

Вважається також відомим, за якими закономірностями блок був створений, якої довжини були записи-байти, як вони формувалися. Інакше кажучи, чим точніше процес стиснення враховує процес створення блоку, тим краще стиснення. А створювані блоки належать, як правило, одному з цих трьох базових видів: біти, байти, слова. Можна додати і ще два «граничних» виду: нульовий - хаос (неможливо виявити ніяких закономірностей - ні для бітів, ні для байтів, ні для слів) і тексти - послідовності слів, які з'являються по деякому правилу. Наприклад, блок містить російські речення, англійські і тексти функцій на мові С. Інші види (наприклад, суміш хаосу і байтів або бітів і слів) зустрічаються значно рідше;

3) у кожному з трьох випадків можливі два варіанти:

а) ймовірності елементів (бітів, байтів, слів) різні і безпосередньо не залежать від попередніх (контексту);

б) ймовірності елементів (бітів, байтів, слів) різні і сильно залежать від попередніх.

Проміжні випадки настільки рідкісні, що ними можна знехтувати. Видно, що випадок, коли ймовірності байтів різні і залежать від контексту, ідентичний випадку, коли ймовірності слів різні і не залежать від контексту, а ось інші - не ідентичні. Пов'язано це з тим, що за визначенням кінцева послідовність байтів -

це слово, а бітів - код, а не байт. Інакше кажучи, байт - це запис фіксованої довжини, а слово – довільної;

4) для спрощення структури даних визначаються такі рекомендації:

а) стискаючи блоки слів, краще створювати блоки байтів і/або бітів;

б) стискаючи блоки байтів, створювати тільки блоки бітів;

в) стискаючи блоки бітів, утворювати кінцевий стиснутий блок у вигляді нестисливого «хаосу».

З різних причин більшість сучасних програм-компресорів просто пропускає третій етап (стиснення блоків бітів), або «склеюють» його з іншим (стиснення блоків байтів), починаючи процес з припущення, що блок слів заданий. І тільки архіватори з опціями для мультимедійного стиснення перевіряють, чи це не блок байтів. Тому прийнято говорити в термінах «первинного» і «вторинного» стиснення: спочатку - моделювання або сортування (тасування байтів), потім - encoding (дотискання) байтів в коди. Наприклад, всі перші архіватори (ARC, ARJ, LHA, PAK, ZIP тощо) використовували LZ77 або LZ78 з подальшим дотискання методом Хаффмена або Шеннона [20].

Що ж це за причини? По-перше, більшість природних даних, що з'являються в комп'ютерах з пристроїв введення (клавіатура, сканер, мікрофон, відеокамера), - це дійсно блоки слів (в загальному випадку не 8-бітних байтів: сучасні графічні пристрої оперують 24- або 32- бітними байтами - «пікселями», а аудіо - 8-, 16- або 32-бітними «семплами»).

Відповідно і код, який виконується процесором (файли .exe, .dll тощо) - блоки слів. Блоки байтів і бітів з'являються, як правило, лише при комп'ютерній обробці цих природних даних. Таким чином, основне завдання стискаючого алгоритму - з'ясувати, як були побудовані слова блоку, за якими правилами-закономірностям. Саме первинний алгоритм грає вирішальну роль в досягненні прийнятної якості стиснення. По-друге, комп'ютерові легше оперувати з байтами, ніж з бітами. Кожен байт має свою адресу в пам'яті, кожна адреса

вказує на один байт (8-бітний, якщо це IBM-сумісний комп'ютер). Всього лише 20 років тому з'явилися перші 16-бітові процесори, здатні обробляти більше, ніж 8 біт однією інструкцією. По-третє, відомих алгоритмів для стиснення блоків бітів практично немає (за першими двома причинами). Прекрасні результати дають правильні первинні і вторинні алгоритми стиснення, а третій крок займає багато часу, але мало покращує якість стиснення.

Близько 20 років тому, коли розроблявся «універсальний» формат .zip, майже всі тексти були блоками 8-бітних ASCII-символів, виконувані модулі - для 16- або 8-бітових процесорів, а більшість мультимедійних файлів - графічних і звукових - блоки 8- або 4-бітних байтів. Тому цілком достатньо було мати один алгоритм для всіх даних: і слів, і 8-бітних байтів. З тих пір було створено багато спеціальних алгоритмів для конкретних типів даних, але мало універсальних програм, що дають відповідні результати [21].

Що можна сказати про статичне і динамічне кодування? У ті «доісторичні» часи, коли комп'ютери були великими, а файли - маленькими, проблеми виділення в заданому файлі логічно різних фрагментів не виникало. По-перше, відносно менше було файлів з різнорідними даними, та й самих типів даних було менше: розмірність байтів рідко досягала навіть 24 і тим більше 32 або 64 (4-канальний звук з 16-бітовою якістю). По-друге, самі логічно різні підфрагменти були настільки малі, що витрати на опис меж підфрагментів можна було порівняти з виграшем в разі їх використання. На запис нового бінарного дерева (методи Хаффмена або Шеннона-Фено, що використовують 8-бітові байти [22]) потрібно близько 100 байтів, до того ж складність і час роботи методу можуть збільшитися майже вдвічі - тому звертати увагу на фрагменти, менші кілобайта, немає сенсу. Тим більше, якщо середній розмір файлу - 100 Кб, а ймовірність того, що записи в ньому абсолютно різні, - менше відсотка. Зараз з цим інакше: і середній розмір в десяток разів більший, і ймовірність різнорідності фрагментів файлу - приблизно в стільки ж. Теоретично будь-який алгоритм стиснення можна

зробити як «зовсім статичним» (всі параметри жорстко задані спочатку), так і «зовсім динамічним» (всі параметри періодично переобчислюються).

Хоча стиснення даних набуло широкого поширення разом з Інтернетом і після винаходу алгоритмів Лемпелем і Зівом (алгоритми LZ), можна навести кілька більш ранніх прикладів стиснення. Морзе, відкриваючи свій код у 1838 році, розумно встановив відповідність буквам, які найчастіше використовуються в англійській мові, "e" і "t", найкоротші послідовності (точка і тире відповідно). Незабаром після виникнення мейнфреймів в 1949 році був придуманий алгоритм Шеннона - Фано, який призначав символам у блоці даних коди, грунтуючись на ймовірності їх появи в блоці. Імовірність появи символу в блоці була обернено пропорційна довжині коду, що дозволяло стиснути представлення даних.

На рисунку 1.2 представлена ієрархія алгоритмів стиснення [23]. Найпростіший в реалізації є алгоритм RLE (від англійського Run Length Encoding - групове кодування) - один з найстаріших і найпростіших алгоритмів архівації графіки. Зображення в ньому (як і в кількох алгоритмах, описаних нижче) витягується в ланцюжок байтів по рядках растра. Саме стиснення в RLE відбувається за рахунок того, що в оригінальному документі зустрічаються ланцюжки однакових байт. Заміна їх на пари <лічильник повторень, значення> зменшує надмірність даних. В даному алгоритмі ознакою лічильника (counter) служать одиниці в двох верхніх бітах зчитаного файлу, що представлено на рисунку 1.3. Відповідно 6 біт, що залишилися, витрачаються на лічильник, який може приймати значення від 1 до 64. Рядок із 64 повторюваних байтів перетворюється в два байта, тобто стискається в 32 рази.

Алгоритм розрахований на ділову графіку - зображення з великими областями кольору, що повторюється. Ситуація, коли файл збільшується, для цього простого алгоритму не така вже і рідкісна. Її можна легко отримати, застосовуючи групове кодування до оброблених кольорових фотографій. Для того, щоб збільшити зображення в два рази, алгоритм треба застосувати до

зображення, в якому значення всіх пікселів більші двійкового 11000000 і поспіль попарно не повторюються.

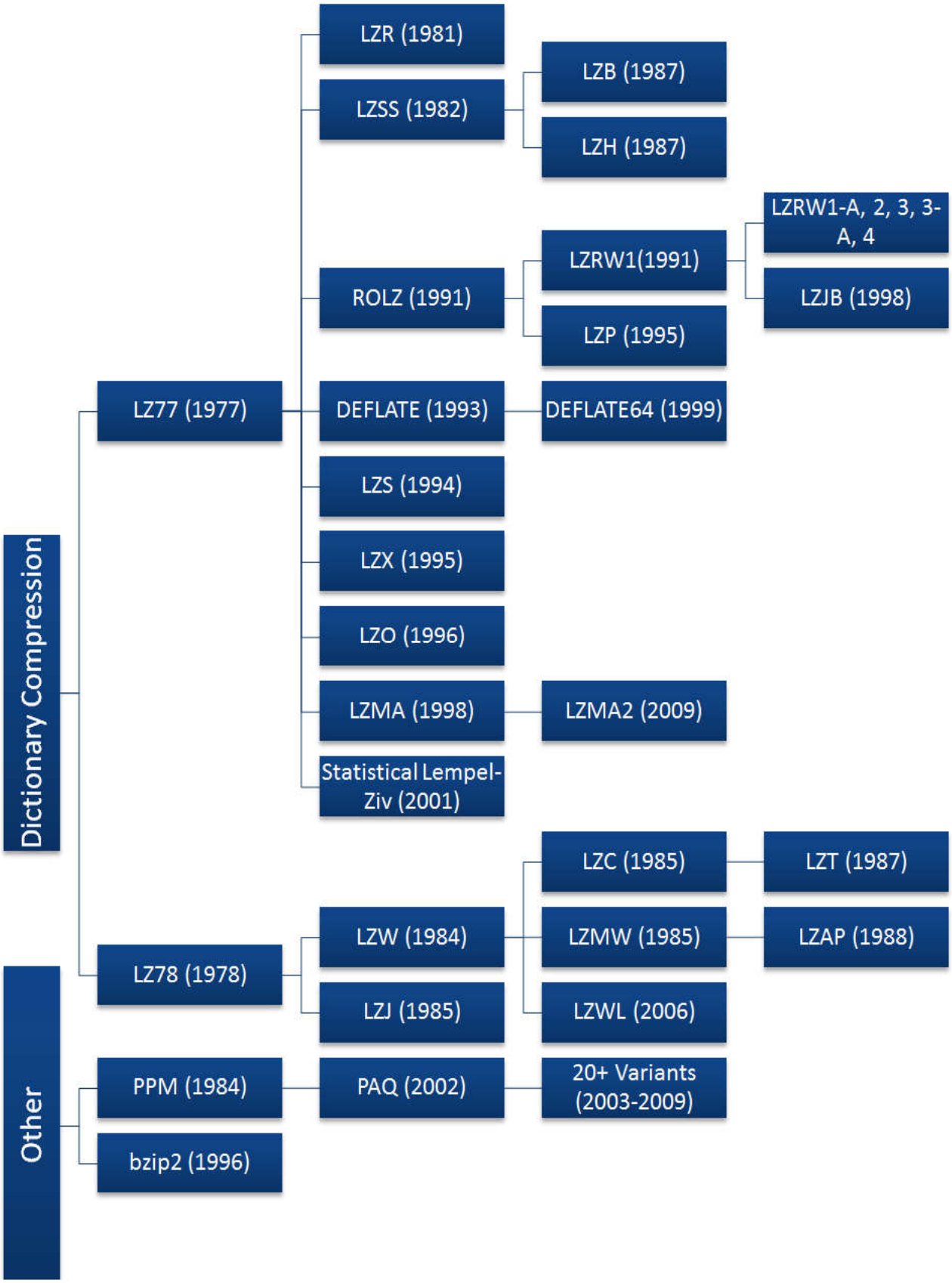

Рисунок 1.2 - Ієрархія алгоритмів стиснення

Рисунок 1.3 – Спрощена схема алгоритму RLE

Другий варіант цього алгоритму має більший максимальний коефіцієнт архівації та менше збільшує в розмірах вихідний файл.

Ознакою повтору в даному алгоритмі є одиниця в старшому розряді відповідного байта (рисунок 1.4).

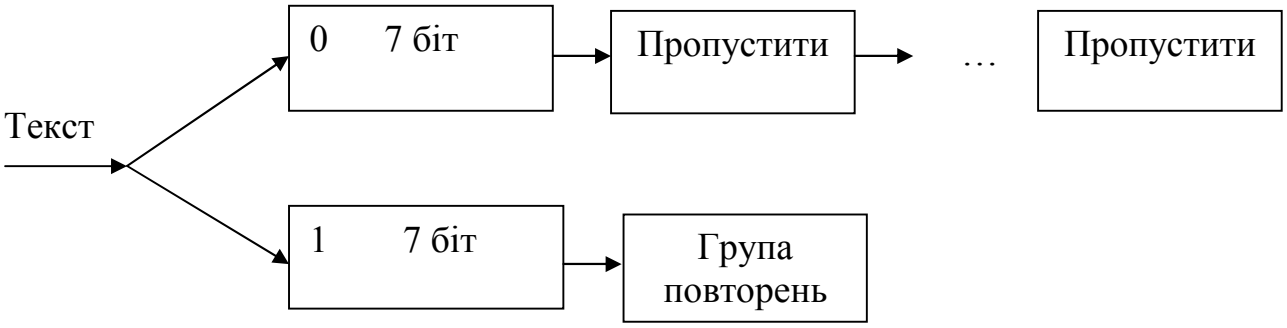

Рисунок 1.4 – Ознака повтору в алгоритмі

Як можна легко підрахувати, в кращому випадку цей алгоритм стискає файл в 64 рази (а не в 32 рази, як у попередньому варіанті), в гіршому - збільшує на 1/128. Середні показники ступеня компресії даного алгоритму знаходяться на рівні показників першого варіанту.

Схожі схеми компресії використані в якості одного з алгоритмів, які підтримуються форматом TIFF, а також в форматі TGA.

Алгоритм LZW отримав таку назву за першими літерами прізвищ його розробників - Lempel, Ziv і Welch. Стиснення в ньому, на відміну від RLE, здійснюється вже за рахунок однакових ланцюжків байт.

Існує досить велике сімейство LZ-подібних алгоритмів, що відрізняються, наприклад, методом пошуку ланцюжків, що повторюються. Один з досить простих варіантів цього алгоритму, наприклад, передбачає, що у вхідному потоці йде або пара <лічильник, зміщення відносно поточної позиції>, або просто <лічильник> байтів, які пропускаються, і самі значення байтів (як у другому варіанті алгоритму RLE). При розархівації для пари <лічильник, зміщення> копіюється байт з вихідного масиву, отриманого в результаті розархівації, а <лічильник> (тобто число, рівне лічильнику) значень пропущених байтів просто копіюються у вихідний масив з вхідного потоку. Даний алгоритм є несиметричним по часу, оскільки вимагає повного перебору буфера при пошуку однакових підрядків. В результаті складно поставити великий буфер через різке зростання часу компресії. Однак потенційно побудова алгоритму, в якому на <лічильник> і на <зміщення> буде виділено по 2 байти (старший біт старшого байта лічильника - ознака повтору рядка/копіювання потоку), дасть можливість стискати всі підрядки, які повторюються, розміром до 32Кб в буфері розміром 64 Кб (рисунок 1.5).

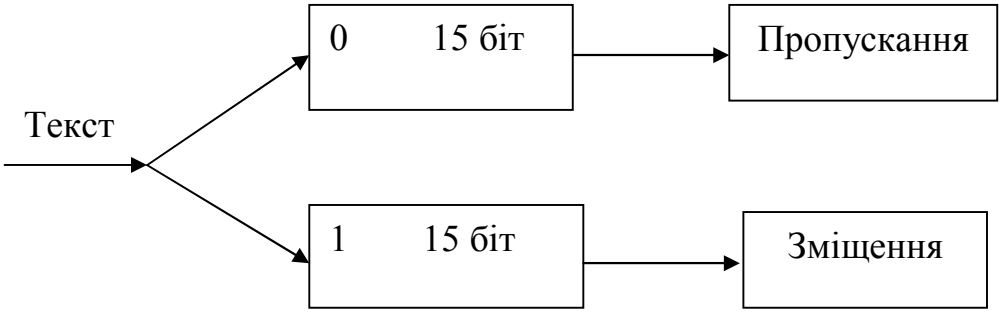

Рисунок 1.5 – Схема алгоритму LZ

При цьому отримується збільшення розміру файлу в гіршому випадку на 32770/32768 (у двох байтах записано, що потрібно переписати у вихідний потік наступні 215 байт), що зовсім непогано. Максимальний коефіцієнт стиснення буде приблизно 8192. Точно вказати неможливо, оскільки максимальне стиснення отримується при перетворенні 32 Кб буфера в 4 байти, а буфер

такого розміру назбирається не відразу. Однак мінімальний підрядок, для якого вигідно проводити стиснення, повинен складатися в загальному випадку мінімум з 5 байт, що і визначає малу цінність даного алгоритму. До переваг LZ можна віднести надзвичайну простоту алгоритму декомпресії.

Алгоритм LZW використовує дерево для подання та зберігання ланцюжків. Очевидно, що це досить сильне обмеження на вид ланцюжків, і далеко не всі однакові підланцюжки будуть використані при стисненні. Однак в даному алгоритмі вигідно стискати навіть ланцюжки, що складаються з 2 байт.

Процес стиснення виглядає досить просто. Зчитуються послідовно символи вхідного потоку і перевіряється, чи є у створеній таблиці відповідний рядок. Якщо рядок є, то зчитується наступний символ, а якщо немає, то заноситься в потік код для попереднього знайденого рядка, рядок заноситься в таблицю і починається пошук знову.

Таблиця рядків ініціалізується так, щоб вона містила всі можливі рядки, що складаються з одного символу. Наприклад, якщо стискаються байтові дані, то таких рядків в таблиці буде 256 (0, 1, ..., 255). Для коду очищення (ClearCode) і коду кінця інформації (CodeEndOfInformation) зарезервовані значення 256 і 257. В даному варіанті алгоритму використовується 12-бітний код, і, відповідно, під коди для рядків залишаються значення від 258 до 4095. Додані рядки записуються в таблицю послідовно, при цьому індекс рядка в таблиці стає її кодом.

Розглянемо приклад. Нехай стискається послідовність 45, 55, 55, 151, 55, 55, 55. Тоді, відповідно до викладеного вище алгоритму, у вихідний потік спочатку поміщається код очищення <256>, потім додається до спочатку порожнього рядка 45 і перевіряється, чи є рядок 45 в таблиці. Оскільки при ініціалізації занесені в таблицю всі рядки з одного символу, то рядок 45 є в таблиці. Далі зчитується наступний символ 55 з вхідного потоку і перевіряється, чи є рядок 45, 55 в таблиці. Такого рядка в таблиці поки немає.

Тоді заноситься в таблицю рядок 45, 55 (з першим вільним кодом 258) і записується в потік код <45>. Можна коротко представити архівацію так:

1) 45 - є в таблиці;

2) 45, 55 - немає. Додається в таблицю <258> 45, 55. У потік: <45>;

3) 55, 55 - немає. У таблицю: <259> 55, 55. У потік: <55>;

4) 55, 151 - немає. У таблицю: <260> 55, 151. У потік: <55>;

5)151, 55 - немає. У таблицю: <261> 151, 55. У потік: <151>;

6) 55, 55 - є в таблиці;

7) 55, 55, 55 - немає. У таблицю: 55, 55, 55 <262>. У потік: <259>.

Послідовність кодів для даного прикладу, що потрапляють у вихідний потік: <256>, <45>, <55>, <55>, <151>, <259>.

Особливість LZW полягає в тому, що для декомпресії не треба зберігати таблицю рядків у файл для розпакування. Алгоритм побудований таким чином, що можна відновити таблицю рядків, користуючись тільки потоком кодів.

Крім того, для кожного коду треба додавати в таблицю рядок, що складається з уже наявного там рядка і символу, з якого починається наступний рядок в потоці (рисунок 1.6).

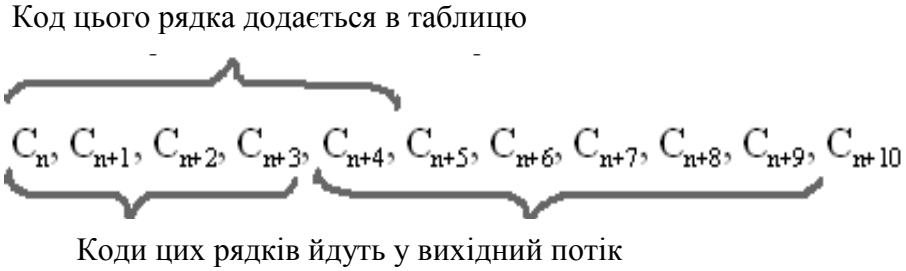

Рисунок 1.6 – Схема додавання рядка до коду

Підрахуємо кращий і гірший коефіцієнти компресії для даного алгоритму. Кращий коефіцієнт, очевидно, буде отримано для ланцюжка однакових байт великої довжини (тобто для 8-бітного зображення, всі точки якого мають, для визначеності, колір 0). При цьому в 258 рядок таблиці записується рядок 0, 0, в 259 - 0, 0, 0, ..., в 4095 - рядок з 3839 нулів. При цьому в потік потрапить (можна

перевірити за алгоритмом) 3840 кодів, включаючи код очищення. Отже, порахувавши суму арифметичної прогресії від 2 до 3839 (тобто довжину стиснутого ланцюжка) і поділивши її на 3840·12/8 (в потік записуються 12-бітові коди), отримується кращий коефіцієнт компресії.

Найгірший коефіцієнт буде отримано, якщо жодного разу не зустрінеться підрядок, який вже є в таблиці (в ній не повинно зустрітися жодної однакової пари символів).

У разі, якщо постійно буде зустрічатися новий підрядок, то у вихідний потік запишеться 3840 кодів, яким буде відповідати рядок з 3838 символів. Це складе збільшення файлу майже в 1,5 рази.

LZW реалізований в форматах GIF і TIFF.

1.2 Суть арифметичного кодування

Арифметичне кодування є методом, що дозволяє упаковувати символи вхідного алфавіту без втрат за умови, коли відомий розподіл частот цих символів [24-25]. Арифметичне кодування є оптимальним, досягаючи теоретичної межі ступеня стиснення. Арифметичне кодування - блочне і вихідний код є унікальним для кожного з можливих вхідних повідомлень; його не можна розбити на коди окремих символів, на відміну від кодів Хаффмана, які є неблочними, тобто кожній букві вхідного алфавіту ставиться у відповідність певний вихідний код [26].

Текст, стиснутий арифметичним кодером, розглядається як деякий двійковий дріб з інтервалу [0, 1). Результат стиснення можна уявити як послідовність двійкових цифр із запису цього дробу. Кожен символ вихідного тексту представляється відрізком на числовій осі з довжиною, що дорівнює ймовірності його появи і початком, що збігається з кінцем відрізка символу, що

передує йому в алфавіті. Сума всіх відрізків, очевидно, повинна дорівнювати одиниці. Якщо розглядати на кожному кроці поточний інтервал як ціле, то кожен вхідний символ "вирізає" з нього підінтервал пропорційно своїй довжині і положенню [27].

На рисунку 1.7 представлена побудова інтервалу для повідомлення «АБВГ ...».

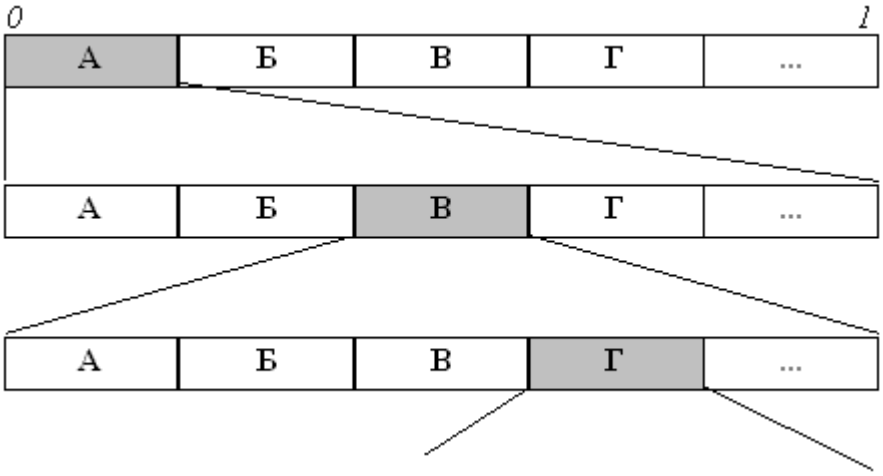

Рисунок 1.7 – Схема побудови інтервалу для повідомлення «АБВГ ...»

Пояснимо роботу методу на прикладі. Нехай алфавіт складається з двох символів: "a" і "b" з вірогідністю відповідно 3/4 і 1/4. Розглянемо (відкритий справа) інтервал [0, 1). Розіб'ємо його на частини, довжини яких пропорційні ймовірностям символів. У нашому випадку це [0, 3/4) і [3/4, 1). Суть алгоритму в наступному: кожному слову у вхідному алфавіті відповідає певний підінтервал з [0, 1). Порожньому слову відповідає весь інтервал [0, 1). Після отримання кожного чергового символу арифметичний кодер зменшує інтервал, обираючи ту його частину, яка відповідає символу, який надійшов. Кодом повідомлення є інтервал, виділений після обробки всіх його символів, точніше, число мінімальної довжини, що входить в цей інтервал. Довжина отриманого інтервалу пропорційна ймовірності появи кодованого тексту. Виконаємо алгоритм для ланцюжка "aaba" (таблиця 1.1).

Таблиця 1.1 – Схема кодування слова «aaba»

Крок	Переглянутий ланцюжок	Інтервал
0	“”	[0, 1) = [0, 1)
1	“a”	[0, 3/4) = [0, 0.11)
2	“aa”	[0, 9/16) = [0, 0.1001)
3	“aab”	[27/64, 36/64) = [0.011011, 0.100100)
4	“aaba”	[108/256, 135/256) = [0.01101100, 0.10000111)

На першому кроці беруться перші 3/4 інтервалу, які відповідають символу "a", потім залишається від нього ще тільки 3/4. Після третього кроку від попереднього інтервалу залишиться його права чверть відповідно до положення та ймовірністю символу "b". І, нарешті, на четвертому етапі залишається лише перші 3/4 від результату. Це і є інтервал, якому належить вихідне повідомлення.

В якості коду можна взяти будь-яке число з діапазону, отриманого на кроці 4. Виникає питання: "А де ж тут стиснення? Оригінальний текст можна було закодувати чотирма бітами, а отримали восьмибітний інтервал". Справа в тому, що в якості коду можна вибрати, наприклад, найкоротший код з інтервалу, що дорівнює 0.1 і отримати чотириразове скорочення обсягу тексту. Для порівняння, код Хаффмана не зміг би стиснути подібне повідомлення.

Арифметичний декодер працює синхронно з кодером: почавши з інтервалу [0, 1), він послідовно визначає символи вхідного ланцюжка. Зокрема, в нашому випадку він спочатку розділить (пропорційно частотам символів) інтервал [0, 1) на [0, 0.11) і [0.11, 1). Оскільки число 0.1 (переданий кодером код ланцюжка "aaba") знаходиться в першому з них, то можна отримати перший символ: "a". Потім треба поділити перший підінтервал [0, 0.11) на [0, 0.1001) і [0.1001, 0.1100) (пропорційно частотам символів). Знову вибираємо перший,

так як 0 <0.1 <0.1001. Продовжуючи цей процес, однозначно можна декодувати всі чотири символу.

При розгляді цього методу виникають дві проблеми: по-перше, необхідна матеріальна арифметика, взагалі кажучи, необмеженої точності, і по-друге, результат кодування стає відомий лише при закінченні вхідного потоку. Подальші дослідження, однак, показали, що можна практично без втрат обійтися цілочисельною арифметикою невеликої точності (16-32 розряди), а також домогтися інкрементальної роботи алгоритму: цифри коду можуть видаватися послідовно в міру читання вхідного потоку [28].

Подібно алгоритму Хаффмана, арифметичний кодер також є двопрохідним і вимагає передачі разом з закодованим текстом ще і таблиці частот символів. Взагалі, ці алгоритми дуже схожі і можуть легко взаємозамінюватися. Отже, існує адаптивний алгоритм арифметичного кодування з усіма випливаючими достоїнствами і недоліками. Його основна відмінність від статичного в тому, що нові інтервали ймовірності перераховуються після отримання кожного наступного символу з вхідного потоку [29].

Одним з представників методів арифметичного кодування є метод Хаффмана [30]. Він простий, але ефективний тільки в тому випадку, коли ймовірності появи символів рівні числам $(1/2)^n$, де – n - будь-яке ціле позитивне число. Це пов'язано з тим, що код Хаффмана присвоює кожному символу алфавіту код з цілим числом біт. Разом з тим, в теорії інформації відомо, що, наприклад, при ймовірності появи символу, рівною 0,4, йому в ідеалі слід поставити код довжиною $-\log_2 0{,}4 \approx 1{,}32$ біт. Зрозуміло, що при побудові кодів Хаффмана не можна задати довжину коду в 1,32 біта, а тільки лише в 1 або 2 біта, що призведе в результаті до погіршення стиснення даних. Арифметичне кодування вирішує цю проблему шляхом присвоєння коду всьому, звичайно, великому файлу, який передається, замість кодування окремих символів.

Ідею арифметичного кодування краще всього розглянути на простому прикладі. Нехай необхідно закодувати три символи вхідного потоку, для визначеності – це рядок SWISS_MISS із такими заданими частотами появи символов: S = 0,5, W – 0,1, I – 0,2, M – 0,1 і _ - 0,1. В арифметичному кодері кожний символ представляється інтервалом в діапазоні чисел [0, 1) у відповідності з частотою його появи. В даному прикладі для символів нашого алфавіту отримаємо набори інтервалів, представлені на рисунку 1.8.

Рисунок 1.8 - Розподіл інтервалів представлення символів

Процес кодування починається зі зчитування першого символу вхідного потоку і присвоєння йому інтервалу з початкового діапазону [0, 1). В даному випадку для першого символу S отримується діапазон [0,5, 1). Потім зчитується другий символ - W, якому відповідав би діапазон [0,4, 0,5). Але вихідний діапазон [0, 1) вже скоротився до [0,5, 1), тому символ W необхідно представити в цьому новому діапазоні. Для цього достатньо обчислити нові нижню і верхню межі. Значення 0,4 відповідатиме значенню 0,7, а значення 0,5 - значенню 0,75 (рисунок 1.9).

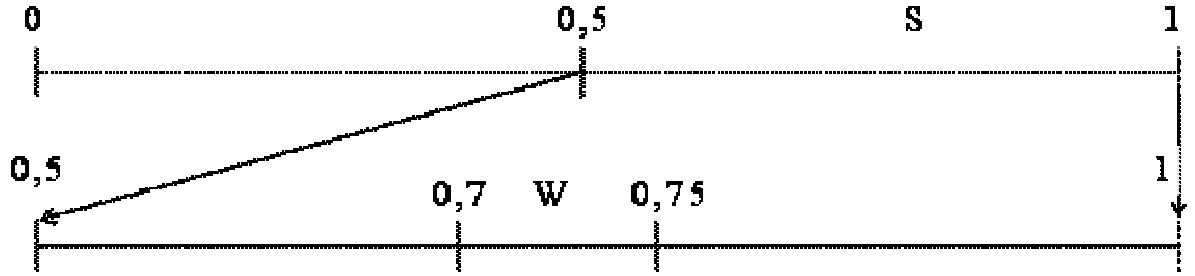

Рисунок 1.9 - Схема представлення нових границь символа W

Дані границі можна вичислити по таких формулах:

$$NewHigh = OldLow + (OldHigh-OldLow)\cdot HighRange(X); \quad (1.1)$$

$$NewLow = OldLow + (OldHigh-OldLow)\cdot LowRange(X); \quad (1.2)$$

де OldLow – нижня границя інтервалу, в якому представляється поточний символ;

OldHigh – верхня границя інтервала;

HighRange(X) – вихідна верхня границя кодованого символа;

LowRange(X) – вихідна нижня границя кодованого символа.

Використовуючи дані формули до обчислення границь символа W, отримуєм: OldLow = 0,5; OldHigh = 1; HighRange(W) = 0,5; LowRange(W) = 0,4; NewHigh = 0,5 + (1-0,5) ·0,5 = 0,75; NewLow = 0,5 + (1-0,5) ·0,4 = 0,7.

Аналогічним чином виконується кодування символу I, для якого нові інтервали також можна обчислити формулами (1.1)-(1.2): OldLow = 0,7; OldHigh = 0,75; HighRange(I) = 0,4; LowRange(I) = 0,2; NewHigh = 0,7 + (0,75--0,7) ·0,4 = 0,72; NewLow = 0,7 + (0,75-0,7) ·0,2 = 0,71.

Нижче, в таблиці 1.2 представлені значення границь при кодуванні рядка SWISS_MISS.

Кінцевий вихідний код - це останнє значення змінної Low, рівне 0.71753375, з якого слід взяти лише вісім цифр – 71753375 - для запису у файл.

Тепер потрібно розглянути можливість відновлення закодованої інформації по восьми цифрам 71753375 і відомим інтервалах символів. Перша з восьми цифр - це 7, тобто 0,7. Вона належить одному із заданих інтервалів [0.5, 1), який відповідає символу S. Тому перший декодований символ - це S. Тепер потрібно повернутися до рисунка 1.9 і зауважити, що другий символ був представлений в інтервалі символу S, тобто [0.5, 1). Але для зручності декодування його краще уявити в вихідному інтервалі [0, 1). Для цього

достатньо інтервал [0,5, 1) збільшити до початкового, тобто помножити на два і границі зрушити на величину 0.5·2 = 1 (рисунок 1.10).

Таблиця 1.2 - Значення границь при кодуванні рядка SWISS_MISS

Символ	Границі	
	Нижня	Верхня
S	0.0+(1.0-0.0)·0.5 = 0.5	0.0+(1.0-0.0) ·1.0 = 1.0
W	0.5+(1.0-0.5) ·0.4=0.70	0.5+(1.0-0.5) ·0.5=0.75
I	0.7+(0.75-0.7) ·0.2=0.71	0.7+(0.75-0.7) ·0.4=0.72
S	0.71+(0.72-0.71) ·0.5=0.715	0.71+(0.72-0.71) ·1.0=0.72
S	0.715+(0.72-0.715) ·0.5=0.7175	0.715+(0.72-0.715) ·1.0=0.72
_	0.7175+(0.72-0.7175) · ·0.0=0.7175	0.7175+(0.72-0.7175) · ·0.1=0.71775
M	0.7175+(0.71775-0.7175) · ·0.1=0.717525	0.7175+(0.71775-0.7175) · ·0.2=0.717550
I	0.717525+(0.717550-0.717525) · ·0.2=0.717530	0.717525+(0.717550-0.717525) ·0.4=0.717535
S	0.717530+(0.717535-0.717530) · ·0.5=0.7175325	0.717530+(0.717535-0.717530) ·1.0=0.717535
S	0.7175325+(0.717535- -0.7175325) ·0.5=0.71753375	0.7175325+(0.717535- 0.7175325) ·1.0=0.717535

Застосовуючи цю схему до числа 0.71753375, отримуємо нижню межу наступного закодованого символу, ніби він був початковим при кодуванні: 0.71753375·2-1 = 0.4350675.

Отримане значення належить діапазону [0.4, 0.5), який відповідає символу W. Потім отримане число 0.4350675 слід нормувати, що в загальному випадку виконується за формулою:

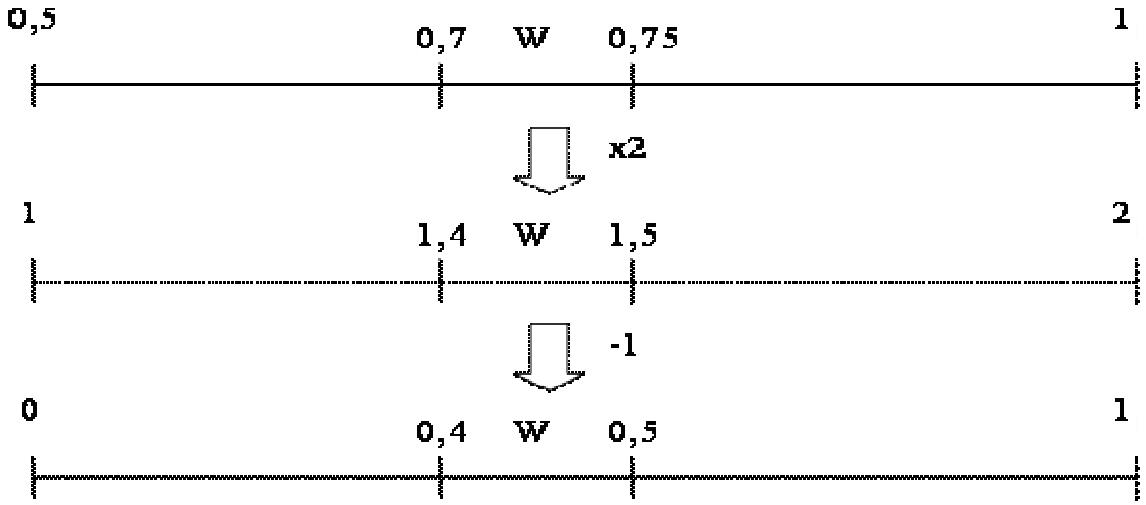

Рисунок 1.10 - Схема відновлення вихідних інтервалів символу W

Code = (Code-LowRange (X)) / (HighRange (X) -LowRange (X)), (1.3)

де Code - поточне значення коду.

Наприклад, користуючись цією формулою для коду 0.71753375, отримуєм значення Code = (0.71753375-0.5) / (1-0.5) = 0.4350675, яке в точності збігається з попередньою схемою обчислення. Аналогічним чином виконується декодування всіх символів рядка. У таблиці 1.2 представлені коди, що обчислюються при декодуванні символів. Можна помітити, що процес декодування можна зупинити, якщо значення коду дорівнює нулю.

Може здатися, що наведений вище приклад не робить ніякого стиснення. Для того, щоб з'ясувати ступінь стиснення результати кодування потрібно перевести в двійкову форму. Так як з кінцевого інтервалу [0.71753375 0.717535) можна вибрати будь-яке число, виберемо найменше для зберігання - це 717 534, якому відповідало б бітове подання 10101111001011011110, яке містить 20 біт. Тобто рядок з 10 символів стискається в 20 біт. Чи це хороше стиснення? Для цього треба знайти ентропію кодованої послідовності [31], яка дорівнюватиме $H(X) = -0,5\log_2 0,5 - 0,2\log_2 0,2 - 0,1\log_2 0,1 - 0,1\log_2 0,1 - 0,1\log_2 0,1 \approx 1,96$ біт/символ.

Загальна ентропія $I = H(X) \cdot 10 \approx 19,6$ біт, що в цілих значеннях дорівнює 20 біт. Отже, отриманий код досяг мінімально можливого значення і є

оптимальним. У загальному випадку можна показати, що при досить великій послідовності арифметичний кодер завжди приводить до оптимальних результатів стиснення, тобто є найкращим серед усіх ентропійних кодерів [32].

1.3 Постановка задачі

Метою даної магістерської роботи є розробка алгоритму арифметичного кодування інформації з низькою надлишковістю для стиснення цифрової інформації та порівняння його параметрів з існуючими алгоритмами. На рисунку 1.11 представлено дерево рішень для розробки магістерської роботи.

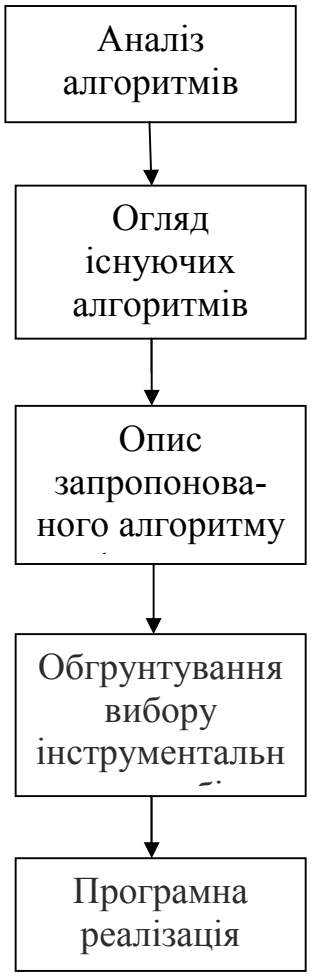

Рисунок 1.1 – Дерево рішень дипломного проекту

Для досягнення поставленої мети потрібно вирішити наступні завдання:

– аналіз методів та засобів стиснення цифрової інформації;

– огляд та аналіз сучасних алгоритмів арифметичного кодування з низькою надлишковістю;

– розробка алгоритму арифметичного кодування на основі перетворення вхідного потоку в число з плаваючою точкою;

– обгрунтування вибору інструментальних засобів для реалізації алгоритму;

– програмна реалізація запропонованого алгоритму арифметичного кодування;

– порівняльний аналіз алгоритмів арифметичного кодування;

– експериментальне дослідження ефективності стиску.

2 АЛГОРИТМИ АРИФМЕТИЧНОГО КОДУВАННЯ З НИЗЬКОЮ НАДЛИШКОВІСТЮ

2.1 Унарні коди

Компактне представлення інформації - дуже важлива проблема в областях, де доводиться працювати із стисненням даних. Мета - стиснення потоку R-бітових елементів. У загальному випадку ніяких припущень про властивості значень елементів не робиться, тому можна говорити про опис способів представлення цілих чисел.

Арифметичне кодування відомо сьогодні як один з найбільш ефективних методів стиснення даних, який можна застосувати для великого класу джерел інформації [33-35]. Основна ідея арифметичного кодування була сформульована Елайєсом ще на початку 60-х років. Перевага арифметичного коду по відношенню до інших методів полягає в тому, що він дозволяє досягти довільно низькою надмірності на символ джерела (надмірність - центральне поняття в теорії стиснення інформації; будь-які дані з надлишковою інформацією можна стиснути; в яких немає надмірності - стиснути не можна) та показує високу ефективність для дрібних нерівномірних інтервалів розподілу ймовірностей кодованих символів [36-37].

Основна ідея полягає в тому, щоб окремо зберігати порядок значення елемента X_i (експоненту E_i) і окремо - значущі цифри значення (мантиссу M_i).

Методи даної групи є трансформуючими і поточними, тобто можуть застосовуватися навіть у тому випадку, коли обсяг вхідних даних заздалегідь невідомий. У загальному випадку швидкість роботи компресора (що містить пряме, стискаюче перетворення) дорівнює швидкості декомпресора (реалізує зворотне, розтискаюче перетворення) і залежить тільки від обсягу вихідних даних. Пам'яті потрібно всього кілька байтів.

Існує чотири варіанти цього методу [38-40]:

- Fixed + Fixed (фіксована довжина експоненти - фіксована довжина мантиси);

- Fixed + Variable (фіксована довжина експоненти - змінна величина мантиси);

- Variable + Variable (змінна довжина експоненти - змінна величина мантиси);

- Variable + Fixed (змінна довжина експоненти - фіксована довжина мантиси).

Унарний код зіставляє числу i двійкову комбінацію виду $1^i 0$.Запис виду 0^m або 1^m означає відповідно серію з m нулів або одиниць. Наприклад, унарними кодами чисел 1, 2, і 3 є такі послідовності: unar (1) = 10, unar (2) = 110 і unar (3) = 1110 відповідно. Довжина кодового слова для числа n дорівнює $l_n=n+1$. На рисунку 2.1 наведено унарний код чисел від 0 до 6.

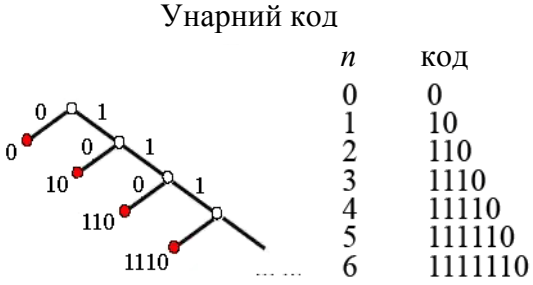

Рисунок 2.1 - Унарний код чисел від 0 до 6

Унарний код оптимальний, якщо числа i розподілені по геометричному закону $p^i=(1-\alpha)^{i-1}\alpha$ з параметром $\alpha=1/2$, де $i=1,2,\cdots$.

Для значень $\alpha<1/2$ більш ефективний код Голомба. Це сімейство ентропійних кодерів є загальним випадком унарного коду. Кодування ентропії – це кодування словами (кодами) змінної довжини, при якій довжина коду символу має обернену залежність від ймовірності появи символу в переданому повідомленні. Зазвичай, ентропійні кодувальники використовують для стиснення даних коди, довжини яких пропорційні від'ємному логарифму

ймовірності символу. Таким чином, найбільш ймовірні символи використовують найкоротші коди. Згідно з теоремою Шеннона, оптимальна довжина коду для символу дорівнює $-\log_b P$, де b- це кількість символів, які використовуються для виготовлення вихідного коду, P- ймовірність вхідного символу.

Унарний код - це ентропійне кодування, яке представляє число n у вигляді n одиниць з замикаючим нулем (або n нулів і одиниця). Наприклад, 5 представляється у вигляді 111110. Унарне кодування оптимальнt для розподілу ймовірності: $P(x)=2^{-(x+1)}$. Також під кодом Голомба може матися на увазі один з представників цього сімейства [41].

Код Голомба дозволяє уявити послідовність символів у вигляді послідовності двійкових слів. Це уявлення буде оптимальним за умови, що розподіл ймовірності символів підпорядковується геометричному закону:

$$P(i)=(1-p)p^i, \tag{2.2}$$

де i - номер символу;

p - параметр геометричного розподілу.

Також необхідно дотримуватися умови $p^m=1/2$, де m - основний параметр коде Голомба. Для кодування символу з номером n необхідно представити n в такому вигляді:

$$n=qm+r, \tag{2.3}$$

де q і r - цілі додатні числа, $0 \le r < m$.

Потім r кодується унарним кодом, а q - бінарним. Отримані двійкові послідовності об’єднуються в результуюче слово.

Розглянемо приклад. Основний параметр коду m=4, кодоване число n=13. Далі виконується така послідовність операцій:

- частка $q=\left\lfloor \frac{n}{m} \right\rfloor=\left\lfloor \frac{13}{4} \right\rfloor=3$;
- унарний код q - 1110;
- остача $r=n \bmod m=13 \bmod 4=1$;
- бінарний код r - 01;
- результуюче кодове слово - 111001.

Декілька прикладів кодів Голомба для різного параметра m наведені в таблиці 2.1.

Таблиця 2.1 - Коди Голомба для різних параметрів m

	m				
n	1	2	3	4	5
0	0	00	00	000	000
1	10	01	010	001	001
2	110	100	011	010	010
3	1110	101	100	011	0110
4	11110	1100	1010	1000	0111
5	111110	1101	1011	1001	1000
6	1111110	11100	1100	1010	1001
7	11111110	11101	11010	1011	1010
8	111111110	111100	11011	11000	10110
9	1111111110	111101	11100	11001	10111
10	11111111110	1111100	111010	11010	11000
11	111111111110	1111101	111011	11011	11001
12	1111111111110	11111100	111100	111000	11010
13	11111111111110	11111101	1111010	111001	110110
14	111111111111110	111111100	1111011	111010	110111
15	1111111111111110	111111101	1111100	111011	111000
16	11111111111111110	1111111100	11111010	1111000	111001
17	111111111111111110	1111111101	11111011	1111001	111010

На рисунку 2.2 наведено пояснення формування даних кодів, тобто як саме двійкове представлення залишку слідує за унарним кодом.

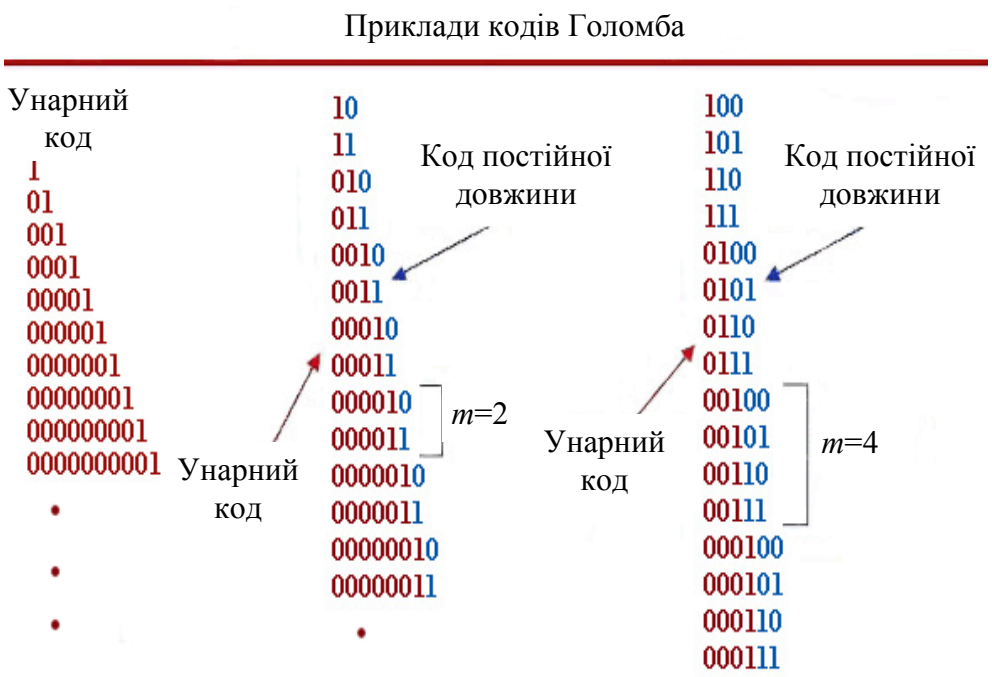

Рисунок 2.2 - Формування кодів Голомба

Отже, для його реалізації метод Голомба вимагає введення певного параметра *M*. Якщо значення *M* дорівнює двійці в цілому позитивному степені, то код Голомба переходить в свій окремий випадок - код Райса.

Нехай є число *N*, яке потрібно закодувати, і фіксоване число *M*. Кроки алгоритму записуються так:

- знаходиться частка $q=N/M$ з цілочисельним розподілом;

- шукається залишок від ділення *N*/*M*: $r=N \bmod M$;

- закодоване число *N* має такий формат: <код *q*> <код *r*>;

- код *q* є просто унарним кодуванням числа *q*, тобто представляється у вигляді: 111 ... 110, де кількість одиниць дорівнює самому числу *q*;

- для знаходження коду *r* треба ввести параметр $b = [\log_2 (M)]$, причому *b* округлюється в сторону більшого цілого, і параметр $c = 2b\text{-}M$. Далі, якщо число $0 \le r < c$, то код *r* - це просто бінарний код числа *r*, поміщений в кількість біт, що дорівнює *b*-1; якщо ж $r \ge c$, то код *r* - це бінарний код числа (*r*+c), поміщений в кількість біт, що дорівнює *b*. Блок-схема, що описує логіку вищеописаного алгоритму, показана на рисунку 2.3.

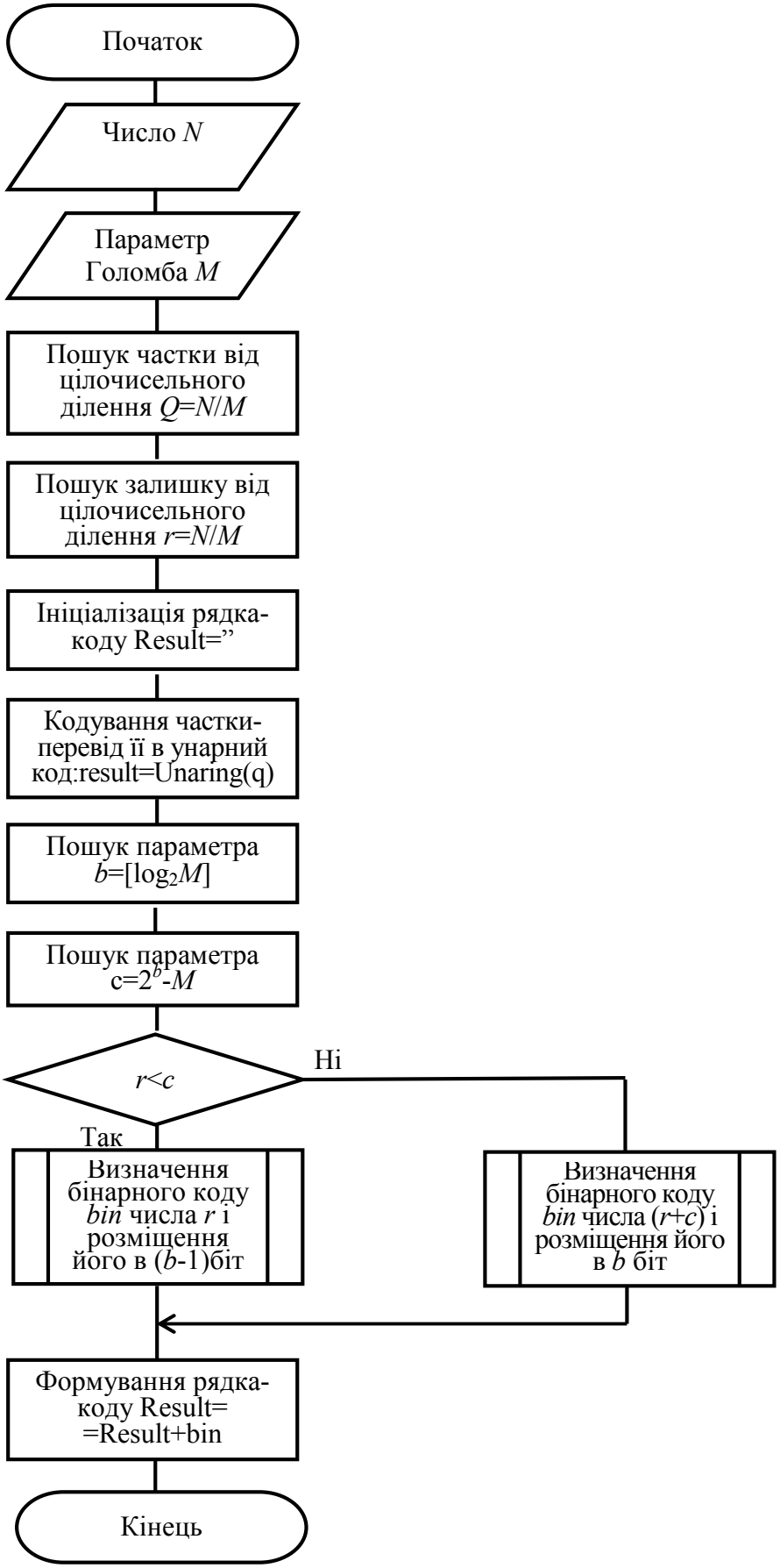

Рисунок 2.3 - Блок-схема алгоритму кодування методом Голомба

2.2 Коди Фібоначчі

Це є найцікавіші, нетривіальні коди [42]. В даному кодуванні вихідне число n розкладається в суму чисел Фібоначчі f_i ($f_1 = 1$; $f_2 = 2$; $f_i = f_{i-1} + f_{i-2}$). Відомо, що будь-яке натуральне число однозначно можна представити у вигляді суми чисел Фібоначчі. Тому можна побудувати код числа як послідовність бітів, кожен з яких вказує на факт наявності в n певного числа Фібоначчі [43].

Зауважимо також, що якщо в розкладанні числа n присутній член f_i, то в цьому розкладанні не може бути числа f_{i+1}. Тому логічно для кінця коду використовувати додаткову одиницю. Тоді дві одиниці, що йдуть підряд, будуть означати закінчення кодування поточного числа.

Числа Фібоначчі - елементи числової послідовності: 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, … , в якій кожне наступне число дорівнює сумі двох попередніх чисел. Свою назву послідовність отримала по імені середньовічного математика Леонардо Пізанського (або Фібоначчі).

Більш формально, послідовність чисел Фибоначчі $\{F_n\}$ задається рекурентним співвідношенням:

$$F_1 = 1,\ F_2 = 1,\ F_{n+1} = F_n + F_{n-1},\ n \in N; \tag{2.4}$$

Інколи числа Фібоначчі розглядаються і для недодатніх номерів n як двостороння нескінченна послідовність, що задовольняє основному співвідношенню. Члени з такими номерами легко отримати з допомогою еквівалентної формули «назад»:

$$F_n = F_{n+2} - F_{n+1} \tag{2.5}$$

Легко бачити, що $F_{-n} = (-1)^{n+1} F_n$.

Нехай є деяке число X, яке слід закодувати за допомогою чисел або послідовності Фібоначчі. По суті, потрібно розкласти це число X на числа Фібоначчі.

Покроковий алгоритм виглядатиме так:

- шукається число Фібоначчі, найбільш близьке до числа X. Нехай це буде число F_x; а i_x - порядковий номер числа F_x, тобто $F(i_x) = F_x$.

- в i_x-му біті коду ставиться «1»;

-від числа X віднімається F_x;

- кроки 1, 2, 3 повторюються до тих пір, поки $X > 0$;

- в отриманій кодовій послідовності на місцях, де немає «1», ставиться «0», а після останньої «1» ставиться ще одна «1».

Блок-схема, що описує логіку вищеописаного алгоритму, показана на рисунку 2.4.

2.3 Коди Еліаса

Гамма-код Еліаса – це двійковий префіксний код для представлення натуральних чисел [44]. Число кодується в два етапи. Визначається кількість двійкових розрядів кодованого числа. Спочатку кодується n за допомогою інвертованого унарного коду, після чого в незмінному вигляді дописують n молодших розрядів (старший одиничний розряд, таким чином, опускається). Іншими словами, кодом є число в двійковому представленні, заповнене нулями, довжина якого менша на 1 [45]. В загальному довжина коду повинна дорівнювати $2n+1$. У таблиці 2.2 наведені гамма-коди Еліаса для малих чисел натурального ряду. Розглянемо суть алгоритму кодування на прикладі цілого додатного числа N:

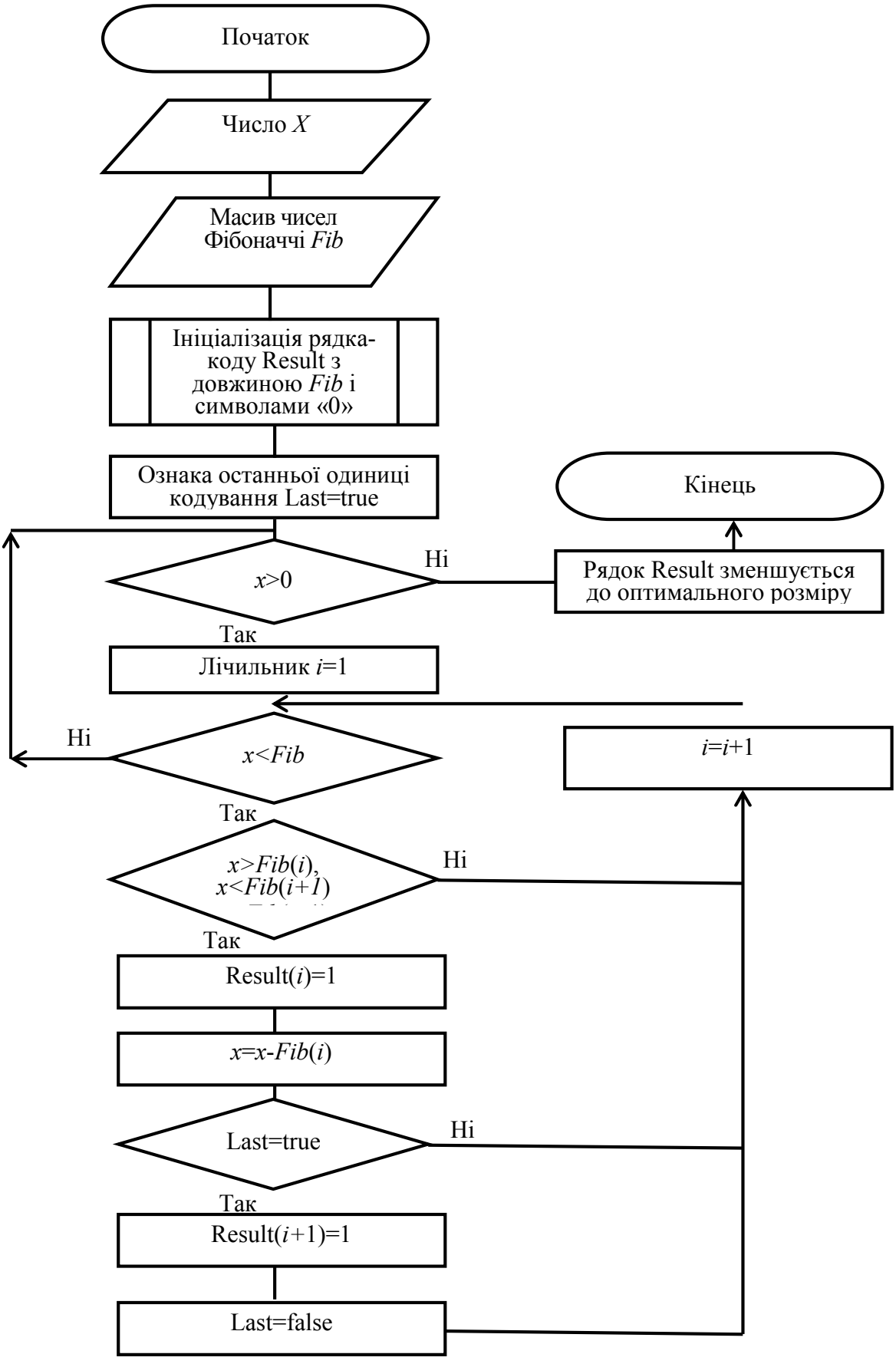

Рисунок 2.4 - - Блок-схема алгоритму кодування з допомогою чисел Фібоначчі

Таблиця 2.2 - Гамма-коди Еліаса для малих натуральних чисел

Числа	Код	Довжина
1	1	1
2-3	01x	3
4-7	001xx	5
8-15	0001xxx	7
16-31	00001xxxx	9
32-63	000001xxxxx	11
64-127	0000001xxxxxx	13

- шукається $b = \log_2(N)$ - максимальна ціла степінь, підносячи до якої «2», отримується число, максимально наближене до N;

- шукається число $q = N\text{-}2^b$;

- гамма-код числа N має вигляд: <код b><код q>;

- код b - інверсний унарний код числа b, тобто його можна представити у вигляді: 000…001, где кількість нулів дорівнює самому числу b;

- код q - просто бінарний код числа q, поміщений в кількість біт, яка дорівнює b.

Блок-схема, яка описує логіку вищеописаного алгоритму, показана на рисунку 2.5.

Дельта-код Еліаса є похідним від гамма-коду. На початку кодується кількість значущих двійкових розрядів в числі за допомогою гамма-коду, після чого записуються всі значущі розряди, за винятком старшої одиниці (загальною кількістю, яка на 1 менше закодованої кількості). У таблиці 2.3 наведені дельта-коди Еліаса для деяких чисел. Закодована кількість розрядів і решта числа розділені пропуском. Знаки x відповідають розрядам двійкового представлення кодованого числа.

Дельта-код дає більш оптимальний розподіл довжин для великих чисел, ніж для малих (відношення довжини коду до числу його розрядів прямує до одиниці). Декілька прикладів γ та δ кодів Еліаса наведені в таблиці 2.4.

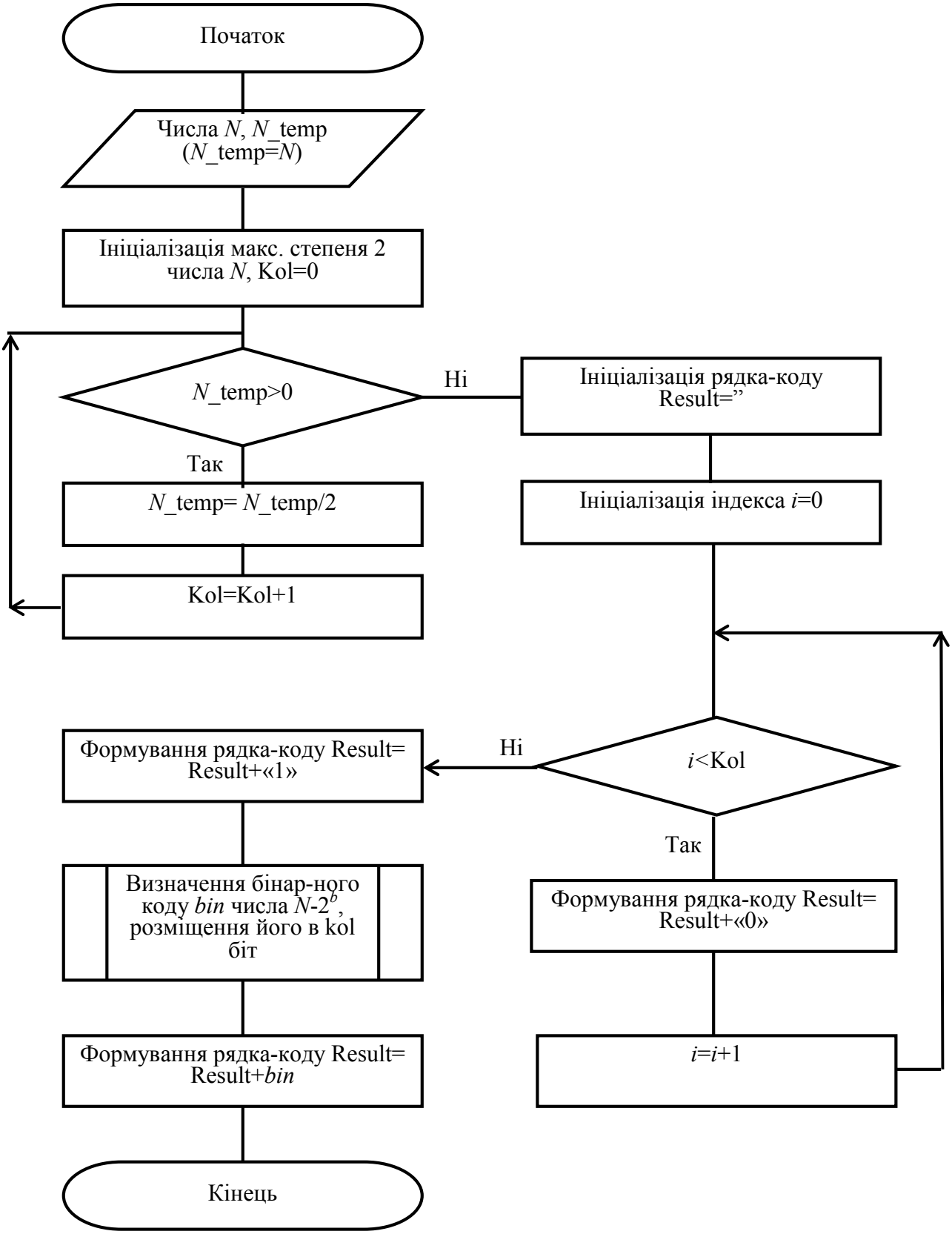

Рисунок 2.5 - Блок-схема алгоритму гамма - кодування Еліаса

Таблиця 2.3 - Дельта-коди Еліаса для деяких чисел

Числа	Код	Довжина
1	1	1
2-3	010 x	4
4-7	011 xx	5
8-15	00100 xxx	8
16-31	00101 xxxx	9
32-63	00110 xxxxx	10
64-127	00111 xxxxxx	11
128-255	0001000 xxxxxxx	14
256-511	0001001 xxxxxxxx	15

Таблиця 2.4 – γ- та δ- коди Еліаса

n	γ-код	δ-код
0	-	1
1	1	0 1
2…3	01x	0 01x
4…7	001xx	0 001xx
8…15	0001xxx	0 0001xxx
16…31	00001xxxx	0 00001xxxx
32…63	000001xxxxx	0 000001xxxxx

Розглянемо суть алгоритму кодування на прикладі цілого додатного числа *N*:

- шукається $b = \log_2(N)$ - максимальна ціла степінь, підносячи до якої «2», отримується число, максимально наближене до *N*;

- шукається число $q = N\text{-}2^b$;

- шукається параметр $b_1=b+1$;

- дельта-код числа *N* має вигляд: <код b_1><код q>;

- код b_1 - дельта-код Еліаса числа *b1*; код q - просто бінарний код числа q, поміщений в кількість біт, яке рівне b.

Блок-схема, яка описує логіку вищеописаного алгоритму, показана на рисунку 2.6.

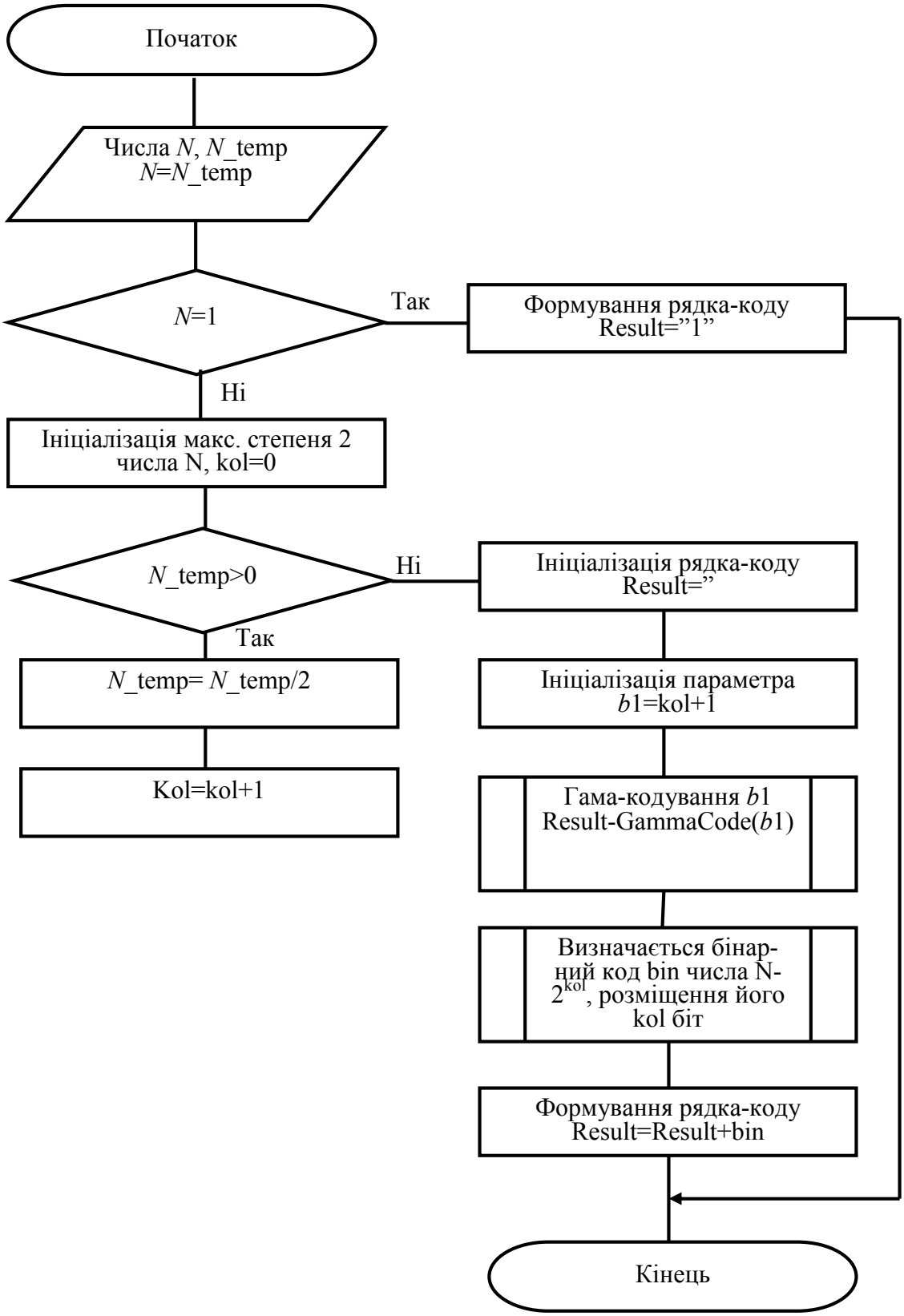

Рисунок 2.6 - Блок-схема алгоритму дельта - кодування Еліаса

2.4 Код Хаффмана

Код Хаффмана (Huffman code) ще називають мінімально-надлишковим префіксним кодом (minimum-redundancy prefix code) [46-48]. Ідея, що лежить в основі коду Хаффмана, досить проста. Замість того, щоб кодувати всі символи однаковим числом біт (як це зроблено, наприклад, в ASCII кодуванні, де на кожен символ відводиться рівно по 8 біт), символи, які зустрічаються частіше, кодуються меншою кількістю біт, ніж ті, які зустрічаються рідше. Код при цьому повинен бути оптимальний або, іншими словами, мінімально-надмірний. Першим такий алгоритм опублікував Девід Хаффман в 1952 році. Алгоритм Хаффмана двопрохідний. На першому проході будується частотний словник і генеруються коди. На другому проході відбувається безпосередньо кодування. Варто відзначити, що за 65 років з дня опублікування, код Хаффмана нітрохи не втратив своєї актуальності і значущості. Так з упевненістю можна сказати, що ми стикаємося з ним в тій чи іншій формі (справа в тому, що код Хаффмана рідко використовується окремо, частіше працюючи в зв'язці з іншими алгоритмами) практично кожен раз, коли архівуємо файли, дивимося фотографії, фільми , посилаємо факс або слухаємо музику [49-50].

Завдання побудови коду Хаффмана рівносильна задачі побудови відповідного йому дерева. Загальна схема побудови дерева Хаффмана:

1) складається список кодованих символів (при цьому кожен символ розглядається як одноелементне бінарне дерево, вага якого дорівнює вазі символу);

2) зі списку вибирається 2 вузли з найменшою вагою;

4) формується новий вузол і до нього приєднуються, як дочірні, два вузли, обраних зі списку. При цьому потрібно, що вага сформованого вузла дорівнювала сумі ваг дочірніх вузлів;

5) сформований вузол додається до списку;

6) якщо в списку більше одного вузла, то повторюється 2-5.

Найкраще продемонструвати цей алгоритм на простому прикладі. Нехай є п'ять символів a_1, a_2, a_3, a_4, a_5 з відомими ймовірностями: p_1=0,4, p_2=0,2, p_3=0,2, p_4=0,1, p_5=0,1. Для побудови кодів, спочатку потрібно вибрати пару символів з найменшими ймовірностями - це символи a_4, та a_5. Найменш ймовірному символу ставимо у відповідність бітовий нуль, а більш ймовірного - бітову одиницю: $a_4 \rightarrow 1$, $a_5 \rightarrow 0$.

Символи a_4 та a_5 умовно обє'днуються в єдиний символ a_{45} з ймовірністю появи 0,2. Потім береться третій символ з впорядкованого списку – це символ a_3 з ймовірністю 0,2. Тому код символа a_3 буде починатися з бітової 1, а до кодів символів a_4 та a_5 дописується бітовий нуль: $a_3 \rightarrow 1$, $a_4 \rightarrow 10$, $a_5 \rightarrow 00$.

Далі умовно об'єднуються всі три символи в символ a_{345} з ймовірністю появи 0,4 і розглядаються з наступним символом списку - a_2, ймовірність якого 0,2. Так як ймовірність появи символу a_2 менша ймовірності появи умовного символа a_{345} , то код символа a_2 буде починатися з нуля, а кодам символів a_3, a_4 та a_5 приписується бітова одиниця: a_2=0, $a_3 \rightarrow 11$, $a_4 \rightarrow 101$, $a_5 \rightarrow 001$.

Аналогічно для останнього символа a_1 отримується: $a_1 \rightarrow 0$, $a_2 \rightarrow 01$, $a_3 \rightarrow 111$, $a_4 \rightarrow 1011$, $a_5 \rightarrow 0011$.

В підсумку отримуєм коди Хаффмана, але записані в оберненому порядку, тобто для кодування символа a_1 маємо код 0, для a_2 - код 10, для a_3 - 111, a_4 - 1101 і a_5 - 1100. Відмітимо, що дані коди можут бути коректно декодовані. Наприклад, послідовність символів a_1, a_2, a_3, a_4, a_5 буде представлена такою послідовністю біт: 01011111011100.

В цій послідовності першим зустрічається бітовий нуль, але з нуля начинается только один код – код символа a_1, тому він так і декодується. Потім слідує код, який починається з бітової одиниці. Таких кодів декілька, тому необхідно прочитати наступний біт, який дорівнює 0. Код 10 – єдиний, який відповідає символу a_2. Після цього слідує код з трьома одиницями підряд (111).

Це говорить про те, що це є символ a_3. Нарешті останні два коди точно декодують символи a_4 та a_5.

На рисунку 2.7 представлена блок-схема алгоритму кодування за допомогою методу Хаффмана.

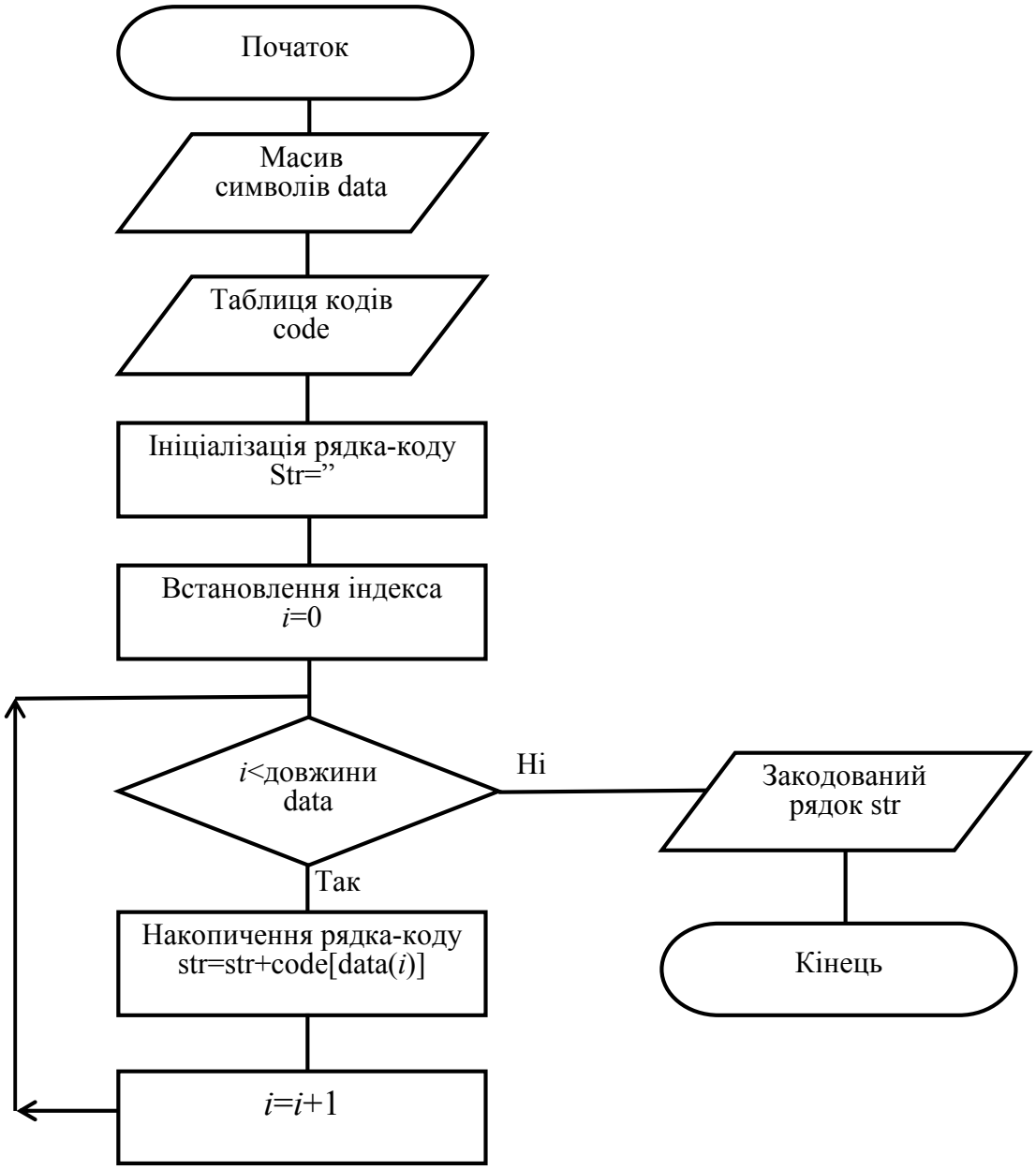

Рисунок 2.7 - Блок-схема алгоритму кодування за допомогою методу Хаффмана

Побудовані таким чином однозначні коди будуть в середньому тратити 2,2 біти на символ: 0,4·1+0,2·2+0,2·3+0,1·4+0,1·4=2,2 біт/символ.

Для порівняння, звичайний двійковий код містив би 3 біти для представлення одного символа, тобто в цьому випадку послідовність можна було б стиснути в 3/2,2=1,36 рази.

Щоб виконати декодування даних необхідно знати побудовані коди. Найкращий спосіб передати декодеру частоти появи символів в потоці, тобто в даному випадку – це додаткових п'ять чисел. На основі цих частот декодер будує коди, а потім застосовує їх для декодування послідовності. Тому, чим довша кодована послідовність, тим менша питома вага службової інформації, яка передається декодеру.

І тут виникає питання: а як можна чисельно визначити ступінь статистичної надмірності в заданій цифровій послідовності? Адже тоді, знаючи цю характеристику, можна було б визначати максимально можливу ступінь стиснення цієї послідовності і порівнювати конкретний алгоритм з цією нижньою межею, тобто можна було б об'єктивно оцінювати якість роботи алгоритму стиснення. Виявляється, що такою величиною є ентропія, яка визначає мінімальне число біт, необхідне для подання заданої послідовності чисел з подальшою можливістю повного відновлення інформації.

У 1948 р співробітник лабораторії Bell Labs Клод Шеннон показав, що мінімальне число біт, який необхідно витратити для представлення одного символу тієї чи іншої інформації можна знайти за допомогою такої формули:

$$H = -\sum_{i=1}^{N} p_i \log p_i \,, \tag{2.6}$$

де p_i - частота (ймовірність) появи i-го числа в послідовності; N - число унікальних чисел в послідовності. Наприклад, використовуючи дану формулу до нашої послідовності чисел, можна визначити, що число унікальних символів дорівнює 5 – це a_1, a_2, a_3, a_4, a_5. Частота появи цифр дорівнює p_1=0,4, p_2=0,2, p_3=0,2, p_4=0,1, p_5=0,1. В результаті мінімальне число біт для представлення

одного символа в такій послідовності визначається таким чином: H= =-0,4·$\log_2$0,4-2·0,2·$\log_2$0,2-2·0,1·$\log_2$0,1≈2,1 біт/символ.

Порівнюючи отриманий результат з раніше отриманим для кодів Хаффмана (2,2 біт/символ), можна побачити, що коди Хаффмана опинилися більш близькими до нижньої границі, ніж рівномірний код, якому необхідно 3 біти/символ.

Також можна порахувати і степінь стиску послідовності. Якщо рівномірний код буде витрачати 3 біти на символ, то потенціальний стиск буде дорівнювати k=3/2,1=1,4 рази, а з допомогою нерівномірних кодів Хаффмана отримаєм k=3/2,2=1,36 разів.

Отже, представлений алгоритм стиску не являється найкращим. Крім того, з допомогою ентропії можна показати, що коли послідовність містить випадковий набір чисел (тобто, не має закономірностей), то ймовірності $p_i \to 1/N$, а можливість стиску $k \to 1$, тобто випадковий набір даних стиснути неможливо. Це положення, зокрема, говорить про те, що якщо один раз до даних був використаний хороший алгоритм стиску, наприклад, zip, то повторний стиск цим або іншим алгоритмом не приведе до кращих результатів, а, швидше, навпаки. Таким чином, на виході будь-якого хорошого алгоритму стиску буде отримуватися майже випадковий набір даних.

2.5 Опис запропонованого алгоритму

Метод, подібно до арифметичного кодування, заснований на ідеї перетворення вхідного потоку в число з плаваючою точкою з інтервалу [0, 1). Кожен символ, який поступив на обробку, зменшує цей інтервал пропорційно ймовірності своєї появи. У міру надходження чергового символу інтервал зменшується, а число біт, що представляють цей інтервал - збільшується.

Зауважимо, що спочатку довжина інтервалу дорівнює 1. Відомо, що сума всіх ймовірностей появи символів алфавіту також дорівнює 1. Нехай N - кількість символів алфавіту, p_i - ймовірність появи в тексті символу з порядковим номером i. Тоді i-му символу ставиться у відповідність такий інтервал:

$$\left[\sum_{k=0}^{i-1} p_k, \sum_{k=0}^{i} p_k\right), i = 1,..., N, p_0 = 0. \quad (2.7)$$

Якщо ймовірності появи символів в тексті р самого початку не відомі, то можна скористатися якою-небудь їх оцінкою (наприклад, для тексту на природній мові можна взяти середньостатистичні ймовірності появи символів для даної мови). Вважаємо, що перед початком роботи алгоритму нам доступний весь текст, звідки можна отримати точні ймовірності всіх символів (p_i = кількість входжень i-го символу/загальна кількість символів).

Якщо першим на вхід надходить символ з порядковим номером k в алфавіті, то замість вихідного беремо k-ий інтервал і ділимо його на N частин в тих же пропорціях, що і вихідний.

Потім для побудованого інтервалу знаходиться число, що належить його внутрішній частині і дорівнює цілому числу, поділеному на мінімально можливий позитивний цілий степінь двійки.

Це число і буде кодом для даної послідовності. З нього можна однозначно відновити вихідну послідовність символів. Оскільки всі можливі конкретні коди – це числа з інтервалу [0, 1), то з міркувань економії можна відкидати початковий нуль і десяткову точку, але потрібен ще один спеціальний код-маркер, який сигналізує про кінець повідомлення.

Пояснимо все вище сказане на прикладі. Розглянемо текст "compressor" алфавіту {c, e, m, o, p, r, s}. Будуємо таблицю значень ймовірностей та інтервалів.

І кодувальнику, і декодувальнику відомо, що на самому початку інтервал [0, 1). Після перегляду першого символу "c", кодувальник звужує інтервал до [0.0, 0.1), який модель виділяє цьому символу. Другий символ "o" звузить цей новий інтервал до проміжка [0.3, 0.5). Аналогічно можна отримати результат представлений в таблиці 2.6.

Таблиця 2.5 – Значення ймовірностей та інтервалів

Символ	Ймовірність	Інтервал
c	0,1	[0,0;0,1)
e	0,1	[0,1;0,2)
m	0,1	[0,2;0,3)
o	0,2	[0,3;0,5)
p	0,1	[0,5;0,6)
r	0,2	[0,6;0,8)
s	0,2	[0,8;1,0)

Таблиця 2.6 – Межі кодування символів

Символ	Нижня межа	Верхня межа
Початок	0,0	1,0
c	0,0	0,1
o	0,03	0,05
m	0,034	0,036
p	0,0350	0,0352
r	0,03512	0,03516
e	0,035124	0,035128
s	0,0351272	0,0351280
s	0,03512764	0,03512800
o	0,035123748	0,035127820
r	0,0351239912	0,0351239056

Отже, текст закодований інтервалом [0.0351239912, 0.0351239056). Число, яке характеризує цей інтервал – 2357387/67108864. Оскільки $67108864=2^{26}$, тобто вихідний текст можна закодувати за допомогою 26 біт. Нехай декодувальник знає про текст лише остаточний інтервал [0.0351239912, 0.0351239056). Він зразу ж розуміє, що перший закодований символ є "c", оскільки підсумковий інтервал цілком лежить в інтервалі, виділеному моделлю цьому символу згідно таблиці ймовірностей. Тепер повторюються дії кодувальника: на початку [0.0, 1.0), після перегляду "c" - [0.0, 0.1). Звідси зрозуміло, що другий символ - "o", оскільку це приводить до інтервала [0.03, 0.05), який повністю вміщає підсумковий інтервал [0.0351239912, 0.0351239056). Продовжуючи таким же чином, декодувальник відновлює весь текст.

Однак, щоб завершити процес, декодувальнику треба вчасно розпізнати кінець текста. Для цього потрібно позначити завершення кожного тексту спеціальним символом EOF, відомим і кодувальнику, і декодувальнику. Коли декодувальник зустрічає цей символ, він припиняє свій процес.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ПОРІВНЯЛЬНИЙ АНАЛІЗ ЗАПРОПОНОВАНОГО АЛГОРИТМУ АРИФМЕТИЧНОГО КОДУВАННЯ

3.1 Обгрунтування вибору інструментальних засобів

Сучасні програмно-інструментальні засоби розробки програмного забезпечення характеризуються великою різноманітністю характеристик. Так, в даний час інструментальні засоби дозволяють:

- базуючись на стандартних компонентах, створювати інтерфейс програми в залежності від стану системи передавати управління різним процесам;

- створювати бази даних і оболонки для баз даних;

- виконувати коректну обробку виняткових ситуацій, що дозволяє підвищити надійність програм;

Сучасні засоби розробки характеризуються такими параметрами:

- підтримка об'єктно-орієнтованого стилю програмування;

- можливість використання CASE-технологій для проектування розроблюваної системи, використання візуальних компонент для наочного проектування інтерфейсу;

- наявність візуальної технології розробки інтерфейсу;

- можливість використання алгоритмів реляційної алгебри для керування базами даних;

- надання засобів синхронізації і контролю версій складових частин проекту (ці засоби використовуються при розробці програмного забезпечення групами програмістів);

- створення інсталяційних пакетів для поширення розробленого програмного забезпечення.

При створенні прототипу програмного забезпечення головними критеріями вибору програмно-інструментальних засобів розробки є:

- швидкість розробки додатків;

- зручність використання;

- можливість швидкого внесення змін до програми.

Забезпечити мінімальний час розробки можна тільки при виконанні цих умов. Виходячи з наведених вимог, виділимо наступні характеристики засобів розробки програмного забезпечення:

- вартість IDE;

- невисока потреба ресурсів;

- наочність розробки інтерфейсу;

- надаються можливості роботи з базами даних;

- швидкість роботи розробленого програмного забезпечення;

- обробка виняткових ситуацій;

- час створення розробленого програмного забезпечення;

- зручність експлуатації;

- засоби контролю версій складових частин проекту;

- наявність зручної довідкової системи.

Для вибору інструментального засобу скористаємося методом варіантних мереж. Цей метод призначений для вибору найкращого варіанту з декількох запропонованих і складається з наступних етапів:

- визначення критеріїв, за якими буде проведене їх порівняння;

- ступінь їх важливості;

- кожен варіант оцінюється по отриманому переліку критеріїв (виходить числове значення - оцінка);

- знаходження загальної кількості балів для кожного з варіантів (можна враховувати важливість критеріїв).

Для вирішення поставленого завдання потрібно буде використовувати перелік характеристик, наведений вище. Кожну характеристику будемо оцінювати балом в діапазоні [1..10], а також ваговим коефіцієнтом в тому ж діапазоні. Вибір будемо проводити з таких поширених програмно-

інструментальних засобів розробки програм, як Java Eclipse, Borland Delphi 7, Microsoft VC ++ 6. Кращим буде той варіант, який набере максимальну кількість балів. Вибір засобів розробки методом варіантних мереж представлений в таблиці 3.1.

Таблиця 3.1 – Порівняння засобів розробки методом варіантних мереж

Характеристика засобів розробки	Ва-га	Оцінка засобів розробки		
		Java Eclipse	Borland Delphi7	Microsoft VC++ 6
Мінімальна вартість IDE	7	10	5	5
Невисока потреба ресурсів	6	7	8	8
Наочність розробки інтерфейса	5	5	9	6
Швидкість роботи розробленого програмного забезпечення	8	7	8	9
Обробка виняткових ситуацій	8	9	8	8
Мінімальний час створення розробленого програмного забезпечення	8	5	9	5
Зручність експлуатації	7	8	8	6
Наявність зручної довідкової системи	5	6	8	9
Сума		391	424	376

Отже, в результаті використання описаного методу варіантних мереж було встановлено, що найкращим програмно-інструментальним засобом з точки зору розробника програм в даному випадку являється середовище Borland Delphi7.

3.2 Програмна реалізація запропонованого алгоритму арифметичного кодування

На рисунку 3.1 представлена блок-схема розробленого алгоритму арифметичного кодування.

Для кодування тексту на вхід подається масив інтервалів frequency[0..*n*] (*n* – число символів алфавіту), які ділять відрізок [0, 1] відповідно частотам входженя символів в текст. Нижче наведено фрагмент тексту програми для кодування:

```
procedure encode_text(frequency);
begin
low = 0.0;
high = 1.0;
repeat
symbol = next_symbol();
range = high - low;
high = low + range *frequency[symbol];
low = low + range *frequency[symbol - 1];
until (symbol == EOF);
end.
```

В рядках 3 - 4 відбувається ініціалізація стартового відрізка. В рядках 5-10 алгоритм послідовно обробляє кожний символ текста. В рядках 5 і 10 викликається процедура next_symbol, яка повертає наступний символ декодованого текста. Змінна symbol зберігає в собі номер символу в алфавіті. З врахуванням частоти входження конкретного символу змінюється робочий інтервал. По завершенню обробки всіх символів текста отримається шуканий інтервал - [low, high).

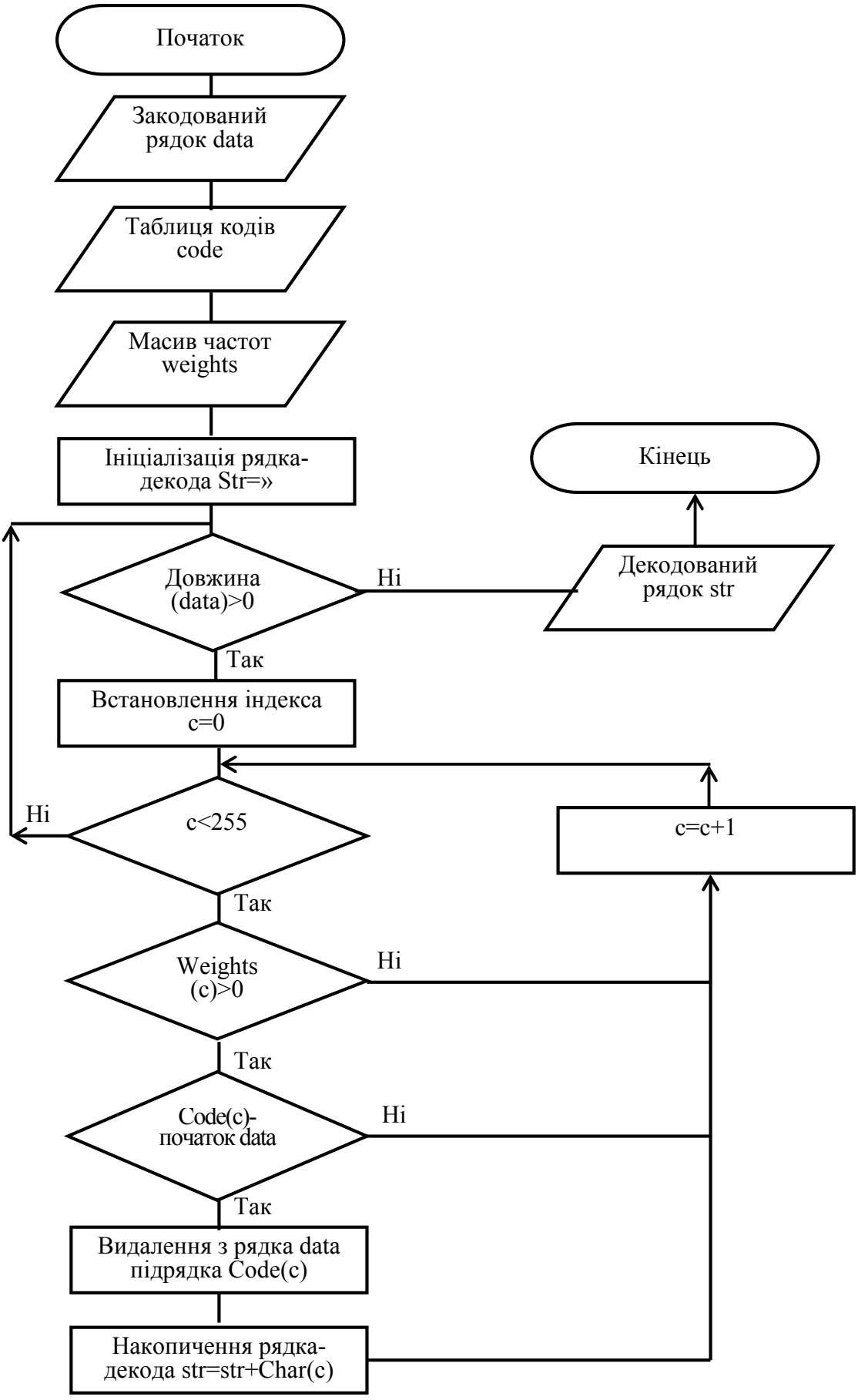

Рисунок 3.1 - Блок-схема розробленого алгоритму арифметичного кодування

При декодуванні тексту на вхід подається число value, яке було отримане в результаті кодування, а також масив frequency. Нижче наведено фрагмент тексту програми для декодування:

```
procedure decode_text (value, frequency);
begin
low = 0.0;
high = 1.0;
repeat
repeat
symbol++;
until (frequency[symbol -1] <= (value - low)/(high - low)) and((value - low)/(high - low) < frequency[symbol]);
range = high – low;
high = low + range *cum_freq [symbol - 1];
low = low + range *cum_freq [symbol ];
print(symbol);
until (symbol == EOF);
end.
```

У рядках 3 - 4 відбувається ініціалізація роботи на стартовому відрізку. У циклі 5 - 13 відбувається декодування символу за символом підряд, до мітки, яка визначає кінець тексту.

У циклі, визначеному рядками 6 - 8 алгоритм за допомогою перебору алфавіту знаходить символ, який потрапляє в конкретний інтервал. У рядках 9 - 12 звужується інтервал для наступного кроку і виводиться перекодований символ.

На рисунку 3.2 представлено головне вікно програмної реалізації для арифметичного кодування.

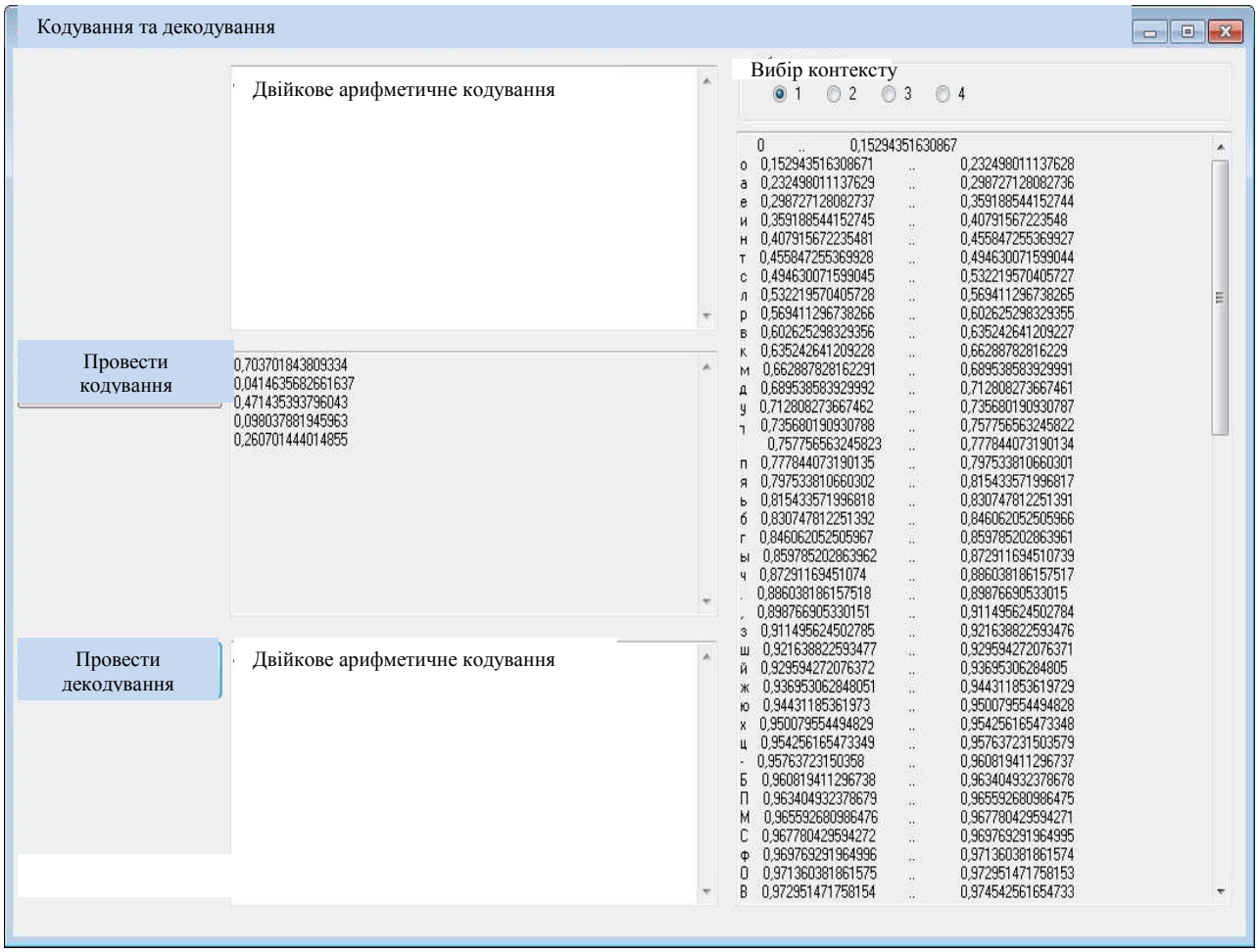

Рисунок 3.2 – Головне вікно програми для арифметичного кодування

3.3 Порівняльний аналіз алгоритмів арифметичного кодування

На відміну від префіксного, арифметичне кодування дозволяє генерувати коди як цілої, так і нецілої довжини. Будучи теоретично оптимальним методом, арифметичне кодування перевершує в ефективності префіксне кодування. Воно також випереджає префіксне кодування в швидкості побудови системи кодів, однак через підвищену складність нерідко помітно поступається в швидкості самого кодування (мається на увазі процес генерації кодової послідовності).

Незважаючи на деякі істотні переваги арифметичного кодування, даний метод до останнього часу був недостатньо поширений на практиці. На

сьогоднішній день в більшості комерційних додатків для побудови системи кодів змінної довжини використовується кодування Хаффмана, що є найкращою з точки зору ефективності реалізацією префіксного кодування. Арифметичне кодування застосовується лише в тих випадках, коли потрібно добитися максимально можливої якості інформаційного уявлення і коли швидкість роботи не має вирішального значення.

Розглянемо це питання докладніше. Для цього порівняємо найбільш популярний метод статистичного кодування інформації - алгоритм Хаффмана та запропонований алгоритм арифметичного кодування.

На початковому етапі роботи в алгоритмі Хаффмана кожному символу інформаційного алфавіту ставиться у відповідність вага, рівна ймовірності появи даного символу в тексті. Символи поміщаються в список, який сортується по спаданню їх ймовірностей. На кожному кроці алгоритму два останніх елемента списку об'єднуються в новий елемент, який потім поміщається в список замість двох об'єднаних.

Новому елементу списку ставиться у відповідність вага, яка дорівнює сумі ваг попередніх елементів. Кожна ітерація закінчується упорядкуванням отриманого нового списку, який завжди містить на один елемент менше, ніж старий список.

Паралельно з роботою зазначеної процедури здійснюється послідовна побудова кодового дерева. На кожному кроці алгоритму будь-якого елементу списку відповідає кореневий вузол бінарного дерева, що складається з вершин, які відповідають елементам, об'єднанням яких був отриманий даний елемент. При об'єднанні двох елементів списку відбувається об'єднання відповідних дерев в одне нове бінарне дерево, в якому кореневий вузол відповідає новому елементу, який поміщається в список, а заміщеним елементам списку відповідають дочірні вузли цього кореневого вузла. Алгоритм завершує роботу, коли в списку залишається один елемент, відповідний кореневому вузлу побудованого бінарного дерева. Це дерево називається деревом Хаффмана.

Система префіксних кодів може бути отримана шляхом присвоювання конкретних двійкових значень ребрам цього дерева.

На рисунку 3.3 наведений приклад роботи алгоритма Хаффмана для тексту “abcbb” з алфавітом {a, b, c}. Для елементів a, b та c їх ймовірності появи відповідно дорівнюють 0.2, 0.6 та 0.2.

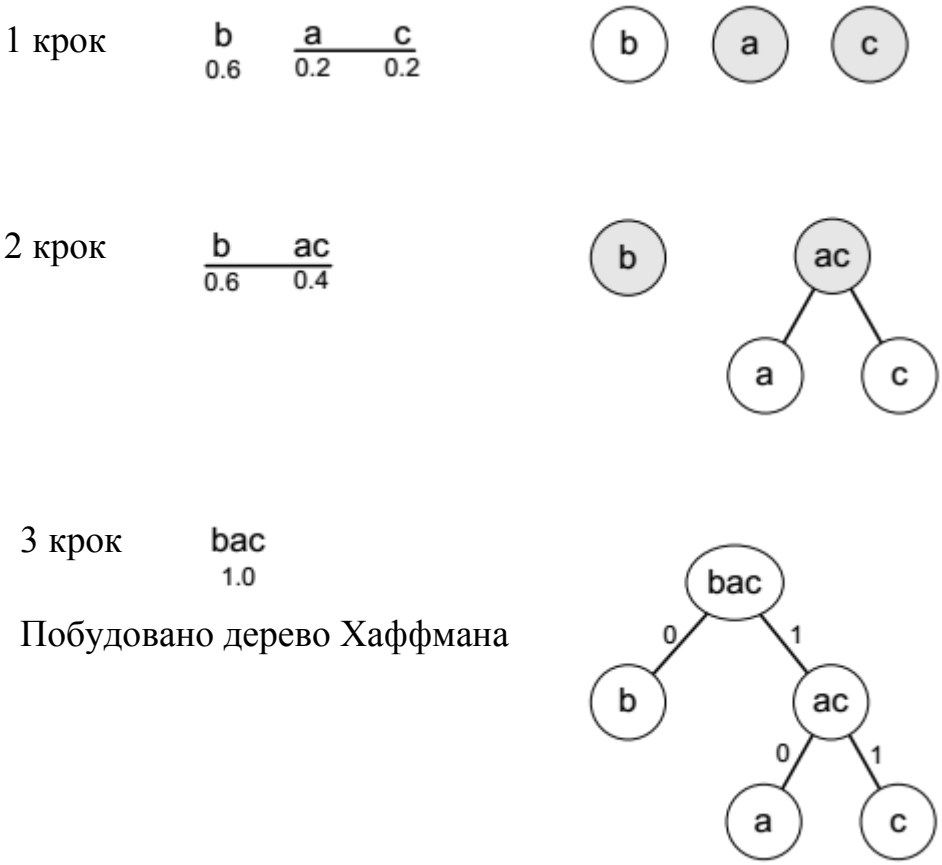

Рисунок 3.3 – Побудова дерева Хаффмана

Згідно побудованого дерева Хаффмана для кодування вихідного повідомлення буде потрібна послідовність з 7 бітів: "1001100" (a - "10", b - "0", c - "11").

Для порівняння введемо наступну величину: будемо характеризувати метод кодування числом *ML* (*S*), де *S* - вихідне повідомлення. *ML* (*S*) дорівнює середньому числу біт, витрачених на кодування кожного символу повідомлення обраним методом стиснення, тобто виражається наступною формулою:

$$ML(S) = L(S') / L(S), \qquad (3.1)$$

де $L(S')$ - кількість біт, необхідна для подання закодованого повідомлення;

$L(S)$ - кількість біт, необхідна для подання вихідного повідомлення.

З теорії, основи якої були закладені К. Шенноном, слідує, що в системі подання інформації з основою m символу a_k, ймовірність появи якого дорівнює $p(a_k)$, оптимально і, що особливо важливо, теоретично допустимо ставити у відповідність код довжиною: $-\log_m p(a_k)$.

З цього твердження, зокрема, випливає, що символ з високою ймовірністю повинен кодуватися кількома бітами, тоді як малоймовірний вимагає багатьох бітів. Також дана формула показує, чому арифметичне кодування в загальному випадку перевершує по ефективності префіксні методи, які, на відміну від першого, кодують символи за допомогою цілої кількості біт, що знижує ступінь стиснення і зводить нанівець точні передбачення ймовірностей. У свою чергу, метод арифметичного кодування забезпечує теоретично необмежену точність.

Розглянемо наступний приклад. Закодуєм слово з алфавіту {0, 1} з використанням методу арифметичного кодування, і порівняємо показник *ML* з аналогічним показником для методу Хаффмана.

Нехай дано текст S = 1110. В таблиці 3.2 наведено розподіли ймовірностей та інтервалів.

Таблиця 3.2 - Розподіли ймовірностей та інтервалів

Символ	Ймовірність	Інтервал
1	3/4	[0,0; 3/4)
0	1/4	[3/4;1,0)

Використаємо метод арифметичного кодування і отримаємо межі для кожного символу, які представлені в таблиці 3.3.

Таблиця 3.3 – Межі кодування символів

Символ	Нижня межа	Верхня межа
Початок	0,0	1,0
1	0,0	3/4
1	0,0	9/16
1	0,0	27/64
0	81/256	27/64

Тоді для кодування нам підійде значення 3/8. У двійковому коді - це 0.011. Разом величина *ML*(*S*) = 3 біти/4 символи = 0,75.

Для кодування можна було б вибрати і інше число, наприклад, 81/256. Тоді довжина коду дорівнювала б не трьом, а восьми бітам, так як 81/256 в двійковій формі представляється як 0.01010001. Ясно, що в цьому випадку отримали б вкрай неефективне інформаційне представлення.

При кодуванні методом Хаффмана і на 0, і на 1 доведеться витрачати не менше одного біта, тобо *ML*(*S*) = 4 біти/4 символи = 1.

Отже, середня кількість біт на одиницю інформації для арифметичного кодування істотно менша, ніж для кодування методом Хаффмана. Показаний приклад демонструє переваги арифметичного кодування. Крім того, його властивістю є те, що ефективність кодування з ростом обсягу даних лише збільшується, на відміну від інших методів, ефективність яких незмінна при зміні вхідних параметрів.

3.4 Адаптивний алгоритм арифметичного кодування

Для арифметичного кодування також можна запропонувати адаптивний алгоритм, тобто алгоритм, який при кожному зіставленні символу коду змінює

також внутрішній хід обчислень так, що наступного разу цьому ж символу може бути зіставлений інший код, тобто відбувається адаптація алгоритму до символів, які надходять на кодування. При декодуванні відбувається аналогічний процес. Необхідність застосування адаптивного алгоритму виникає в тому випадку, якщо імовірнісні оцінки для символів повідомлення невідомі до початку роботи алгоритму.

Побудова арифметичного коду для послідовності символів із заданої множини можна реалізувати наступним алгоритмом. Кожному символу зіставляється його вага, спочатку вага для всіх дорівнює 1. Всі символи розташовуються в природному порядку, наприклад по зростанню. Імовірність кожного символу встановлюється рівною його вазі, поділеній на сумарну вагу всіх символів. Після отримання чергового символу і побудови інтервалу для нього вага цього символу збільшується на 1. Для того, щоб забезпечити зупинку алгоритму розпакування, на початку стиснутого повідомлення, треба виставити його довжину або ввести додатковий символ-маркер.

Розглянемо приклад: нехай необхідно закодувати слова "ABBACD".

На першому кроці ваги всіх символів рівні одиниці. Поточний інтервал - [0,1]. Першим кодується символ A. Так як символів 4 і їх ваги однакові, то береться чверть від вихідного відрізка. Виходить інтервал [0, 1/4]. Символ A оброблений, тому його вага збільшується на одиницю. Загальна вага так само збільшилася на 1. Наступним кодується символ B. Поточний інтервал розбивається відповідно відношенням ваг кожного символу до ваги всіх символів. Вибирається інтервал, відповідний символу B - [1/10, 3/20]. Збільшується вага символу B на одиницю і так далі. Усі кроки докладно описані в таблиці 3.4.

Оскільки $1979/16384=1979/2^{14}$, то звідси отримується, що вихідне повідомлення можна представити двійковим числом $0.00011110111011_2 \in [1826/15120, 1827/15120]$. Таким чином, повідомлення закодуване за допомогою 14 бітів і ML = 2,3 біт/символ.

Таблиця 3.4 – Кроки кодування адаптивним алгоритмом

Вага символу				Загальна вага	Кодована буква	Поточна довжина проміжку	Отриманий інтервал
A	B	C	D				
1	1	1	1	4	A	1	[0,1/4)
2	1	1	1	5	B	1/4	[1/10,3/20)
2	2	1	1	6	B	1/20	[7/60,8/60)
2	3	1	1	7	A	1/60	[7/60,51/420)
3	3	1	1	8	C	1/210	[202/1680,203/1680)
3	3	2	1	9	D	1/1680	[1826/15120,1827/15120)

Декодування відбувається наступним чином: на кожному кроці визначається інтервал, що містить даний код - з цього інтервалу однозначно задається вихідний символ повідомлення. Потім з поточного коду віднімається нижня межа інтервалу, а отримана різниця ділиться на довжину цього ж інтервалу. Отримане число вважається новим поточним значенням коду.

Отримання маркера кінця або заданого перед початком роботи алгоритму числа символів означає завершення роботи.

Приклад: розпакуємо код 00011110111011, знаючи множину символів, з яких складалося вихідне повідомлення. Отже, $00011110111011_2 = 1979/16384$.

Результати розрахунків наведені в таблиці 3.5.

Таким чином, з коду 00011110111011_2 відновлено вихідне повідомлення (ABBACD).

3.5 Кодування довжин повторень

Кодування довжин ділянок (або повторень) може бути достатньо ефективним при стисканні двійкових даних, наприклад, чорно-білих

факсимільних зображень, чорно-білих зображень, що містять багато прямих ліній і однорідних ділянок, схем і т.п. Кодування довжин повторень є одним з елементів відомого алгоритму стиснення зображень JPEG.

Таблиця 3.5 - Кроки декодування адаптивним алгоритмом

Вага символу				Число-код і його інтервал	Декодований символ	Довжина інтервалу
A	B	C	D			
1	1	1	1	1979/16384∈[0,1/4]	A	1/4
2	1	1	1	1979/4096∈[2/5,3/5]	B	1/5
2	2	1	1	1703/4096∈[1/3,2/3]	B	1/3
2	3	1	1	1013/4096∈[0,2/7]	A	2/7
3	3	1	1	7091/8192∈[6/8,7/8]	C	1/8
3	3	2	1	7576/8192∈[8/9,1]	D	1/8

Ідея стиснення даних на основі кодування довжин повторень полягає в тому, що замість кодування власне даних піддаються кодуванню числа, що відповідають довжинам ділянок, на яких дані зберігають незмінне значення.

Припустимо, що потрібно закодувати двійкове (двокольорове) зображення розміром 8 х 8 елементів, наведене на рисунку 3.4.

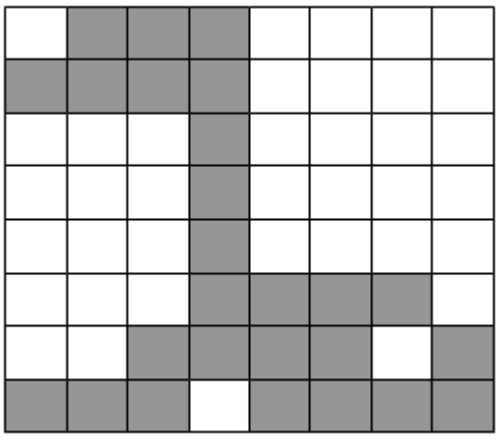

Рисунок 3.4 – Чорно-біле зображення

Проскануєм це зображення по рядках (двом кольорам на зображенні будуть відповідати 0 і 1), в результаті отримаємо двійковий вектор даних X = (011100001111000000010000000100000001000000011110001111010111101111) довжиною 64 біти (швидкість вихідного коду складає 1 біт на елемент зображення).

Виділимо у векторі X ділянки, на яких дані зберігають незмінне значення, і визначимо їх довжини. Результуюча послідовність довжин ділянок - додатних цілих чисел, відповідних вихідному вектору даних X, - буде мати вигляд r=(1, 3, 4, 4, 7, 1, 7, 1, 7, 1, 7, 4, 3, 4, 1, 4, 1, 4).

Тепер цю послідовність, в якій помітна певна повторюваність (одиниць і четвірок набагато більше, ніж інших символів), можна закодувати арифметичним кодом, що має відповідну таблицю кодування (таблиця 3.6).

Таблиця 3.6 – Відповідність кодових слів довжинам ділянок

Довжина ділянки	Кодове слово
4	0
1	10
7	110
3	111

Для того, щоб вказати, що кодована послідовність починається з нуля, потрібно додати на початку кодового слова префіксний символ 0. У результаті отримаємо кодове слово $B(r)$ = (0100011010110101101011001110100100) довжиною в 34 біта, тобто результуюча швидкість коду R складе 34/64, або трохи більше 0,5 біта на елемент зображення. При стисненні зображень більшого розміру, які містять множину елементів, що повторюються, ефективність стиснення може виявитися істотно вищою.

Нижче наведено інший приклад використання кодування довжин повторень, коли в цифрових даних зустрічаються ділянки з великою кількістю нульових значень. Кожного разу, коли в потоці даних зустрічається "нуль", то він кодується двома числами. Перше - 0, що є прапорцем початку кодування

довжини потоку нулів, і друге - кількість нулів у черговій групі. Якщо середнє число нулів в групі більше двох, то буде мати місце стискання. З іншого боку, велика кількість окремих нулів може призвести навіть до збільшення розміру кодованого файлу. Нехай маємо такі послідовності:

7 8 54 0 0 0 97 5 16 0 45 23 0 0 0 0 0 3 67 0 0 8 ...;

7 8 54 0 3 97 5 16 0 1 45 23 0 5 3 67 0 2 8 ... .

Відповідність між ними представлена на рисунку 3.5.

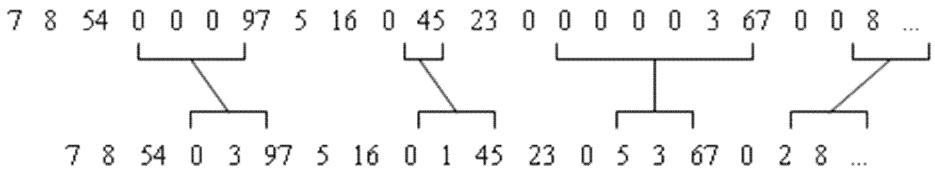

Рисунок 3.5 – Відповідність між послідовностями

Даний метод також зменшує надлишковість кодування.

3.6 Ефективність стиску

При арифметичному кодуванні тексту кількість біт у закодованому рядку рівна ентропії цього тексту відносно використаної для кодування моделі.

Три фактора викликають погіршення цієї характеристики:

- витрати на завершения тексту;
- використання арифметики скінченної точності;
- масштабування лічильників, щоб їх сума не перевищувати максимальної частоти.

Однак вплив кожного з них незначний. Арифметичне кодування повинно досилати додаткові біти в кінець кожного тексту, тобто витрачати додаткові зусилля на завершения тексту. Для ліквідації неясності з останнім символом

процедура done_encoding () посилає два біти. У разі, коли перед кодування потік бітів повинен блокуватися у 8-бітові символи, буде необхідно заокруглюватися до кінця блоку. Таке комбінування може додатково потребувати 9 бітів.

Витрати при використанні арифметики скінченної точності проявляються у скороченні залишків при розподілі. Це видно при порівнянні з теоретичною ентропією, яка виводить частоти з лічильників, які так само масштабовані, як і при кодування. Тут затрат незначні - порядка 10^{-4} біт/символ.

Додаткові затрати на масштабування лічильників почасти більші, але все одно дуже малі. Для коротких текстів (менших 2^{14} байт) їх немає. Але навіть з текстами в 10^5 - 10^6 байтів накладні витрати, підраховані експериментально, складають менше 0.25% від рядка, який кодується.

Адаптивна модель при загрозі перевищення загальною сумою накопичених частот значення Max_frequency зменшує всі лічильники. Це приводить до того, що зважувати останні події важче, ніж попередні. Тобто показники мають тенденцію прослідковувати зміни у вхідній послідовності, які можуть бути дуже корисними. Відомі випадки, коли обмеження лічильників до 6-7 бітів давало кращі результату, ніж підвищення точності арифметики). Звичайно, це залежить від джерела, до якого використовується модель.

Для усунення часових витрат, таких як виклики деяких процедур, і деякої простої оптимізації у розробленій версії програми на мові С були зроблені наступні зміни:

- процедури input_bit (), output_bit () і bit_plus_follow () були переведені у макроси, чим уникаються витрати на виклик процедур;
- величини, які часто використовуються, були поміщені в регістрові змінні;
- множення на два були замінені доповненнями ("+ ="");
- індексний доступ до масиву в циклах був замінений операцією з покажчиками.

Така середньо оптимізована реалізація на С показала час виконання в 214/252 мкс на вхідний байт для кодування/декодування 100.000 байтів англійського тексту на VAX-11/780, як показано в таблиці 3.6. Там же дані результати для тієї ж програми на Apple Macintosh і SUN-3/75.

Таблиця 3.6 - Результати кодування та декодування 100000-байтових файлів

		VAX-11/780		Macintosh 512K		SUN-3/75	
	Файл (байти)	Код. (mkc)	Дек. (mkc)	Код. (mkc)	Дек. (mkc)	Код. (mkc)	Дек. (mkc)
Середньооптимізована реалізація на С							
Текстові файли	57718	214	262	687	881	98	121
С-програми	62991	230	288	729	950	105	131
Об’єктні файли VAX	73501	313	406	950	1334	145	190
Алфавіт	59292	223	277	719	942	105	130
Асиметричні показники	12092	143	170	507	645	70	85
Оптимізована реалізація на асемблері							
Текстові файли	57718	104	135	194	243	46	58
С-програми	62991	109	151	208	266	51	65
Об’єктні файли VAX	73501	158	241	280	402	75	107
Алфавіт	59292	105	145	204	264	51	65
Асиметричні показники	12092	63	81	126	160	28	36

Як можна бачити, кодування програмі на С однієї і тієї ж довжини всюди здійснюється трохи довше, виключаючи тільки виконавчі об'єктні файли. У тестовий набір були включені два штучних файли. 100000-байтний "алфавіт" складається з 26-літерного алфавіту, який повторяється. Асиметричні показники містять 10000 копій рядка "aaaabaaaac". Ці приклади показують, що файли можуть бути стиснуті щільніше, ніж 1 біт/символ (12092-х байтний вихід дорівнює 93736 бітам).

Подальше зниження в 2 раза часових затрат може бути досягнуто перепрограмуванням наведеної програми на мову асемблера. Ретельно оптимізована версія програми (адаптивна модель) була реалізована для VAX і M68000. Регістри використовувалися повністю, а code_value було взято розміром в 16 біт, що дозволило прискорити деякі важливі операції порівняння і спростити віднімання Half.

Часові характеристики асемблерної реалізації на VAX-11/780 дані в таблиці 3.7. Вони були отримані при використанні можливостей профіля UNIX. Цей механізм створює гістограму значень програмного лічильника переривання годин реального часу і страждає від статистичної варіантності та деяких системних помилок). Обчислення границь відноситься до початкових частин encode_symbol () і decode_symbol (), які містять операції множення і ділення. Зсув бітів - це головний цикл в процедурах кодування і декодування. Обчислення cum в decode_symbol () вимагають множення і ділення, а також виконання подальшого циклу для визначення наступного символу, - це декодування символу. А оновлення моделі відноситься до адаптивної процедури update_model ().

Як і передбачалось, обчислення границь і оновлення моделі вимагають однакової кількості часу і для кодування, і для декодування в межах помилки експерименту. Зсув бітів здійснюється швидше для текстових файлів, ніж для С-програм і об'єктних файлів через краще його стиснення. Додатковий час для декодування в порівнянні з кодуванням виникає через крок "декодування

символу" - циклу, який виконується частіше (у середньому 9 раз, 13 раз і 35 раз відповідно). Це також впливає на час відновлення моделі, тому що пов'язане з кількістю накопичень в лічильниках, значення яких необхідно збільшувати. У гіршому випадку, коли символи розподілені однаково, ці цикли виконуються у середньому 128 разів. Такий стан можна поліпшити, застосувавши дерево зі складнішою організацією, але це сповільнить операції з текстовими файлами.

Таблиця 3.7 – Часові інтервали асемблерної версії VAX-11/780

	Час кодування (мкс)	Час декодування (мкс)
Текстові файли	104	135
Обчислення границь	32	31
Зсув бітів	39	30
Обновлення моделі	29	29
Декодування символу	-	45
Решта	4	0
Сі - програма	109	151
Обчислення границь	30	28
Зсув бітів	42	35
Обновлення моделі	33	36
Декодування символу	-	51
Решта	4	1
ОбъектныйфайлVAX	158	241
Обчислення границь	34	31
Зсув бітів	46	40
Обновлення моделі	75	75
Декодування символу	-	94
Решта	3	1

Результати стиснення вар'юються від 4.8-5.3 біт/символ для коротких англійських текстів (10^3-10^4 байтів) до 4.5-4.7 біт/символ для довгих (10^5-10^6 байтів). Хоча існують і адаптивні техніки Хаффмана, вони все ж відчувають нестачу концептуальної простоти, властивої арифметичному кодуванню. При порівнянні вони виявляються більш повільними. Наприклад, таблиця 3.8 приводить характеристики средньооптимізованої реалізації арифметичного кодування на С та адаптивного кодування Хаффмана з використанням подібної моделі.

Таблиця 3.8 – Порівняння арифметичного кодування та Хаффмана

	Арифм. кодування			Кодування Хаффмана		
	Файл (байти)	Код. (mkc)	Дек. (mkc)	Файл (байти)	Код. (mkc)	Дек. (mkc)
Текстові файли	57718	214	262	57781	550	414
С-програми	62991	230	288	63731	596	441
Об'єктні файли VAX	73501	313	406	76950	822	606
Алфавіт	59292	223	277	60127	598	411
Асиметричні показники	12092	143	170	16257	215	132

Перевірка показує, що арифметичне кодування виконується приблизно в 2 раза швидше. Показники стиснення в деякій мірі кращі в арифметичному кодуванні для всіх тестових файлів. Різниця буде помітною у випадку використання більш складних моделей, які передбачають символи з ймовірностями, залежними від визначених обставин (наприклад, проходження за буквою *q* букви *u*).

ВИСНОВКИ

1. Удосконалено формалізований опис арифметичних кодів, що дозволило обгрунтувати вибір програмного забезпечення для реалізації алгоритмів.

2. Побудовано математичну модель запропонованого алгоритму арифметичного кодування на основі перетворення вхідного потоку в число з плаваючою точкою.

3. Розроблено адаптивний алгоритми арифметичного кодування, що дозволило реалізувати кодування довжин повторень у вхідному потоці інформації.

4. Обґрунтовано підхід та розроблено алгоритм арифметичного кодування, що дозволило застосувати відповідне програмне забезпечення для реалізації запропонованого алгоритму;

5. На основі середовища розробки Borland Delphi7 реалізовано програмне забезпечення запропонованого алгоритму арифметичного кодування, яке продемонструвало переваги його переваги в порівнянні з існуючими.