УДК 681.516.77:519.72

О.М. Солошенко

ЗАСТОСУВАННЯ ШАБЛОНУ ПРОЕКТУВАННЯ COMPOSITE ТА ТЕХНОЛОГІЇ LINQ ДО ЗАДАЧІ ПРЕФІКСНОГО КОДУВАННЯ ТЕКСТУ ЗА ДОПОМОГОЮ АЛГОРИТМУ ХАФФМАНА

The designed methods are the implementation of the object-oriented architecture as the class diagram corresponding to the implementation of the Composite design pattern according to the requirements of the flexible implementation of the Huffman tree, the method of the dynamic iterative Huffman tree building. This tree uses the dynamic collection of the node interface using the Visual C# language and the creation method of the initial set of the leaf nodes using the LINQ technology. The implementation methodologies are the idea of the Composite design pattern for the handling with the tree hierarchical structures, the possibilities of the Visual Studio 2010 integrated development environment and the possibilities of the .NET 4.0 Framework and the LINQ technology within Visual C# programming language. The research results are the object-oriented program code in Visual C# language and the class diagram in UML language that illustrates application of the design pattern and principles of inheritance, abstraction, polymorphism and encapsulation for the implementation of the flexible and scalable object architecture of the Huffman tree. The advantages of the mentioned software architecture, the examples of the organization and the storage of large arrays of the text data with the key for decoding symbol table are presented. This proves practical value of the Shannon lemma in the field of information and coding theory.

Keywords: Huffman algorithm, Huffman coding, object-oriented design, Composite design pattern, LINQ, Visual C#, information and coding theory, prefix encoding, entropy, Shannon lemma.

Вступ

Одним з основних практичних напрямів теорії кодування та інформації є питання мінімізації розміру даних на фізичних носіях інформації, тобто архівація даних, що передбачає забезпечення оптимальної по критерію мінімізації кількості бітів (а в реальній ситуації, мінімальної достатньої кількості байтів по 8 біт), необхідної для цілісного збереження інформації [1]. Ця задача відома як наближення до мінімальної середньої довжини коду. Типовим прикладом є обчислення мінімальної середньої довжини коду як наближення до ентропії тексту. Твердження леми Шеннона про нижню границю полягає у тому, що неможливо досягнути середньої довжини коду символів у тексті нижче ентропії даного тексту [2]. Подання кожного символу тексту відбувається як певна послідовність бітів довжиною L_i.

Алгоритм префіксного кодування полягає у зіставленні листків бінарного дерева символам тексту, а саме: послідовність бітів визначає шлях від вершини дерева до його листка, тобто кожен біт (0 або 1) послідовності визначає рух по дереву в одному з двох можливих напрямків (вліво або вправо), в результаті після досягнення листка дерева, отримується символ, після чого алгоритм переходить назад на вершину дерева для декодування наступного символу. Одним з простих прикладів дерева є, наприклад, дерево кодування тексту з трьох символів такими послідовностями бітів: $a = 00$, $b = 01$, $c = 1$, тоді послідовність "10000011" означає текст "caabc". Найпростіше бінарне дерево префіксного кодування задається такими послідовностями бітів: $A = 0$, $B = 1$. Методом математичної індукції можна довести, що $\sum_{i=1}^{N} 2^{-L_i} = 1$ при префіксному кодуванні. Середня довжина коду обчислюється з використанням ймовірностей – частот символів у кодованому тексті (де частота – частка кількості елементів символу від усього обсягу тексту): $H(\mathbf{L}) = \sum_{i=1}^{N} p_i L_i$. Задача мінімізації середньої довжини коду записується таким чином:

$$\begin{cases} H(\mathbf{L}) = \sum_{i=1}^{N} p_i L_i \to \min, \\ \sum_{i=1}^{N} 2^{-L_i} = 1. \end{cases}$$

Функція Лагранжа записується так: $\Psi(\lambda, \mathbf{L}) = \sum_{i=1}^{N} p_i L_i - \lambda\left(\sum_{i=1}^{N} 2^{-L_i} - 1\right)$, а система рівнянь пошуку вектора оптимальних довжин коду $\mathbf{L}$ має такий вигляд:

$$\begin{cases} \dfrac{\partial \Psi(\lambda, \mathbf{L})}{\partial L_i} = p_i + \lambda 2^{-L_i} \ln 2 = 0 \quad \forall i = 1,..N, \\ 1 - \sum\limits_{i=1}^{N} 2^{-L_i} = 0; \end{cases}$$

$$\begin{cases} L_i = \log_2 \left(-\dfrac{\lambda \ln 2}{p_i} \right) \quad \forall i = 1,..N, \\ 1 - \sum\limits_{i=1}^{N} 2^{-L_i} = 0. \end{cases}$$

Щоб обчислити λ, достатньо просумувати всі однотипні рівності системи: $\sum\limits_{i=1}^{N}(p_i + \lambda 2^{-L_i} \ln 2) = 0$, тоді, враховуючи, що сума ймовірностей дорівнює одиниці $\left(\sum\limits_{i=1}^{N} p_i = 1 \right)$ та суму $\sum\limits_{i=1}^{N} 2^{-L_i} = 1$, очевидно, що $\lambda = -\dfrac{1}{\ln 2}$. Як результат, вектор оптимальної довжини коду записується так: $L_i = -\log_2 p_i$.

Величина глобального мінімуму $H(\mathbf{p}) = -\sum\limits_{i=1}^{N} p_i \log_2 p_i$ називається ентропією тексту [2].

Суть леми Шеннона така: $\forall \mathbf{L} : \sum\limits_{i=1}^{N} p_i L_i \geq H(\mathbf{p})$. Бінарне дерево, що найближче відображає ентропію тексту, називається деревом Хаффмана. Алгоритм побудови дерева Хаффмана полягає у послідовному об'єднанні розрізнених листів та вузлів дерева по два за ознакою мінімальних частот, що забезпечує бінарність дерева; алгоритм завершується збіжністю в одну вершину [2].

Недоліком більшості програмних реалізацій алгоритму Хаффмана мовами третього покоління типу C++ та спеціалізованими мовами типу Matlab [3] є те, що вони базуються на структурах даних типу масивів, списків або матриць [3] без чітко визначеної об'єктно-орієнтованої архітектури, наприклад на процедурному рівні.

Тому актуальним є завдання програмно-архітектурної об'єктно-орієнтованої реалізації алгоритму побудови дерева Хаффмана за допомогою ідеї шаблону проектування Композит (Composite) [4], можливостей технології мови інтегрованих запитів (Language Integrated Query, LINQ), інтерфейсу IComparable, узагальнених типізованих колекцій та інших можливостей Visual C# .NET 4.0 [5] – завдання застосування шаблону проектування Composite до реалізації префіксного кодування тексту.

Статтю присвячено розробленню оптимальної діаграми класів за допомогою UML [5] та відповідного програмного коду мовою Visual C#, а також супровідного коду на Visual C#, що забезпечує динамічний алгоритм побудови дерева Хаффмана.

Постановка задачі

Об'єктами дослідження є алгоритм префіксного кодування Хаффмана, шаблон проектування Composite, можливості мови програмування Visual C#, зокрема технології LINQ.

Предметом дослідження є застосування шаблону проектування Composite та можливостей технології LINQ до задачі префіксного кодування тексту за допомогою алгоритму Хаффмана.

Метою роботи є розроблення таких методів: 1) реалізації об'єктно-орієнтованої архітектури у формі діаграми класів, що відповідає впровадженню шаблона проектування Composite згідно з потребами гнучкої та масштабованої реалізації дерева Хаффмана; 2) динамічної ітеративної побудови дерева Хаффмана засобами платформи Visual C# .NET 4.0 за допомогою динамічної колекції від інтерфейсу вузла; 3) створення початкової множини листових вузлів за допомогою технології LINQ.

Метод реалізації архітектури дерева Хаффмана за допомогою шаблону проектування Composite

У цій статті запропоновано оригінальну архітектуру відношення об'єктно-орієнтованих класів дерева Хаффмана у вигляді зображеної діаграми класів (рис. 1), що відповідає шаблону проектування Composite.

У розробленій архітектурі вводиться абстрактний клас INode, який реалізує стандартний інтерфейс IComparable мови Visual C# (тобто метод порівняння об'єктів CompareTo) з метою реалізації подальшого алгоритму побудови дерева Хаффмана. Клас INode інкапсулює атрибут freq – цілочислову частоту, що характерна як для конкретного символу тексту (листка дерева), так і для множини символів тексту (власне вузла дерева). В цілому абстрактний клас INode відображає вузол дерева, до якого можна віднес-

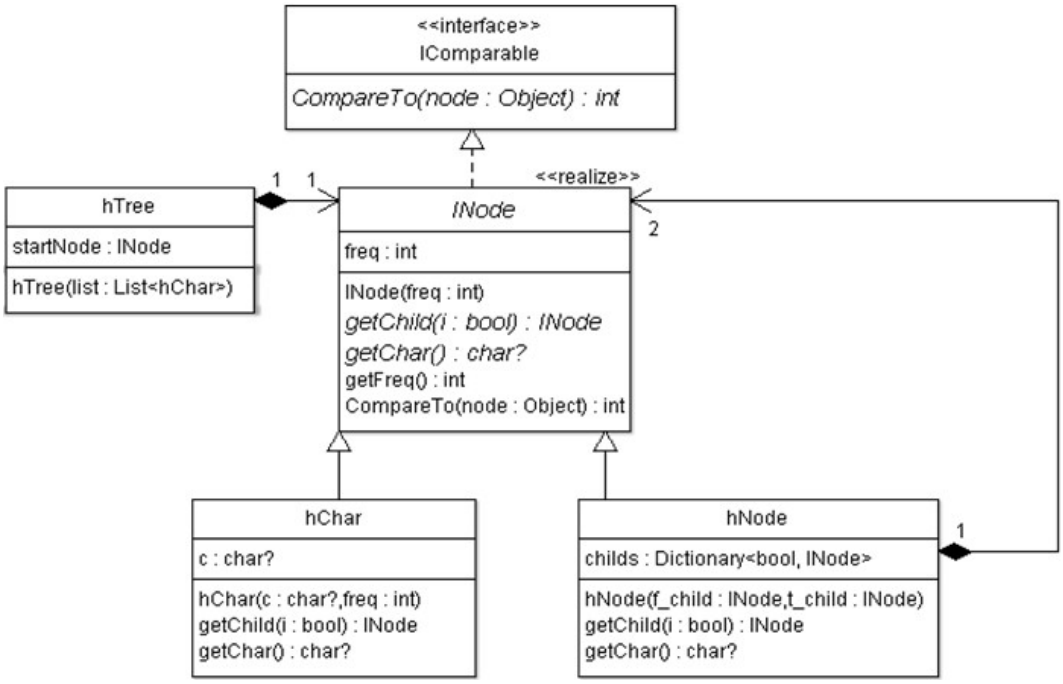

Рис. 1. Діаграма класів дерева Хаффмана

ти й поняття листка дерева, тобто листки – це підмножини вузлів, а саме вузли, які не мають нащадків. Крім конструктора INode(int freq), абстрактний клас INode визначає метод, загальний для всіх вузлів (зокрема й листків бінарного дерева) – метод getFreq(), що повертає значення атрибуту freq. Щодо методів getChild(bool i) та getChar(), то вони оголошені абстрактними. Метод getChar() має сенс лише для листкових вузлів, а метод getChild(bool i) лише для власне вузлів-агрегатів. Метод getChild(bool i) завдяки булевому вхідному параметру накладає дуже важливе обмеження в проектуванні системи – параметр може набувати лише двох значень, що відповідають двом гілкам бінарного дерева. Код абстрактного класу INode такий:

```
public abstract class INode : IComparable
    {
        private int freq = 0;
        public INode(int freq) { this.freq = freq; }
        public abstract INode getChild(bool i);
        public abstract char? getChar();
        public int getFreq() { return freq; }
        public int CompareTo(Object node)
        {
            int res = this.getFreq() - ((INode)node).getFreq();
            return (res < 0) ? -1 : ((res > 0) ? 1 : 0);
        }
    }
```

Конкретний клас листкового вузла hChar наслідує абстрактний клас абстрактного вузла INode, визначаючи два методи: getChar(), що повертає символ, та getChild(bool i), що не має сенсу для листкового вузла, тому повертає порожній вказівник:

```
public class hChar : INode
    {
        private char? c = null;
        public hChar(char? c, int freq)
            : base(freq)
        {
            this.c = c;
        }
        public override INode getChild(bool i) { return null; }
        public override char? getChar() { return c; }
    }
```

Конкретний клас агрегата-вузла hNode наслідує абстрактний клас абстрактного вузла INode, визначаючи два методи: getChild(bool i), що повертає один з двох вкладених вузлів, та

getChar(), що не має сенсу для агрегата-вузла, тому повертає порожній вказівник. Тип словникової колекції Dictionary<bool, INode> дає можливість зберігати лише два вкладені вузли, що відповідають значенням “false” та “true”:

```
public class hNode : INode
    {
        private Dictionary<bool, INode> childs = new Dictionary<bool, INode>();
        public hNode(INode f_child, INode t_child)
            : base(f_child.getFreq() + t_child.getFreq())
        {
            childs.Add(false, f_child);
            childs.Add(true, t_child);
        }
        public override INode getChild(bool i) { return childs[i]; }
        public override char? getChar() { return null; }
    }
```

Конкретний клас hTree є класом-клієнтом, що має атрибут startNode – вказівник на вершину дерева Хаффмана.

Метод побудови дерева Хаффмана за допомогою динамічної колекції від інтерфейсу вузла

Суть пропонованого методу полягає у визначенні конструктора класу hTree дерева Хаффмана від множини листкових елементів – символів List<hChar>. Проміжна динамічна колекція від інтерфейсу абстрактного вузла типу List<INode> використовується для власне ітеративної побудови дерева Хаффмана. Алгоритм починається з перенесення всіх символів у динамічну колекцію. Надалі суть алгоритму описується таким чином: виконувати наступні кроки, доки в динамічній колекції більше одного елемента (тобто поки вершина дерева не досягнута):

- сортування колекції по зростанню згідно з методом CompareTo;
- додавання в кінець колекції нового вузла, що є об’єднанням двох абстрактних вузлів з найменшими частотами;
- видалення двох абстрактних вузлів з найменшими частотами, що були об’єднані в один новий вузол.

По завершенню алгоритму вершиною бінарного дерева Хаффмана вважається єдиний елемент динамічної колекції, який залишився:

```
public class hTree
    {
        public INode startNode { get; private set; }
        public hTree(List<hChar> list) /*конструктор дерева від множини листкових елементів hChar*/
        {
            List<INode> dynList = new List<INode>(); /*динамічна колекція від інтерфейсу вузла*/
            foreach (var r in list) { dynList.Add(r); } /*початкове перенесення символів у колекцію*/
            while (dynList.Count > 1) /*цикл виконується, поки в динамічній колекції не лишиться лише вершина*/
            {
                dynList.Sort(); /*сортування колекції по зростанню згідно з методом CompareTo*/
                dynList.Add(new hNode(dynList[0], dynList[1])); /*додавання в кінець колекції нового вузла*/
                for (int i = 0; i < 2; i++) dynList.RemoveAt(0);/*видалення двох найменших об’єднаних вузлів*/

Console.WriteLine("\n#Iteration:");/*виведення інформації про ітерацію*/
                foreach (var e in dynList) Console.WriteLine(e.getFreq()); /*виведення частот колекції*/

Console.WriteLine("###########");/*завершення виводу інформації про ітерацію*/
            }
            startNode = dynList[0]; /*визначення вершини бінарного дерева Хаффмана*/
        }
        private void prefixPrint(string prefix, INode node)
        {
            if (node.getChar() == null)
            {
                string str0 = prefix + "0";
                string str1 = prefix + "1";
                prefixPrint(str0, node.getChild(false));
```

```
                prefixPrint(str1,
node.getChild(true));
            }
            else
                Console.WriteLine("Symbol:
'{0}'\t{1}", node.getChar(), prefix);
        }
        public void showAll()
        {
            prefixPrint("Code:                0",
startNode.getChild(false));
            prefixPrint("Code:                1",
startNode.getChild(true));
        }
    }
```

Рекурсивний інкапсульований (прихований) метод prefixPrint(string prefix, INode node) дає змогу відкритому методу showAll() надрукувати бінарні коди символів тексту, реалізуючи рекурсивний прохід дерева.

Метод створення початкової множини листкових вузлів за допомогою технології LINQ

Суть пропонованого методу полягає у завантаженні списку елементів типу Nullable<char> з потоку вхідного файлу з подальшим частотним аналізом списку символів за допомогою технології LINQ to Objects [5]. Частотний аналіз полягає у застосуванні агрегатної функції типу count [5, 6] для підрахунку частот (кількості входжень у текст) унікальних елементів списку символів, що є елементом класичної команди "group by" [5, 6]. Надалі метод передбачає перенесення результату виконання групування за допомогою LINQ в окремий список об'єктів конкретного класу листкового вузла hChar, що наслідує абстрактний клас абстрактного вузла INode. Потім, через виклик конструктора класу hTree, метод посилається на описаний вище метод побудови дерева Хаффмана за допомогою динамічної колекції від інтерфейсу вузла (тобто від абстрактного класу абстрактного вузла INode).

Розглянемо організацію базового класу програми Program, що містить статичний метод, що є точкою входу програми – static void Main(string[] args):

```
class Program
    {
        static void Main(string[] args)
        {
            hTree myTree;
            Console.WriteLine("Please enter
TXT-file name:");
            string filename =
Console.ReadLine();
            List<char?> list = new
List<char?>();
            try
            {
                using (StreamReader file = new
StreamReader(filename))
                {
                    while (!file.EndOfStream)
                    {
                        list.Add((char?)file.Read());
                    }
                }
                var freq = from c in list
                           group c by c
                               into gr
                               orderby gr.Key
                               select gr;
                List<hChar> clist = new
List<hChar>();
                Console.WriteLine();
                foreach (var g in freq)
                {
                    Console.WriteLine("Char: '{0}'
\t Freq: {1}", g.Key, g.Count());
                    clist.Add(new hChar(g.Key,
g.Count()));
                }
                myTree = new hTree(clist);
                Console.WriteLine("\nFinal tree
weight: {0}", myTree.startNode.getFreq());
                myTree.showAll();
            }
            catch (FileNotFoundException e) {
Console.WriteLine("File Not Found: " +
e.Message); }
            Console.ReadKey();
        }
    }
```

Стосовно технології LINQ, оператор "var freq = from c in list group c by c into gr orderby gr.Key select gr;" дає можливість виконати класичну SQL-подібну команду GROUP BY [6] засобами LINQ [5] з метою підрахунку частот символів тексту. Щоб отримати ознаки (тут символи) групування в операторі "foreach (var g in freq)" та агреговану кількість (тут частоту) достатньо звернутися до наступних атрибутів

(властивостей) та методів результуючого набору: g.Key та g.Count().

Наведемо приклад побудови дерева Хаффмана, задаючи текстовий файл, що містить рядок “abab abaz”:

```
 Please enter TXT-file name:
C:\New Fun + Work\aspirantura-kpi\PhD article\second_article\Huffman_Application
\myfile.txt

Char: ' '       Freq: 1
Char: 'a'       Freq: 4
Char: 'b'       Freq: 3
Char: 'z'       Freq: 1

#Iteration:
3
4
2
###########

#Iteration:
4
5
###########

#Iteration:
9
###########

Final tree weight: 9
Symbol: 'a'     Code: 0
Symbol: 'z'     Code: 100
Symbol: ' '     Code: 101
Symbol: 'b'     Code: 11
```

Відповідне дерево Хаффмана можна подати у вигляді, наведеному на рис. 2.

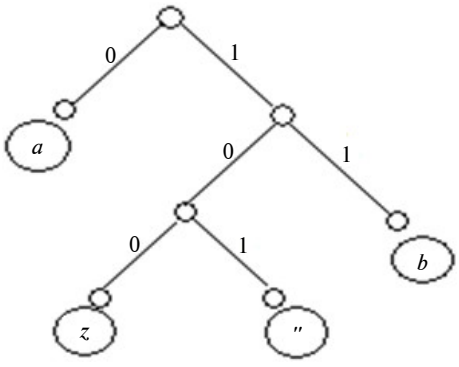

Рис. 2. Дерево Хаффмана для тексту “abab abaz”

Організація збереження закодованого тексту має враховувати збереження двох складових частин:

- послідовність бітів (байтів) у бінарному файлі;
- дерево кодування та декодування (дерево Хаффмана) у вигляді таблиці кодування символів.

Висновки

Запропоновано такі методи:

1) реалізації об’єктно-орієнтованої архітектури у формі діаграми класів, що відповідає впровадженню шаблона проектування Composite згідно з потребами гнучкої та масштабованої реалізації дерева Хаффмана;

2) динамічної ітеративної побудови дерева Хаффмана засобами платформи Visual C# .NET 4.0 за допомогою динамічної колекції від інтерфейсу вузла;

3) створення початкової множини листкових вузлів за допомогою технології LINQ.

Одними з головних істотних переваг проектування та відмінностей від наявних реалізацій для запропонованих наведених вище методів є такі:

- гнучка та масштабована об’єктно-орієнтована архітектура, яка вкладається в рамки класичного шаблону проектування Composite;
- прозорий алгоритм побудови дерева Хаффмана з використанням динамічної колекції елементів, що реалізують елемент узагальненого абстрактного вузла;
- рекурсивний алгоритм проходу дерева Хаффмана з метою отримання кодів символів тексту;
- всеохоплююче використання булевого типу даних з метою введення жорсткого обмеження, яке гарантує бінарність дерева Хаффмана.

Запропонована об’єктно-орієнтована реалізація з використанням шаблону проектування Composite дає можливість вирішувати завдання побудови оптимального дерева кодування – дерева Хаффмана для кодування довільної текстової інформації з метою зменшення фізичного об’єму для збереження тексту та наочного пояснення суті елементів дерева кодування й алгоритму Хаффмана в рамках парадигми об’єктно-орієнтованого програмування.

Перспективи подальших досліджень включають розроблення та реалізацію:

- алгоритму формування бінарного вихідного потоку виводу в файл послідовності бай-

тів, що відповідає послідовності бітів кодів символів згідно з деревом Хаффмана, з розробкою алгоритму коректного завершення файлу, оскільки виникає питання збереження завершального байту, що покриває кількість останніх бітів, рівну остачі від ділення довжини послідовності бітів на вісім;

- методів збереження та обробки метаданих – таблиці кодування символів згідно з деревом Хаффмана;
- алгоритмів обробки вхідного потоку у вигляді послідовності байтів, використовуючи інших вхідний потік, що відповідає за завантаження та розбір метаданих.

Список літератури

1. *Методы* сжатия данных. Устройство архиваторов, сжатие изображений и видео / Д. Ватолин, А. Ратушняк, М. Смирнов и др. – М.: ДИАЛОГ-МИФИ, 2002. – 384 с.
2. *Теорія* інформації та кодування: Навч. посібник / В.Л. Кожевников, А.В. Кожевников. – Д.: Нац. гірн. ун-т, 2011. – 108 с.
3. *Кавчук С.В.* Сборник примеров и задач по теории информации. Руководство для практических занятий на базе Mathcad 6.0 Plus: Учебн. пособие. – Таганрог: Изд-во ТРТУ, 2002. – 64 с.
4. *Паттерны* проектирования / Э. Фримен, К. Сьерра, Б. Бейтс. – СПб.: Питер, 2011. – 656 с.
5. *Стиллмен Э., Грин Дж.* Изучаем C#. – 2-е изд. – СПб.: Питер, 2011. – 696 с.
6. *Фленов М.Е.* Transact-SQL. – СПб.: БХВ-Петербург, 2006. – 576 с.

Рекомендована Радою
Навчально-наукового комплексу
“Інститут прикладного системного
аналізу” НТУУ “КПІ”

Надійшла до редакції
14 травня 2014 року