Софийски университет „Св. Климент Охридски“

Факултет по математика и информатика

ДОКУМЕНТАЦИЯ

Курсов проект по „Структури от данни“

Тема: Алгоритъм на Хъфман

Име: Георги Цеков

Курс: 3

Група: 2

Ф.н: 72039

Специалност: „Информационни системи“

1. Алгоритъм на Хъфман:

Алгоритъмът на Хъфман представлява алгоритъм за компресия без загуба на данни. Той се базира на това, че най-често срещаните символи в поредицата се записват с възможно най-малък брой битове. По този начин се построява нова азбука, която следва тази идея и след това превежда информацията в новата азбука. Може да се направи и обратното, т.е. кодираната вече информация да се декомпресира и така да се получи първоначалната поредица от символи.

Компресирането на началната поредица се извършва чрез две основни стъпки: построяването на дървото на Хъфман и след това направата на кодова таблица, която „казва“ двоичния код на всеки един от символите, който се среща в първоначалната поредица.

2. Построяване на дървото на Хъфман:

1. Създаваме честотна таблица за низа – за всеки символ се записва броят на срещанията му;
2. Нека различните символи в низа са n на брой. Създаваме n дървета от по един възел, който съдържа наредена двойка: символ и число, което е чесотата на срещането на символа в низа. Създаваме приоритетна опашка, която подрежда възлите във възходящ ред спрямо честотата на срещанията им;
3. Взимаме първите два елемента от опашката и създаваме ново дърво, чиито ляв и десен наследник (дървото на Хъфман е двоично) са първите два взети върха от опашката, и стойността на корена на това получено дърво е сумата от съответните честоти от двата взети върха;
4. Повтаряме 3. Докато не получим само едно дърво – дървото на Хъфман за подадения низ.

Листата на това дърво са n на брой, точно толкова, колкото са и броя на различните символи, които се срещат в първоначалния низ. Останалите върхове на дървото на Хъфман съдържат в себе си само сумата от честотите на преките си двама наследници.

3. **Построяване на кодова таблица:**

На всяко ребро от дървото на Хъфман съпоставяме 0 за ляво ребро и 1 за дясно ребро. Така на всеки път от корена на дървото до кое да е листо отговаря двоичен код. И така, за всеки символ отговаря някаква двоична последователност, която съответства на пътя от корена до листото, в което се намира този символ.

След като получим кодовата таблица, извършваме кодиране на низа – всеки символ заместваме с неговия двоичен код. И по този начин получаваме компресирана последователност от 0 и 1.

4. **Декомпресиране на компресираната информация:**

Процесът на декомпресиране се случва благодарение на дървото на Хъфман и компресирания низ като ги обхождаме едновременно, като ако срещнем в компресирания низ символ 0, се насочваме към лявото поддърво на дървото на Хъфман, а в противен случай – към дясното поддърво. Ако сме стигнали до листо, то извеждаме съответния символ и започваме отново от корена на дървото до някое листо. И така, докато не минем през всични листа. По този начин получаваме първоначалния низ.

5. **Класове, които са използвани за направата на цялостния алгоритъм:**

- Клас **HTree** – клас, който е свързан пряко с алгоритъма на Хъфман и дървото на Хъфман – създаване на честотна таблица, създаване на дървото на Хъфман, създаване на кодовата таблица, метода за компресиране на първоначално зададен низ, декомпресиране, пресмятане на степента на компресия и debug compress;
- Клас **Node** – клас, който е свързан с представянето на връх от дървото на Хъфман;
- Изброим тип **Modes** – изброим тип, който съдържа възможните режими, в които програмата може да работи – режим на компресия и режим на декомпресия;
- Клас **HCoding** – клас, който използва класа HTree, т.е в методите ѝ се използват методите на HTree (за компресия, декомпресия и т.н) и е

свързан с писането и четенето на информацията, която ни интересува, във файлове;

- Клас **Commands** – клас, който използваме за командния ред;
- **tests.cpp** - файл, който съдържа тестове, свързани с алгоритъма на Хъфман;

- **HTree.h**:

```cpp
#include "Node.h"

//class, which is directly related with the Huffman algorithm - the class contains methods for creating frquency table,
//creating the Huffman tree, creating code table, compression and decompression, etc.
class HTree{
    private:
        //pointer to the root of the Huffman tree
        Node* root;

        //struct compare, which helps us to create the priorityQueue (min heap)
        struct compare{
            //compares if first->frequency > second->frequency
            bool operator()(Node* first, Node* second) const;
        };

        //helper method, which is contained in the destuctor of the class HTree and it destroys the tree
        //parameters: current - pointer to the root of the Huffman tree
        void clear(Node* current);

        //helper method, which creates the code table
        //parameters: current   - pointer to the root of the Huffman tree
        //            trace     - string, in which we will write the path from root to any leaf (when "current" become a leaf node,
        //                        we insert in the codeTable the path (trace) from root of the tree to the leaf node)
        //            codeTable - a map with key   - any symbol, which is contained in the text, which we want to compress and
        //                                value - the path from the root of the Huffman tree to the key, which is a leaf node
        void createCodeTableHelper(Node* current, const std::string& trace, std::map<char, std::string>& codeTable) const;

        //helper method, which helps us to print the tree in file or in the standard output
        //parameters: current - pointer to the root of the tree
        //            out     - output stream
        void printTreeHelper (Node* current, std::ostream& out) const;

        //helper method, which decompress a symbol
        //parameters: current      - pointer to the root of the tree
        //            compressed   - string, which contains the compressed content and from which we get the "information" for decompression
        //            decompressed - string, in which we will push already decompressed symbol and it will help us
        //                           to return the result of decompression
        //            index        - index, which is related to "compressed" and it is used in "decompress" method
        void decompressSymbol(Node* current, const std::string& compressed, std::string& decompressed, int& index) const;

    public:
        //Big four; we delete the copy constructor and operator=, because we don't want to copy trees
        HTree();
        HTree(const HTree& other) = delete;
        HTree& operator=(const HTree& other) = delete;
        ~HTree();

        //method, which returns pointer to the root of the tree
        Node* getRoot() const;

        //method, which creates a frequency table from given "inputText" and returns a map, which
        //key - symbol, which is contained in the "inputText"
        //and value - the frequency of the symbol in the "inputText"
        //parameters: inputText - string, which contaains the content we want to compress/decompress, etc.
        std::map<char, int> createFriquencyTable(const std::string& inputText) const;

        using priority_queue = std::priority_queue<Node*, std::vector<Node*>, compare>;

        //method, which creates a priority queue from given frequency table (the priority is the frequency of the symbol in the input text)
        //parameters: frequencyTable - a map, which key - symbol, which is any the symbol in the input text and value -
        //the frequency of the symbol in the input text
        priority_queue createPriorityqueueFromFrequencyTable(const std::map<char, int>& frequencyTable) const;

        //method which creates the Huffman tree from given priorityQueue
        //parameters: priorityQueue - priority queue (the priority is the frequency of the symbol in the input text)
        void createHTree(priority_queue priorityQueue);
```

```
82
83      //method, which creates a code table (for every symbol there will be a binary code) and returns a map, which key - symbol, which is any
84      //the symbol in the input text and value - string, which contains a binary code, which is taken from the Huffman tree
85      std::map<char, std::string> createCodeTable() const;
86
87      //method, which compress the "input" and returns the compressed string
88      //parameters: input - the content, which we have input and want to compress, decompress, etc.
89      std::string compress(const std::string& input);
90
91      //method, which decompress the "compressed" with the help of the "current", which is pointer to the root of the tree,
92      //and returns the decompressed string
93      std::string decompress(Node* current, const std::string& compressed) const;
94
95      //method, which makes a debug compress of the compressed information and returns a vector, which contains the values, which are
96      //result of the debug compress
97      //parameters: input - string, which contains compressed information
98      std::vector<int> debugCompress(const std::string& input) const;
99
100     //method, which computes the degree of compression and returns a value, which is result of dividing the size (bits) of the
101     //"compressed" by the "inputText"
102     //parameters: inputText  - the content, which we have input and want to compress, decompress, etc.
103     //            compressed - the string, which contains the compressed content
104     double degreeOfCompression(const std::string& inputText, const std::string& compressed) const;
105
106     //method, which prints a tree in a file or in the standard output
107     //and the Huffman tree is saved in preorder (root, left, righ) (Scheme format)
108     //parameters: out - output stream
109     void printHuffmanTree (std::ostream& out) const;
110
111 };
```

- **Node.h**:

```
3   //struct, which is a part of the representation of a node in a Huffman tree and is something like the std::optional
4   //(node may have a symbol (the symbol of the content, which we want to compress/decompress, etc) or not
5   //(if a node is a leaf, then it as a defined symbol else not))
6   struct Maybe{
7       char data;
8       bool isDefined;
9
10      Maybe();
11      Maybe(const char& _data);
12  };
13
14  //struct, which represents a node in a Huffman tree:
15  //frequency - the frequency of any symbol in a content, which we want to compress/decompress
16  //symbol    - a symbol, which is part of the content
17  struct Node{
18      int friquency;
19      Maybe symbol;
20
21      Node* left;
22      Node* right;
23
24      Node(const int& _frequency, const Maybe& _symbol, Node* _left, Node* _right);
25  };
```

- **Modes.h**:

```
3   //enum, which represents the current mode we work with and it is used in Commands.cpp
4   enum Modes{
5       COMPRESS,
6       DECOMPRESS
7   };
```

- **HCoding.h**:

```cpp
#include "HTree.cpp"

//class, which works with files and works with class HTee (writes the compressed content in a file and writes the Huffman tree in a file
//                                                         and also reads the compressed content and the Huffman tree and creates
//                                                         the decompressed ouput file)
class HCoding{
    private:
        HTree huffmanTree;

        //method, which reads the content, which is already compressed, from a binary file
        //parameters: fileName - the name of the file, from which we will read the compressed content,
        //            content  - the content, in which we will write the information that we have read from "fileName"
        void readCompressedContentFromBinaryFile(const std::string& fileName, std::string& content) const;

        //method, which reads the Huffman tree from a text file and returns pointer to the root of the Huffman tree
        //parameters: fileName - the name of the file, from which we will read the Huffman tree
        Node* readHuffmanTreeFromFile(const std::string& fileName) const;

        //method, which helps to the method "readHuffmanTreeFromFile" to read the Huffman tree and returns pointer to the root of the Huffman tree
        //parameters: in - input stream, which helps us to read from a file
        Node* readHuffmanTree (std::istream& in) const;

    public:

        //method, which compress the content from "fileName" and returns the compressed string
        //parameters: fileName - the name of the file, from which we will read the content and then we will compress it
        std::string compress(const std::string& fileName);

        //method, which outputs the compressed content into the fileName
        //parameters: content  - the compressed content we will output into fileName
        //            fileName - the fileName, in which the content will be saved
        void outputCompressedContentIntoBinaryFile(const std::string& content, const std::string& fileName);

        //method, which save the result Huffman tree into fileName
        //parameters: fileName - the name of the file, which the Huffman tree will be saved
        void outputHuffmanTreeIntoFile(const std::string& fileName);

        //method, which outputs the result of debug compress
        //parameters: content - the compressed content, from which we will get the information for debug compress
        void debugCompress(const std::string& content) const;

        //method, which computes the degree of compression
        //parameters: inputContent      - the content, which we have input first
        //            compressedContent - the compress content, which is result of compression of "inputContent"
        void computeDegreeOfCompression(const std::string& inputContent, const std::string& compressedContent);

        //method, which decompress the compressed content from "fileName" and for this process we need also the file,
        //which contains the Huffman tree
        //parameters: fileName     - the name of the file, from which we will read the compressed content
        //            treeFileName - the name of the file, from which we will read the Huffman tree
        std::string decompress(const std::string& fileName, const std::string& treeFileName) const;

        //method, which outputs the decompressed content in "fileName"
        //parameters: content  - the decompressed content, which we will output in the "fileName"
        //            fileName - the name of the file, in which the decompressed content will be saved
        void outputDecompressedContentIntoFile(const std::string& content, const std::string& fileName);
};
```

- **Commands.h**:

```
7   //struct, which helps us to run the program
8   struct Command{
9       private:
10          //method that output the valid commands with their arguments
11          void intro();
12
13          //method that outputs the content, which we are going to compress/decompress and do other commands, in a text file
14          //parameters: content     - the string which we have input and that is the content which we want to manipulate
15          //            contentFile - the name of the file we want to save the content
16          void outputContent(const std::string& content, const std::string& contnentFile);
17
18      public:
19          //method, that runs the program with the given command parameters
20          void run();
21  };
```