Text Compression Utility

Group Number: 4
Department of Computer Science and Engineering, AUC

Abstract: File compression is essential for space optimization as well as transferring and sharing data across a network. Without it, many basic computer operations would either seize to exist or take significantly large time and space. In this project, we constructed our own text compression utility which compresses ASCII-encoded text files to binary files and decompresses binary files back to their original text format. This was done by utilizing Huffman coding. We also calculated the time complexity, space complexity, compression ratio, and efficiency of our program in contrast to the optimal efficiency expected.

Keywords: Huffman coding, Greedy Algorithm, compression, decompression

---

1. Introduction

Although often associated with the digitalized era, data compression had come to light as early as 1838 through Morse Code, where a shorter code was assigned to the most common English letters, ‘t’ and ‘e’. When it comes to today’s computerized world, Data compression has gained greater and greater significance. File compression algorithms have revolutionized optimizing space as well as the speed and ease of sharing file data by decreasing the disk space it takes up. Lossless algorithms enable the reduction and resizing of files and messages back to their original size and format remaining perfectly unchanged. We will be implementing a compression/decompression through a popular encoding algorithm; Huffman coding. This encoding algorithm is used in compression formats like GZIP as well as Multimedia codecs like JPEG and MP3

2. Problem Definition

In this project, we will be implementing a program designed to enable the user to either losslessly compress or decompress a selected ASCII-encoded text file. The program will also generate a console-based output presenting the compression ratio and the efficiency of the encoding process.

3. Methodology

In order to perform lossless compression of the selected ASCII-encode text file to an acceptable decreased proportion we have built utilizing the infamous greedy algorithm; variable length Huffman Coding. This algorithm generates a Huffman tree in both encoding and decoding and utilizes it to convert from ASCII text to binary and vice versa through the generated codes. The significance of this algorithm is due to the fact that it minimizes the average length < L > of a code word getting as close as possible to Shannon's bound (H).

4. Specification of Algorithms to be used

Firstly, we calculate the repetition frequencies of each symbol/character in the text file which reflects the probability of its occurrence. Through a priority queue resembling a minimum heap based on the likelihood of occurrence for each symbol, alongside a greedy approach, we construct a Huffman Tree. To obtain an optimal merge pattern for the tree construction we place more frequent characters and symbols closer to the root in order to assign to them relatively shorter codes (hence the variable length handling). The tree consequently generates the desired binary prefix-free coding scheme. This scheme is then used to translate the file to a binary code. To improve the compression further, we stored the encoded data in the binary file by converting every 8 bits to its ASCII equivalent to further optimize the compression.

For decompression, we utilized the reversed approach of compression by reconstructing the Huffman tree through the key at the start of the compressed file. After reconstructing the tree we trace each binary digit along the tree until a symbol is obtained. This works due to the code initially being prefix-free. The file is then rewritten fully.

5. Data Specifications

The program is designed to handle ASCII-encoded text files for compression and binary files for decompression.

6. Experimental Results

The compressing and decompressing options were both tested on 8 different files each of varying lengths in terms of lines and word count, as well as character and symbol variations. Every file was successfully compressed to a binary file which was approximately half the size of the original. All files were also successfully decompressed back to their original text format and we confirmed that they were identical to the original text pre-compression.

7. Analysis and Critique

Efficiency: The efficiency of our compression was tested on the examples in our lecture slides and were close to the expected optimal values.

Time Complexity: At the start of the program, calculating the character probability of characters was O(n). Since Huffman coding revolves around the construction and traversal of the Huffman tree its complexity is O(*n log n*). This is because the generation of the minimum heap through a priority queue possesses a complexity of O(*n log n*) and the traversal of the tree requires O(*log n*). Thus since both compression and decompression contain the same steps in reverse orders we have that the overall complexity of both is O(*n log n*) where n is the number of unique symbols.

Space complexity: The Huffman tree has space complexity of O(n) where n is the number of different symbols present in the file. All other aspects of the code utilize one-dimensional data structures and thus the total space complexity does not exceed O(n).

8. Conclusion

Our project relied heavily upon Huffman's coding algorithm which enabled relatively efficient lossless file compression as the average length < L > of a code word generated is very close to the optimal Shannon's bound (H). Our adapted implementation of the algorithm managed to successfully compress inputted ASCII-encoded files to almost half their original size as a binary file. Our reversed approach for decompression also successfully returned the original text file without loss of data with all being done at a time complexity of O(*n log n*).

9. Acknowledgments

We would like to thank our professor Dr. Amr Goneid and our teacher assistant Eng. Mohamed Hany for the crucial role they played in making this project and semester a success. Their thorough explanation of the core concepts of Huffman coding and greedy algorithms as well as Dr. Goneid's notes and slides eased our understanding and thus implementation of the project's requirements.

10. References

Ethw. (2019, January 22). *History of lossless data compression algorithms*. ETHW. Retrieved December 10, 2022, from https://ethw.org/History_of_Lossless_Data_Compression_Algorithms#:~:text=Morse%20code%2C%20invented%20in%201838%2C%20is%20the%20earliest,Claude%20Shannon%20and%20Robert%20Fano%20invented%20Shannon-Fano%20coding.

Goneid, A. (n.d.). *Part 8. Greedy Algorithm*. Retrieved from https://www1.aucegypt.edu/faculty/cse/goneid/csce2202/Part%208%20Greedy%20Algorithms.pdf

Hosseini, M. (n.d.). *A survey of data compression algorithms and their applications*. www.researchgate.net. Retrieved December 10, 2022, from https://www.researchgate.net/publication/270408593_A_Survey_of_Data_Compression_Algorithms_and_their_Applications

Huffman coding: Greedy Algo-3. GeeksforGeeks. (2022, October 26). Retrieved December 10, 2022, from https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/

11.Appendix:

Source Code:

```
#include <iostream>
#include <string>
#include <cstring>
#include <queue>
#include <fstream>
#include <sstream>
#include <iomanip>
#include <cmath>
#include <unordered_map>
#include <vector>

using namespace std;

struct node
{
   char id;
   int frequency;
```

```
    node *left, *right;
};

struct comp
{
    bool operator()(node *LEFT, node *RIGHT)
    {
        return LEFT->frequency > RIGHT->frequency;
    }
};

node *create_new_node(char, int, node *, node *); // creates new node

node *toTree(string, node *, long long int &); // reconstruct the tree from the decompressed file

ifstream::pos_type size(string); // to get the size of each file

char ASCII(string); // brings ASCII equivelent

int bintodec(string); // binary to decimal

string dectobin(long long int); // decimal to binary

string ascii_to_binary(string); // from ascii to decimal to binary

string undo(string); // brings back the encoded binary message

void compression(string, string); // to calculate the ratio using size/size not the probability

void HUFFMAN(long long int, string, string); // either compression or decompression

void compressor(node *, string, unordered_map<char, string> &); // constructs the tree and the
map of the encoded values

void decompressor(node *, long long int &, string, long long int, ofstream &); // returns the
original file

void tree(node *, ostream &); // writes in the decompressed file the endcoded table in tree format

void print(node *); // prints the tree
```

```
void CalculateRatio(string, string, unordered_map<char, string>); // to calculculate ratio using frequency

long double Avg_Len(unordered_map<char, string>);

long double entropy(unordered_map<char, long double>);

unordered_map<char, long double> propability(int, unordered_map<char, long long int>);

int main()
{
   int ans;
   string file1 /*= "test.txt"*/, file2 = "compressed", file3 = "decompressed";
   cout << "\t\t  ================================================" << endl
        << "\t\t||                                    ||" << endl
        << "\t\t||          Please chose from below          ||" << endl
        << "\t\t||                                    ||" << endl
        << "\t\t||          1. Compress                ||" << endl
        << "\t\t||          2. Decompress               ||" << endl
        << "\t\t||                                    ||" << endl
        << "\t\t  ================================================" << endl
        << "\t\t||" << endl
        << "\t\t||" << endl
        << "\t\t||" << endl
        << "\t\t||" << endl
        << "\t\t||_______Type here: ";

   cin >> ans;

   system("clear");
   // cin >> file1;

   if (ans == 1 || ans == 2)
   {
      if (ans == 1)
      {
         cout << "\t\t  ================================================" << endl
              << "\t\t||                                        ||" << endl
              << "\t\t||     Please enter the file's full path       ||" << endl
```

```
            << "\t\t||                                          ||" << endl
            << "\t\t  ==============================================" << endl
            << "\t\t||" << endl
            << "\t\t||" << endl
            << "\t\t||" << endl
            << "\t\t||" << endl
            << "\t\t||_______Type here: ";
        cin >> file1;

        system("clear");

        HUFFMAN(ans, file1, file2);
      }
      else
        HUFFMAN(ans, file3, file2);
  }

  else
  {
    cerr << "INVALID SELECTION PROGRAM ENDING.." << endl;
  }

  cout << endl;

  return 0;
}

node *create_new_node(char c, int f, node *left, node *right)
{
  node *n = new node;
  n->id = c;
  n->left = left;
  n->right = right;
  n->frequency = f;
  return n;
}

void compressor(node *root, string s, unordered_map<char, string> &huff_code)
{
  if (root == nullptr)
```

```
        return;

    if (!root->left && !root->right)
    {
        huff_code[root->id] = s;
    }

    compressor(root->left, s + "0", huff_code);
    compressor(root->right, s + "1", huff_code);
}

void decompressor(node *root, long long int &j, string s, long long int k, ofstream &out)
{

    if (root == nullptr)
    {
        return;
    }

    if (!root->left && !root->right)
    {

        // cout << root->id;
        out.put(root->id);

        return;
    }

    if (s[j++] == '0')

        decompressor(root->left, j, s, k, out);
    else
        decompressor(root->right, j, s, k, out);
}

node *toTree(string storeInfo, node *root, long long int &i)
{

    if (storeInfo[i] == '1')
    {
```

```
        root = create_new_node(storeInfo[++i], 0, nullptr, nullptr);
        return root;
    }

    root = create_new_node('\0', 0, nullptr, nullptr);

    root->left = toTree(storeInfo, root->left, ++i);
    root->right = toTree(storeInfo, root->right, ++i);
    return root;
}

void print(node *root)
{
    if (root->left == nullptr && root->right == nullptr)
    {
        // cout << root->id << endl;
        return;
    }
    else if (root == nullptr)
    {
        return;
    }
    print(root->left);
    print(root->right);
}

string dectobin(long long int dec)
{

    if (dec < 1)
        return "0";

    string binStr = "";

    while (dec > 0)
    {
        binStr = binStr.insert(0, string(1, (char)((dec % 2) + 48)));

        dec /= 2;
    }
```

```
    return binStr;
}

string ascii_to_binary(string str)
{
    string bin = "";
    long long int strLength = str.length();

    for (int i = 0; i < strLength; ++i)
    {
        string cBin;
        cBin = dectobin(int(str[i]));
        long long int cBinLength = cBin.length();

        if (cBinLength < 7)
        {
            for (size_t i = 0; i < (7 - cBinLength); i++)
                cBin = cBin.insert(0, "0");
        }
        bin += cBin;
    }

    return bin;
}

string undo(string storeCharacters)
{
    string x, m("");
    for (long long int i = 0; i < storeCharacters.length(); ++i)
    {
        x = storeCharacters[i];
        m += ascii_to_binary(x);
    }
    return m;
}

void tree(node *root, ostream &out)
{
```

```
    if (!root)
    {
        return;
    }
    if (!root->left && !root->right)
    {
        out.put('1');
        out.put(root->id);
    }
    else
    {
        out.put('0');
    }
    tree(root->left, out);
    tree(root->right, out);
}

int bintodec(string n)
{

    int value = 0;
    int ix = 0;
    for (int i = n.length() - 1; i >= 0; i--)
    {

        if (n[i] == '1')
        {
            value += pow(2, ix);
        }
        ix++;
    }
    return value;
}

char ASCII(string s)
{
    int decimal = 0;
    decimal = bintodec(s);
    return char(decimal);
}
```

```
void compression(string file1, string file2)
{
   float original_size;
   float compressed_size;
   float compression_ratio;

   original_size = size(file1);
   compressed_size = size(file2 + ".bin");
   compression_ratio = (compressed_size) / (original_size);

   cout << "The compression ratio is: " << compression_ratio << endl;
}

void HUFFMAN(long long int ans, string File1, string File2)
{
   if (ans == 1)
   {
      unordered_map<char, long long int> f;
      unordered_map<char, string> h_codes;
      priority_queue<node *, vector<node *>, comp> PQ;
      node *root;
      char tc;
      long int bits = 0, totalF(0);
      long int s = 0;
      string m, n; // m is the string that represents the binary number

      ifstream in;
      ofstream out;

      in.open(File1);
      out.open(File2 + ".bin", ios::binary);

      if (in.fail())
      {
         cout << "Error (original file)";
         exit(0);
      }
      else if (out.fail())
      {
```

```
    cout << "Error (writing file)";
    exit(0);
}

while (!in.eof())
{
    in.get(tc);
    if (in.eof())
        break;
    f[tc]++;
    totalF++; // overall
}
in.close();

for (auto pair : f)
{
    PQ.push(create_new_node(pair.first, pair.second, nullptr, nullptr));
}

while (PQ.size() != 1)
{
    node *left = PQ.top();
    PQ.pop();
    node *right = PQ.top();
    PQ.pop();
    s = left->frequency + right->frequency;
    PQ.push(create_new_node('\0', s, left, right));
}
root = PQ.top();
PQ.pop();

compressor(root, "", h_codes);
tree(root, out);

in.open(File1);
while (!in.eof())
{
    in.get(tc);
    if (in.eof())
        break;
```

```
        bits += h_codes[tc].length();
        m += h_codes[tc];

    }
    while (m.length() % 7 != 0)
        m += '0';

    out << '\0' << to_string(bits) << '\0';

    for (long long int i = 0; i < m.length(); i += 7)
    {
        if (m.length() - i < 7)
        {
            tc = ASCII(m.substr(i, m.length() - i));
        }
        else
        {
            tc = ASCII(m.substr(i, 7));
        }
        // cout << tc;
        out.put(tc);
    }
    // cout << "encoded message:\n"
    //      << m << endl;
    in.close();
    out.close();


    // compression(File1, File2);


    CalculateRatio(File1, File2, h_codes);

    unordered_map<char, long double> probs = propability(totalF, f);
    long double Entropy = entropy(probs);

    cout << "Efficiency: " << Entropy / Avg_Len(h_codes);
}
else if (ans == 2)
{
```

```
    ifstream in;
    ofstream out;

    in.open(File2 + ".bin", ios::binary);
    out.open(File1 + ".txt");

    string storeInfo;
    string storeCharacters;
    string storeSize;
    getline(in, storeInfo, '\0');
    getline(in, storeSize, '\0');
    // int k = storeInfo[storeInfo.length()-1];

    node *root;
    long long int i(0);
    root = toTree(storeInfo, root, i);
    string m = ("");
    // print(root);
    // cout << "afterPrint\n";
    while (!in.eof())
    {
       getline(in, storeCharacters);
       m += storeCharacters;
    }
    // cout << "characters: \n"
    //      << m << endl;
    m = undo(m);
    // cout <<"binary encoded: " <<m <<endl;
    long long int j = 0;
    if (m.length() != stoi(storeSize))
       m = m.substr(0, stoi(storeSize));
    int k = storeInfo.length();

    while (j < m.length())
       decompressor(root, j, m, k, out);
}

else
{
    exit(1);
```

```
    }
}

void CalculateRatio(string file1, string file2, unordered_map<char, string> h_codes)
{
    cout << "The compression ratio is " << Avg_Len(h_codes) / 8 << endl;
}

ifstream::pos_type size(string filename)
{
    ifstream in(filename, ifstream::ate | ifstream::binary);
    ifstream::pos_type fsize = in.tellg();

    in.close();

    return fsize;
}

long double Avg_Len(unordered_map<char, string> h_codes)
{
    int count(0), temp(0);

    for (auto i : h_codes)
    {
        temp += i.second.length();
        ++count;
    }

    return (long double)(temp) / (long double)(count);
}

long double entropy(unordered_map<char, long double> probs)
{
    long double Entropy(0.0);

    for (auto i : probs)
        Entropy += (long double)(i.second * log2(1 / i.second));

    return Entropy;
}
```

```
unordered_map<char, long double> propability(int m, unordered_map<char, long long int> f)
{
   unordered_map<char, long double> probs(f.size());

   // cout << "message: " << m << endl;

   for (auto i : f)
      probs[i.first] = (long double)(i.second) / (long double)(m);

   return probs;
}
```