Compressão de Huffman

Antonio Marcos de Oliveira, Felipe Costa Ramos

Instituto Metrópole Digital, Universidade Federal do Rio Grande do Norte

6 de dezembro de 2018

Este relatório está baseado em três pilares:

- Como a codificação de Huffman foi implementada – detalhes e decisões específicas da implementação
- Detalhes sobre a estrutura do código (classes, atributos, métodos, responsabilidades de cada método, etc)
- bibliotecas utilizadas e com que finalidade

Detalhes e decisões específicas

Aqui estão alguns detalhes do programa:

- Foi usado '\o' por recomendação do PDF
- Foi usado hash table para armazenar o caminho dos caracteres e evitar ficar fazendo uma busca na árvore para cada caractere
- A modularização dos arquivos com os códigos relacionados
- Foi usado os conceitos de Mask para ler os bits dos caracteres
- A compressão foi feita conforme o PDF
- A descompressão foi recriada a árvore até o delimitador, depois os bits foram lidos referentes aos caminhos dos caracteres para reconstruir a árvore.

Detalhes sobre estruturas do código

O projeto está modularizado da seguinte forma:

- include: contendo os arquivos de cabeçalhos criados

  – bits.h: (Como bits.h e compress.h compartilham o mesmo nome em dois métodos, cada um foi especificado com um namespace, cada um com seu respectivo nome)

  > **std::string genBinary(std::string &is)**. Pega um binário semelhante a uma string e a transforma em uma real.

> **std::string getBits(unsigned char n)**. Pega uma string com os bits em um determinado char.

> **void printBits(std::string name, std::string ifs)** Printa os bits que compõem cada char da string ifs.

> **void printLikeBits(std::string name, std::string ifs)**. Imprime a string como bits.

– compress.h:

> **std::string compress(std::string &is, DigitalTree &tree)**. Comprime uma string contendo todos os bits para o nível binário real.

> **std::pair<std::string, std::string> uncompress(std::string &is)**. Descomprime de um nível binário real para uma string contendo todos os bits.

> **std::string getDelimiter(void)**. Pega o delimitador entre cabeçalho e dados.

– counter.h

> **std::vector<Node *> generateStats(std::string content)**. Gera a contagem da ocorrência das letras no arquivo original e retorna um vector de ponteiros pra nodes.

– digital-tree.h

> **Node *m_root;** . Raiz da árvore.

> **std::vector< std::string > path_to_leaf**. Hash table para acelerar as coisas ao procurar por um caminho char.

> **bool genPathToChar(unsigned char ch, Node * curr, std::vector<bool> &i_path)**. Função recursiva para obter o caminho para char.

> **void i_preOrder(Node * curr, std::string &acc)**. Função recursiva interna para imprimir a travessia de pré-ordem

> **DigitalTree(std::string &preord)**. Constrói a árvore por em caminho transversal de pré-ordem.

> **Node * genTree(std::queue<std::pair<bool, char» & node_list)**

> **DigitalTree();** Destrutor da classe.

> **std::vector<bool> pathTo(unsigned char ch)**. Pega o caminho binário de um char passado por parâmetro.

> **std::string preOrder()**. Pega a representação transversal da árvore em pré-ordem.

> **std::string decode(std::string & _str)**. Decodifica uma string passada por parâmetro.

> **Node(unsigned char k, Node * m_left, Node * m_right): key(k), left(m_left), right(m_right) {/* */}**

– io.h

> **std::string read(std::ifstream &ifs)**. Lê o conteúdo de um objeto ifstream.

> **bool write(std::ofstream &ofs, std::string content)**. Escreve o conteúdo em um objeto ofstream.

– node.h

> **unsigned char key**. Armazena o char.

> **unsigned long int freq**. Armazena a frequencia.

> **Node * father = nullptr**. Ponteiro para o nó pai.

> **Node * left = nullptr**. Ponteiro para o nó filho esquerdo.

> **Node * right = nullptr**. Ponteiro para o nó filho direito.

> **Node()/* empty */**.

> **Node(unsigned char k, unsigned long int f) : key(k), freq(f) /* */**.

> **unsigned char getKey() std::cout « "ih\n"; return this->key;** .

- src: contendo as implementações previamente assinadas em seus respectivos .h (arquivos de cabeçalho)

  – bits.cpp

  – compress.cpp

  – counter.cpp

  – digital-tree.cpp

  – io.cpp

  – main.cpp

  Após a inserção das bibliotecas e arquivos de cabeçalho, faz-se uma verificação dos argumentos passados, e caso não esteja tudo certo, retorna uma mensagem de erro referente o uso dos parâmetros corretos. Se tudo ocorrer bem, os arquivos de entrada e saída serão gerados.

  Com as informações (caracteres) dos arquivos entrada serão gerados cada estatística de cada char para ver a quantidade de ocorrência de cada um. Com essa informação é gerada a árvore digital com os pesos dos chars.

  Com a árvore, agora o debug é feito da seguinte forma: o stats é percorrido e é impresso todos os caracteres e quantas vezes eles apareceram. Em seguida, a função pathTo() retorna um vector de bool contendo o caminho de 0's e 1's até o nó que contém a chave i->key. Após isso, é feito uma conversão desse vector de bool para string para ser impresso.

  No final, é impressa a representação da árvore em pré-ordem.

  Depois disso, é feita a compressão e descompressão dos dados.

- tests: contendo os casos de testes

  – default.in

  > Contém um exemplo comum de testes para a codificação de Huffman.

  – heavy.in

  > Contém um exemplo maior de texto para codificação.

– jp.in

> Outro exemplo de texto para codificação.

– test.in

> Exemplo de tamanho intermediário de texto para codificação.

Seguido por dois arquivos padrões do Github, que é o .gitignore e o README.md, além do Makefile, para tornar mais fácil a compilação.

Bibliotecas utilizadas

- iostream: Como se trata de um programa onde há entrada/saída de stream, é essencial o uso da iostream.
- string: Para manipular os caracteres no programa, fez-se necessário o uso da biblioteca string.
- vector: Como queremos guardar uma cadeia de bits, a biblioteca utilizada para guardar esse dados foi o vector, por sua facilidade e versatilidade.
- map: Sendo necessário associar cada char a um inteiro, a biblioteca map veio bem a calhar nessa missão.
- queue: Visto que cada char tem um peso diferente na hora de montar a árvore, usar o std::priority_queue() foi necessário para conseguir pegar os elementos certos, visto que ele é uma *container fifo*, ou seja, pegamos os elementos com pesos certos, na ordem certa.
- fstream: Como se recebe os arquivos, onde precisamos ler os arquivos e depois retornar os dados comprimidos, se fez necessário o uso da biblioteca fstream.
- algorithm: Se fez necessário para utilizar a função std::reverse() na hora de pegar os bits dos caracteres.
- utility: Como na descompressão do caractere é necessário ver se a sequencia de bits é realmente correspondente aquele caractere, foi usado a função std::pair() para isso.