Huffmnan Adaptativo

Pedro Garcia Freitas
11/0068408

14 de maio de 2011

Resumo

Neste trabalho, apresentamos e analisamos a performance de um algoritmo que constrói códigos de Huffman dinamicamente em um passo.

1 Introdução

Códigos de tamanho variável, como é o caso dos códigos de Huffman, são utilizados em diversas aplicações que necessitam de compressão. O custo da comunicação em sistemas de comunicação remotos, a crescente demanda por armazenamento de dados e a melhoria na resolução dos meios de mídia são exemplos de importantes aplicações que necessitam de compressão de dados. Códigos de tamanho-variável permitem uma diminuição do espaço consumido na representação de um alfabeto, quando comparados com recursos que utilizam códigos de tamanho-fixo, tais como o código ASCII.

O código de Huffman cria uma árvore binária que minimiza o peso dos caminhos externos de tamanho $\sum_j w_j l_j$ dentre todas outras árvores binárias internas, onde w_j é o peso da $j - sima$ folha e l_j é a profundidade na árvore[2]. Supondo que existem k letras distintas $a_1, a_2, \ldots, a_k$ em uma mensagem a ser codificada, e consideramos uma árvore de Huffman com k folhas em que os pesos w_j, para $1 \leq j \leq k$, é o número de ocorrências de a_j na mensagem.

Uma forma de codificar esta mensagem é atribuir um código estático para cada código de k letras distintas, substituindo cada letra na mensagem por seu código correspondente. O algoritmo estático de Huffman usa um ótimo estático, onde cada ocorrência de a_j é codificada com l_j bits, que corresponde à especificação da largura do caminho percorrido na árvore de Huffman, contado da raiz até a $j - sima$ folha. Os caminhos a serem tomados são registrados por "0" ou "1", dependendo da seleção de direção feita em cada nó (direita ou esquerda).

Uma desvantagem do código de Huffman estático é que ele requer duas passagem sobre toda mensagem. Isto é, o método requer uma passagem para contagem das

frequências de cada símbolo na mensagem e outra passagem para codificar a mensagem, baseada numa árvore Huffman estática. Portanto, isto requer um tempo para cada uma das operações. Além disso, estas duas operações independentes apresenta-se menos eficiente em aplicações reais, uma vez que implica em um atraso na utilização da rede ou aumenta o acesso ao disco[3].

Alguns algoritmos foram propostos, sendo os de Gallager[3], Faller[4] e Knuth[5] foram os primeiros propostos, melhorados posteriormente por Vitter[2]. Nestes métodos dinâmicos, os codificador e o decodificador, iniciam com a mesma árvore inicial e realizam operações inversas, utilizando o mesmo algoritmo para modificar a árvore de Huffman, utilizando o próximo símbolo lido. Desta forma, a tabela que descreve o percorrimento na árvore não necessita ser enviada, diferente do que ocorre no Huffman estático.

Neste trabalho, é apresentado o algoritmo proposto por Vitter, descrito no livro "Introdução à Compressão de Dados", de Khalid Sayood[1], seguido de um comentário sobre os resultados de implementação.

2 O Algoritmo de Huffman Dinâmico

Seja o algoritmo de codificação de Huffman em dois passos:

Algorithm 1 Huffman Estático

Require: mensagem M
Ensure: Armazenar as k folhas em uma lista L
  while L contém ao menos dois nós **do**
    Remova da lista L dois nós x e y de menor peso
    Crie um novo nó p e faça p nó-pai de x e y
    O peso de p deve ser a soma do peso de y com o peso de x
    Insere p em L
  end while

O nó remanescente em L, ao fim da execução do algoritmo, é o nó-raiz da árvore binária, chamada de "árvore de Huffman". Em cada interação da repetição acima, há a escolha de dois nós de peso mínimo a ser removido de L. Diferentes escolhas, podem produzir diferentes árvores de Huffman, mas todas as possíveis árvores terão o mesmo comprimento de caminhos tomados.

No segundo passo do algoritmo de Huffman, a mensagem é codificada usando a árvore de Huffman construída. A primeira coisa que o emissor envia ao receptor é uma tabela formada pelos caminhos que saem da raiz e vão até as folhas, o qual deve corresponder à uma letra codificada. Cada ocorrência do símbolo a_j é codificado pela sequência de zeros e uns que especificam o caminho tomado para a esquerda ou para direita em cada nó que pertence ao caminho. Para decodificação, o decodificador cami-

nha pela árvore de Huffman da raiz até as folhas, gerando uma sequencia binária, que corresponde à um símbolo único.

Códigos como este, que correspondem à um caminho em uma árvore binária, são chamados de códigos de prefixo, uma vez que o código para uma letra não pode ser um prefixo do código de outra letra. O número de bits transmitidos é igual ao peso do caminho externo de largura $\sum_j w_j l_j$ mais o número de bits para codificar a tabela de códigos. O código de Huffman caracteriza-se por minimizar a largura mínima $\sum_j w_j l_j$c̃itehuffman.

As duas principais desvantagens dos dois passos descritos estão na necessidade de percorrer todo o arquivo codificado duas vezes e a necessidade de transmitir a tabela dos caminhos para o decodificador. Para evitar este *overhead*, utilizamos um algoritmo de passo único para codificar e decodificar, que lê, gera e altera a árvore de Huffman de acordo com a alteração na frequência dos símbolos lidos no arquivo original. Isto é, a árvore binária utilizada no processo da $(t+1) - sima$ letra é uma árvore de Huffman da fração M_t da mensagem. O compressor codifica a $(t+1) - sima$ letra a_{i_t} na mensagem com uma sequência de zeros e uns tal que especifica o caminho da raiz da árvore até a folha a_{i_t}. O decodificador recupera a mensagem original pela correspondência da árvore correspondente. Ambos, codificador e decodificador, modificam suas cópias da árvore antes do próximo símbolo ser lido e formar a outra mensagem M_{t+1}. O interessante desta abordagem é que o codificador e o decodificador não precisam trocar a árvore nem as modificações, uma vez que ambos utilizam o mesmo algoritmo de atualização da árvore, ou seja, eles construirão sempre a mesma cópia da árvore.

O algoritmo2 mostra a atualização da árvore de Huffman dinamicamente

Algorithm 2 Atualização

Require: mensagem M
  q recebe nó-folha correspondente à a_{i_j}
  if (q é o nó NYT) e ($k < n - 1$) **then**
    substituir q pelo nó-pai com duas folhas NYT filhas, numeradas na ordem filho da esquerda, filho da direita, pai.
    q recebe filho da direita, recém criado
  end if
  if q é irmão do nó NYT **then**
    Troca q com a folha de maior número com o mesmo peso
    Aumenta o peso de q em 1
    q recebe o nó-pai de q
  end if
  while q não é raiz **do**
    Troca q com o maior nó de mesmo peso, fazendo com q se torne o maior nó numerado deste peso
    Aumente o peso de q em 1
    q recebe o nó-pai de q
  end while

Neste caso, NYT é o símbolo utilizado para indicar símbolos que ainda não foram codificados na árvore de Huffman. Este algoritmo é implementado em [5]. Contudo, Vitter[2] propôs uma modificação nesta rotina de atualização da árvore, de forma gerar códigos ótimos.

A primeira ideia do algoritmo de Vitter é o uso de um esquema para numeração dos nós, chamado de *enumeração implícita*, onde cada nó é enumerado de forma crescente, de acordo com o nível a que pertence na árvore. Nós no mesmo nível são numerados em ordem crescente da esquerda para direita. Em termos de implementação, os nós no mesmo nível são ligados como uma lista encadeada.

2.1 Rotina de Atualização

O algoritmo de Vitter organiza como "blocos" os nós de mesmo nível e que possuam mesmo peso, sendo eles todos nós internos ou folhas. O líder de cada bloco é o maior nó numerado em um bloco. Estes blocos são ligados através de uma lista ordenada de acordo com os pesos. Uma bloco de folhas sempre precede um bloco interno de mesmo peso.

O procedimento de atualização requer que os nós preservem a enumeração correta. O maior nó numerado é dado à raiz da árvore, enquanto o menos número é atribuído ao nó NYT. Os valores atribuídos ao peso do NYT são os menores possíveis – na implementação, utilizamos peso 0. Além disso, os números que vão do nó NYT até a raiz da árvore estão ordenados da esquerda para direita, do menos para o maior nível.

Depois que um símbolo foi codificado ou decodificado, se verifica se o nó externo não possui o maior número do bloco, então, troca-se com aquele que possui este maior nó no bloco. Assim, o nó com maior número no bloco não é mais o mesmo, sendo trocado de posição com sei nó-pai. Então, o peso do nó externo é aumentado em um. Uma vez que trocamos o peso do nó, atualizamos sua posição na árvore de Huffman. Em seguida, o nível acima é examinado, repetindo a operação. Este processo se repete até atingir o nó-raiz. Este processo descrito está ilustrado no algoritmo 3

O algoritmo do procedimento `trocaEIncrementaPeso` é ilustrada no algoritmo4

Estes algoritmos estão esquematizados na figura1

2.2 Codificador

O programa de codificação inicializa a árvore de Huffman apenas com o um nó contendo o símbolo NYT. Em seguida, o codificador lê símbolo por símbolo do arquivo de entrada. Dependendo da ocorrência do símbolo, isto é, se ele ocorre a primeira vez ou não, ele é inserido na árvore ou atualizado, através da função de atualização na árvore. De forma simplificada, o codificador é ilustrado na figura2 e no algoritmo5.

Algorithm 3 Atualização

Require: Símbolo $a_{i_{t+1}}$
  folhaAIncrementar := 0
  q := nó-folha correspondente à $a_{i_{t+1}}$
  if (q é NYT) e (k < n - 1) **then**
    substituir q por um nó interno NYT por com nós-filhos NYT, tal que o filho direito corresponde ao símbolo $a_{i_{t+1}}$
    q := novo nó NYT
    folhaAIncrementar := filho direito de q
  else
    troca q na árvore com o líder do bloco
    if q é nó irmão do nó NYT **then**
      folhaAIncrementar := q
      q := nó-pai de q
    end if
  end if
  while q não é raiz **do**
    chama função `trocaEIncrementaPeso(q)`
  end while
  if folhaAIncrementar $\neq 0$ **then**
    `trocaEIncrementaPeso(folhaAIncrementar)`
  end if

Algorithm 4 trocaEIncrementaPeso

Require: Referência do nó p
  wt := peso de p
  b := bloco que contém p na lista encadeada
  if ((p é folha) e (b é o bloco de nós internos de peso wt)) ou ((p é um nó interno) e (b é bloco de folhas de peso $wt + 1$)) **then**
    desvia p na árvore através dos nós de b
    incrementa o peso de p em $wt + 1$
    if p é folha **then**
      p := novo pai de p
    else
      p := antigo pai de p
    end if
  end if

2.3 Decodificador

Assim como no codificador, o decodificador inicializa-se com uma árvore de apenas um único nó NYT. Como os símbolos recebidos são lidos como um fluxo de bits, e sabendo-

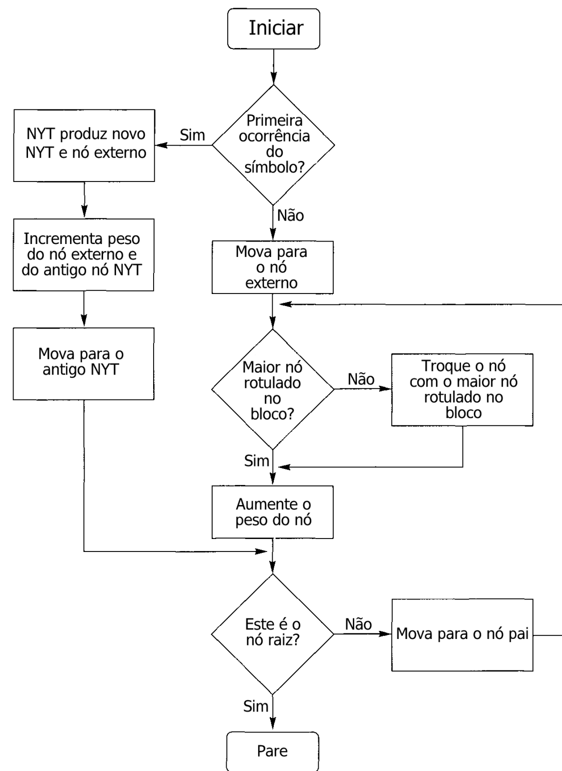

Figura 1: Diagrama de fluxo da rotina de atualização da árvore de Huffman

Algorithm 5 Codificador

Require: Arquivo de entrada f
  $numeroDeSmbolos := 0$ p
  for all todos os símbolos s em f **do**
    $numerodeSmbolos := numerodeSmbolos + 1$
    grava árvore no arquivo
    chama rotina de atualização
  end for
  salva o número de símbolos ao final do arquivo codificado

se que os códigos gerados são sempre códigos de prefixo, a reconstrução dos símbolos consistem no percorrimento da árvore de Huffman, obedecendo o caminho orientado pela sequência de bits lida até uma folha. Este procedimento é ilustrado no algoritmo6 e na figura3

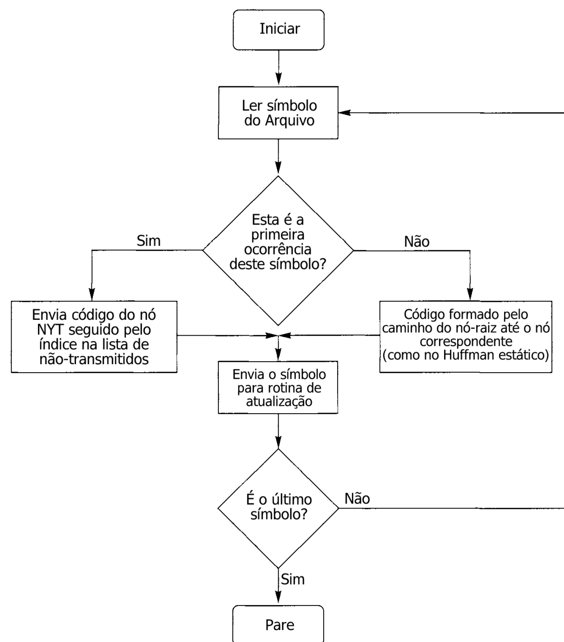

Figura 2: Procedimento de codificação

Algorithm 6 Decodificador

Require: Arquivo de entrada compactado f
  ler o número de símbolos ao final do arquivo f em $numerodeSmbolos$
  while $numerodeSmbolos > 0$ **do**
    $numerodeSmbolos := numerodeSmbolos - 1$
    lê bit-a-bit até atingir um nó-folha, que contém o símbolo
    chama o rotina de atualização da árvore
  end while
  salva os símbolos no arquivo descompactado

---

3 Detalhes de Implementação

Nosso trabalho consistiu na implementação do algoritmo dinâmico de Huffman, tendo como referência o livro de Khalid Sayood[1], o serviço de codificação e decodificação foram divididos em duas classes independentes, `HuffmanCoder` e `HuffmanDecoder`, ambas herdadas de uma classe abstrata comum, `AdaptativeHuffman`, que fornece o método para atualização dos símbolos na árvore de geração de códigos. Esta estrutura hierárquica foi escolhida tanto por motivos abstratos quanto práticos, a fim de facilitar a legibilidade, coesão e reaproveitamento do código. O modelo é simples, onde cada classe possui apenas um método de interface, conforme observamos na Figura4.

A lógica de funcionado destas classes estão detalhados nos diagramas de fluxos. O código relativo a estas interfaces estão declarados nos arquivos `AdaptativeHuffman.h`,

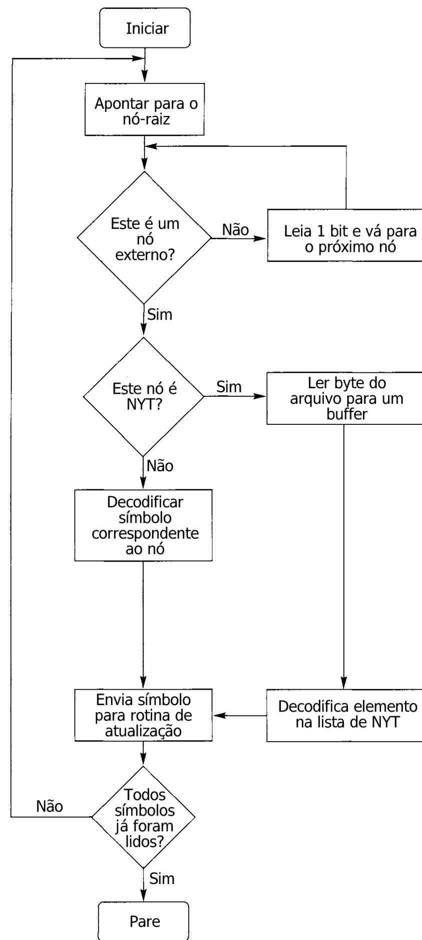

Figura 3: Procedimento de decodificação

`HuffmanEncoder.h` e `HuffmanDecoder.h`. As seguintes seções possuem o código das classes, exemplo de uso e análise da eficiência.

4 Resultados da Implementação

Para testar o algoritmo implementado, diversos tipos de arquivos foram utilizados. Para verificar se o arquivo descompactado possui exatamente o mesmo conteúdo do arquivo original, foram utilizados os algoritmos `MD5` e `SHA1` para comparação. A taxa de compactação é apresentada na tabela4.

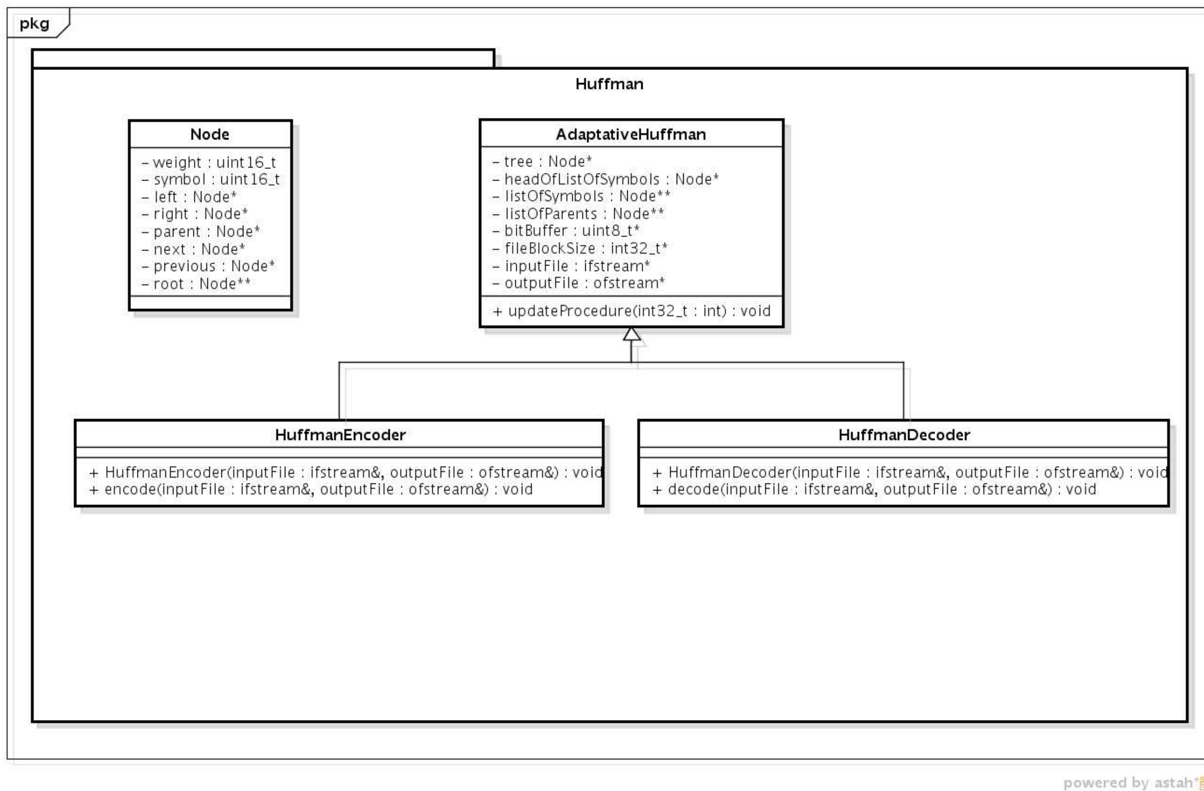

Figura 4: Diagrama de Classes do Projeto

Arquivo	Tamanho Original	Tamanho Compactado	Taxa de Compressão
article.pdf	3.2Mb	3.2Mb	0
lena.eps	328K	265K	1.237
restored.bmp	242K	125K	1.936
test.xls	136K	61K	2.229

Referências

[1] Sayood, Khalid. *Introduction to data compression (2nd ed.)*. 1-55860-558-4. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA. 2000.

[2] Vitter, J.S. *Dynamic Huffman Coding*. ACM Trans. Math. Sojlw. Submitted 1986.

[3] Gallager, R. G. V*ariations on a theme by Huffman. IEEE Trans. Inj Theory IT-24*, 6 (Nov.1978), 668-674.

[4] Faller,N. *An adaptive system for data compression*. In Record ofthe 7th Asilomar Conference on Circuits, Systems, and Computers. 1913, pp. 593-591.

[5] Knuth, D. E. *Dynamic Huffman coding*. J. Algorithms 6 (1985), 163-180.

[6] Huffman, D. A. A method for the construction of minimum redundancy codes. In Proc. IRE 40 (1951), 1098-1101.

Apêndice A Estrutura de Dados – Node.h

Esta implementação utiliza uma classe para representar a árvore de Huffman, que pode ser encontrada no arquivo `Node.h`. Nesta classe, definimos apenas as referências para os nós-filhos e para uma lista que mapeia todos os outros nós, ordenada pelos pesos.

```
/*
 * Node.h
 *
 *  Created on: Apr 26, 2011
 *      Author: pedrogarcia
 */

#ifndef NODE_H_
#define NODE_H_

#include <stdint.h>

class Node
{
    public:
        uint16_t weight;
        uint16_t symbol;
        Node* left;
        Node* right;
        Node* parent;
        Node* next;
        Node* previous;
        Node** root;
        Node();
};
#endif /* NODE_H_ */
```

Apêndice B Interfaces

Apêndice B.1 AdaptativeHuffman.h

```
/*
 * AdaptativeHuffman.h
 *
 *  Created on: Apr 26, 2011
 *      Author: pedrogarcia
```

```
 */

#ifndef ADAPTATIVEHUFFMAN_H_
#define ADAPTATIVEHUFFMAN_H_

#include <iostream>
using std::cout;
using std::endl;
using std::cerr;
using std::ios;
using std::ios_base;
using std::ifstream;
using std::ofstream;

#include <fstream>
using std::fstream;

#include "Node.h"
#include "sharedconfig.h"

class AdaptativeHuffman
{
    public:
        AdaptativeHuffman();
        virtual ~AdaptativeHuffman();

    protected:
        Node* tree;
        Node* headOflistOfSymbols;
        Node* listOfSymbols[NODE_ORDER_LIST_SIZE];
        Node** listOfParents;
        uint8_t bitBuffer[128];
        int32_t fileBlockSize;
        ifstream* inputFile;
        ofstream* outputFile;

        void updateProcedure(int32_t);
    private:
        void eraseParentNode(Node**);
        bool isFirstAppearanceforSymbol(int32_t);
        Node** getParentNode(void);
        void incrementNodeWeight(Node*);
        void givesBirthToNewNYTandExternalNode(int32_t, Node*, Node*);
```

```
        void swapHorizontal(Node*, Node*);
        void swapVertical(Node*, Node*);
};

#endif /* ADAPTATIVEHUFFMAN_H_ */
```

Apêndice B.2 HuffmanEncoder.h

```
/*
 * HuffmanEncoder.h
 *
 *  Created on: Apr 26, 2011
 *      Author: pedrogarcia
 */

#ifndef HUFFMANENCODER_H_
#define HUFFMANENCODER_H_

#include <iostream>
using std::cout;
using std::endl;
using std::cerr;
using std::ios;
using std::ios_base;
using std::ifstream;
using std::ofstream;

#include <fstream>
using std::fstream;

#include <cstdio>

#include "sharedconfig.h"
#include "Node.h"
#include "AdaptativeHuffman.h"

class HuffmanEncoder: public AdaptativeHuffman
{
    public:
        HuffmanEncoder(ifstream&, ofstream&);
        virtual ~HuffmanEncoder();
    private:
```

```
        void encode(ifstream&, ofstream&);
        void flushBuffer(void);
        bool isTheFirstAppearanceOfSybol(int32_t);
        bool isThisTheLastSymbol(void);
        void sendCodeForNYT(Node*, Node*);
        void sendSymbolToFile(int32_t);
        void startVariables(void);
        void writeBitBuffer(int32_t);
        void writeNumberOfSymbols(int32_t);
};

#endif /* HUFFMANENCODER_H_ */
```

Apêndice B.3 HuffmanDecoder.h

```
/*
 * HuffmanDecoder.h
 *
 *  Created on: Apr 26, 2011
 *      Author: pedrogarcia
 */

#ifndef HUFFMANDECODER_H_
#define HUFFMANDECODER_H_

#include <iostream>
using std::cout;
using std::endl;
using std::cerr;
using std::ios;
using std::ios_base;
using std::ifstream;
using std::ofstream;

#include <fstream>
using std::fstream;

#include "AdaptativeHuffman.h"
#include "sharedconfig.h"
#include "Node.h"

class HuffmanDecoder: public AdaptativeHuffman
```

```
{
    public:
        HuffmanDecoder(ifstream&, ofstream&);
        virtual ~HuffmanDecoder();
    private:
        Node *nodeOrderListTail;

        void decode(ifstream&, ofstream&);
        int32_t getASymbol(int32_t&);
        int32_t readBitBuffer();
        int32_t readNumberOfSymbols(void);
        int32_t restoreSymbol(Node*, int32_t*);
        void startVariables(void);
        void writeSymbol(int32_t&);
};

#endif /* HUFFMANDECODER_H_ */
```

Apêndice C Implementações

Apêndice C.1 AdaptativeHuffman.cpp

```
/*
 * AdaptativeHuffman.cpp
 *
 *  Created on: Apr 26, 2011
 *      Author: pedrogarcia
 */

#include <cstdlib>
#include <cstdio>
#include "AdaptativeHuffman.h"
#include "sharedconfig.h"
#include "Node.h"

AdaptativeHuffman::AdaptativeHuffman()
{
}

AdaptativeHuffman::~AdaptativeHuffman()
{
}
```

```
void AdaptativeHuffman::updateProcedure(int32_t symbol)
{
    Node* newNYT;
    Node* externalNode;

    if (isFirstAppearanceforSymbol(symbol)) {
        newNYT = new Node();
        externalNode = new Node();
        givesBirthToNewNYTandExternalNode(symbol, externalNode, newNYT);
        incrementNodeWeight(externalNode->parent);
    } else {
        incrementNodeWeight(listOfSymbols[symbol]);
    }
}

void AdaptativeHuffman::eraseParentNode(Node** parentNode)
{
    *parentNode = (Node *) listOfParents;
    listOfParents = parentNode;
}
bool AdaptativeHuffman::isFirstAppearanceforSymbol(int32_t symbol)
{
    return (listOfSymbols[symbol] == NULL);
}

Node** AdaptativeHuffman::getParentNode(void)
{
    if (listOfParents == NULL)
        return ((Node **) (new Node*));
    else {
        Node** parentNode = listOfParents;
        listOfParents = (Node **) *parentNode;
        return parentNode;
    }
}

void AdaptativeHuffman::givesBirthToNewNYTandExternalNode(int32_t symbol,
                                                           Node *externalNode,
                                                           Node *newNYT)
{
    externalNode->symbol = INTERNAL_NODE;
    externalNode->weight = ONE;
```

```
externalNode->next = headOflistOfSymbols->next;
if (headOflistOfSymbols->next != LEAF) {
    headOflistOfSymbols->next->previous = externalNode;
    if (headOflistOfSymbols->next->weight == ONE) {
        externalNode->root = headOflistOfSymbols->next->root;
    } else {
        externalNode->root = getParentNode();
        *externalNode->root = externalNode;
    }
} else {
    externalNode->root = getParentNode();
    *externalNode->root = externalNode;
}
headOflistOfSymbols->next = externalNode;
externalNode->previous = headOflistOfSymbols;

newNYT->symbol = symbol;
newNYT->weight = ONE;
newNYT->next = headOflistOfSymbols->next;
if (headOflistOfSymbols->next != LEAF) {
    headOflistOfSymbols->next->previous = newNYT;
    if (headOflistOfSymbols->next->weight == ONE) {
        newNYT->root = headOflistOfSymbols->next->root;
    } else {
        newNYT->root = getParentNode();
        *newNYT->root = externalNode;
    }
} else {
    newNYT->root = getParentNode();
    *newNYT->root = newNYT;
}
headOflistOfSymbols->next = newNYT;
newNYT->previous = headOflistOfSymbols;
newNYT->left = newNYT->right = NULL;

if (headOflistOfSymbols->parent != NULL) {
    if (headOflistOfSymbols->parent->left == headOflistOfSymbols) {
        headOflistOfSymbols->parent->left = externalNode;
    } else {
        headOflistOfSymbols->parent->right = externalNode;
    }
} else {
    tree = externalNode;
```

```
    }

    externalNode->right = newNYT;
    externalNode->left = headOflistOfSymbols;
    externalNode->parent = headOflistOfSymbols->parent;
    headOflistOfSymbols->parent = externalNode;
    newNYT->parent = externalNode;
    listOfSymbols[symbol] = newNYT;
}

void AdaptativeHuffman::incrementNodeWeight(Node* node)
{
    Node* externalNode;

    if (node == LEAF)
        return;

    if ((node->next != LEAF) && (node->next->weight == node->weight)) {
        externalNode = *node->root;
        if (externalNode != node->parent)
            swapVertical(externalNode, node);
        swapHorizontal(externalNode, node);
    }
    if (node->previous && node->previous->weight == node->weight) {
        *node->root = node->previous;
    } else {
        *node->root = NULL;
        eraseParentNode(node->root);
    }
    node->weight++;
    if (node->next && node->next->weight == node->weight) {
        node->root = node->next->root;
    } else {
        node->root = getParentNode();
        *node->root = node;
    }
    if (node->parent != LEAF) {
        incrementNodeWeight(node->parent);
        if (node->previous == node->parent) {
            swapHorizontal(node, node->parent);
            if (*node->root == node)
                *node->root = node->parent;
        }
```

```
    }
}

void AdaptativeHuffman::swapHorizontal(Node* node1, Node* node2)
{
    Node* aux;

    aux = node1->next;
    node1->next = node2->next;
    node2->next = aux;

    aux = node1->previous;
    node1->previous = node2->previous;
    node2->previous = aux;

    if (node1->next == node1)
        node1->next = node2;
    if (node2->next == node2)
        node2->next = node1;

    if (node1->next != LEAF)
        node1->next->previous = node1;
    if (node2->next != LEAF)
        node2->next->previous = node2;
    if (node1->previous != LEAF)
        node1->previous->next = node1;
    if (node2->previous != LEAF)
        node2->previous->next = node2;
}

void AdaptativeHuffman::swapVertical(Node* node1, Node* node2)
{
    Node* parent1;
    Node* parent2;

    parent1 = node1->parent;
    parent2 = node2->parent;

    if (parent1 != LEAF) {
        if (parent1->left == node1) {
            parent1->left = node2;
        } else {
            parent1->right = node2;
```

```
        }
    } else {
        tree = node2;
    }

    if (parent2 != LEAF) {
        if (parent2->left == node2) {
            parent2->left = node1;
        } else {
            parent2->right = node1;
        }
    } else {
        tree = node1;
    }
    node1->parent = parent2;
    node2->parent = parent1;
}
```

Apêndice C.2 HuffmanEncoder.cpp

```
/*
 * HuffmanEncoder.cpp
 *
 *  Created on: Apr 26, 2011
 *      Author: pedrogarcia
 */

#include <iostream>
using std::cout;
using std::endl;
using std::cerr;
using std::ios;
using std::ios_base;
using std::ifstream;
using std::ofstream;

#include <fstream>
using std::fstream;

#include <cstdio>
#include <cstdlib>
#include <cstring>
```

```
#include "HuffmanEncoder.h"
#include "sharedconfig.h"
#include "Node.h"

HuffmanEncoder::HuffmanEncoder(ifstream &input, ofstream &output)
{
    ofstream obuff;
    ifstream ibuff;

    // two-steps Huffman Decoding
    // first step
    obuff.open(".buffer.buff", ios::out | ios::binary);
    encode(input, obuff);
    input.close();
    obuff.flush();
    obuff.close();

    // second step
    ibuff.open(".buffer.buff", ios::in | ios::binary);
    encode(ibuff, output);
    ibuff.close();
    output.close();
    remove(".buffer.buff");
}

HuffmanEncoder::~HuffmanEncoder()
{
}

void HuffmanEncoder::encode(ifstream &input, ofstream &output)
{
    int32_t symbol = ZERO;
    int32_t numberOfSymbols = ZERO;

    inputFile = &input;
    outputFile = &output;
    startVariables();

    while (inputFile->read(reinterpret_cast<char *> (&symbol), sizeof(uint8_t))
        && !isThisTheLastSymbol()) {
        numberOfSymbols++;
        sendSymbolToFile(symbol);
```

```
        updateProcedure(symbol);
    }
    writeNumberOfSymbols(numberOfSymbols);
}

void HuffmanEncoder::flushBuffer()
{
    outputFile->write(reinterpret_cast<char *> (bitBuffer), sizeof(uint8_t)
        * ((fileBlockSize + 7) >> 3));
    memset(bitBuffer, 0, sizeof(bitBuffer));
    fileBlockSize = 0;
}

bool HuffmanEncoder::isTheFirstAppearanceOfSybol(int32_t symbol)
{
    return (listOfSymbols[symbol] == NULL);
}

bool HuffmanEncoder::isThisTheLastSymbol(void)
{
    return (inputFile->eof());
}

void HuffmanEncoder::sendCodeForNYT(Node* node, Node* child)
{
    if (node->parent)
        sendCodeForNYT(node->parent, node);
    if (child != LEAF) {
        if (node->right == child)
            writeBitBuffer(ONE);
        else {
            writeBitBuffer(ZERO);
        }
    }
}

void HuffmanEncoder::sendSymbolToFile(int32_t symbol)
{
    if (isTheFirstAppearanceOfSybol(symbol)) {
        sendSymbolToFile(NYT);
        for (int32_t bits = SYMBOL_BITSIZE - 1; bits >= 0; bits--)
            writeBitBuffer((symbol >> bits) & TRUE);
    } else {
```

```
        sendCodeForNYT(listOfSymbols[symbol], LEAF);
    }
}
void HuffmanEncoder::startVariables(void)
{
    memset(bitBuffer, ZERO, sizeof(bitBuffer));
    memset(listOfSymbols, NULL, sizeof(listOfSymbols));
    tree = new Node();
    tree->weight = ZERO;
    tree->symbol = NYT;
    tree->parent = LEAF;
    tree->left = LEAF;
    tree->right = LEAF;
    tree->next = LEAF;
    tree->previous = LEAF;
    listOfParents = LEAF;
    headOflistOfSymbols = tree;
    listOfSymbols[NYT] = tree;
    fileBlockSize = ZERO;
}

void HuffmanEncoder::writeBitBuffer(int32_t bit)
{
    if (fileBlockSize == READED_BLOCK_SIZE) {
        outputFile->write(reinterpret_cast<char *> (bitBuffer), BUFFER_SIZE
            * sizeof(uint8_t));
        memset(bitBuffer, ZERO, sizeof(bitBuffer));
        fileBlockSize = ZERO;
    }
    int32_t shiftSize = (7 - (fileBlockSize % 8));
    bitBuffer[fileBlockSize >> 3] |= bit << shiftSize;
    fileBlockSize++;
}

void HuffmanEncoder::writeNumberOfSymbols(int32_t numberOfSymbols)
{
    flushBuffer();
    for (int32_t bits = 31; bits >= 0; bits--)
        writeBitBuffer((numberOfSymbols >> bits) & TRUE);
    flushBuffer();
}
```

Apêndice C.3 HuffmanDecoder.cpp

```
/*
 * HuffmanDecoder.cpp
 *
 *  Created on: Apr 26, 2011
 *      Author: pedrogarcia
 */

#include <iostream>
using std::cout;
using std::endl;
using std::cerr;
using std::ios;

#include <fstream>
using std::fstream;

#include <cstdio>
#include <cstdlib>
#include <cstring>

#include "HuffmanDecoder.h"
#include "sharedconfig.h"
#include "Node.h"

HuffmanDecoder::HuffmanDecoder(ifstream &input, ofstream &output)
{
    ofstream obuff;

    // first step
    obuff.open(".buffer.buff", ios::out | ios::binary);
    decode(input, obuff);
    input.close();
    obuff.flush();
    obuff.close();

    // second step
    ifstream ibuff;
    ibuff.open(".buffer.buff", ios::in | ios::binary);
    decode(ibuff, output);
    ibuff.close();
    output.close();
```

```
    remove(".buffer.buff");
}

HuffmanDecoder::~HuffmanDecoder()
{
}

void HuffmanDecoder::decode(ifstream &input, ofstream &output)
{
    int32_t symbol = ZERO;
    int32_t numberOfSymbols = ZERO;

    inputFile = &input;
    outputFile = &output;
    startVariables();

    numberOfSymbols = readNumberOfSymbols();
    inputFile->seekg(0, ios::beg);

    while (numberOfSymbols--) {
        symbol = getASymbol(symbol);
        writeSymbol(symbol);
        updateProcedure(symbol);
    }
}

int32_t HuffmanDecoder::getASymbol(int32_t &symbol)
{
    restoreSymbol(tree, &symbol);
    if (symbol == NYT) {
        symbol = ZERO;
        for (int32_t i = ZERO; i < SYMBOL_BITSIZE; i++) {
            int32_t step = symbol << ONE;
            symbol = step + readBitBuffer();
        }
    }
    return symbol;
}

int32_t HuffmanDecoder::readBitBuffer()
{
    if (fileBlockSize == READED_BLOCK_SIZE) {
        inputFile->read(reinterpret_cast<char *> (bitBuffer), BUFFER_SIZE
```

```
            * sizeof(uint8_t));
        fileBlockSize = 0;
    }
    int32_t shiftSize = 7 - (fileBlockSize % 8);
    int32_t shift = bitBuffer[fileBlockSize >> 3] >> shiftSize;
    int32_t maskShift = shift & TRUE;
    fileBlockSize++;
    return maskShift;
}

int32_t HuffmanDecoder::readNumberOfSymbols(void)
{
    int32_t numberOfSymbols = ZERO;
    uint8_t lastFourBytesOfFile[4];

    if (!inputFile->seekg(NUMBER_OF_SYMBOLS_BYTESIZE, ios::end)) {
        exit(EXIT_FAILURE);
    }

    inputFile->read(reinterpret_cast<char *> (&lastFourBytesOfFile),
                    sizeof(lastFourBytesOfFile) * sizeof(uint8_t));
    for (int32_t bytes = 0; bytes < 4; bytes++) {
        for (int32_t bit = 7; bit >= 0; bit--) {
            int32_t symbols = (lastFourBytesOfFile[bytes] >> bit) & TRUE;
            int32_t step = numberOfSymbols << ONE;
            numberOfSymbols = step + symbols;
        }
    }
    return numberOfSymbols;
}

int32_t HuffmanDecoder::restoreSymbol(Node* node, int32_t* symbol)
{
    while (node && node->symbol == INTERNAL_NODE) {
        int32_t bit = readBitBuffer();
        if (bit == ONE)
            node = node->right;
        else
            node = node->left;
    }
    if (node == NULL) {
        exit(EXIT_FAILURE);
    }
```

```
    return (*symbol = node->symbol);
}
void HuffmanDecoder::startVariables(void)
{
    memset(bitBuffer, ZERO, sizeof(bitBuffer));
    memset(listOfSymbols, NULL, sizeof(listOfSymbols));
    tree = new Node();
    tree->weight = ZERO;
    tree->symbol = NYT;
    tree->parent = LEAF;
    tree->left = LEAF;
    tree->right = LEAF;
    tree->next = LEAF;
    tree->previous = LEAF;
    listOfParents = LEAF;
    headOflistOfSymbols = tree;
    listOfSymbols[NYT] = tree;
    nodeOrderListTail = tree;
    fileBlockSize = READED_BLOCK_SIZE;
}

void HuffmanDecoder::writeSymbol(int32_t &symbol)
{
    outputFile->write(reinterpret_cast<char *> (&symbol), sizeof(uint8_t));
}
```

Apêndice D Função Principal – Utilização das Classes

```
//============================================================================
// Name        : AdaptativeHuffman.cpp
// Author      : Pedro Garcia
// Version     :
// Copyright   : 2010 Pedro Garcia
// Description : Huffman encoder and decoder
//============================================================================

#include <iostream>
using std::cout;
using std::endl;
using std::cerr;
using std::ios;
```

```
using std::ifstream;
using std::ofstream;

#include <fstream>
using std::fstream;

#include <cstring>
using std::string;

#include <cstdlib>

#include "HuffmanEncoder.h"
#include "HuffmanDecoder.h"

void howto(char *input)
{
    cerr << "Usage: " << input << " <operation> inputfile outputfile\n";
    cerr << "Possible operations: \n";
    cerr << "\t e -> encode \n\t d -> decode" << endl;
}

int main(int argc, char **argv)
{
    ifstream fin;
    ofstream fout;
    AdaptativeHuffman *huff;

    if (argc != 4) {
        howto(argv[0]);
        exit(EXIT_FAILURE);
    }

    fin.open(argv[2], ios::in | ios::binary);
    fout.open(argv[3], ios::out | ios::binary);

    if (!fin | !fout) {
        howto(argv[0]);
        return EXIT_FAILURE;
    }

    if (*argv[1] == 'd') {
        cout << "Decoding..." << endl;
        huff = new HuffmanDecoder(fin, fout);
```

```
        delete huff;
    } else {
        cout << "Encoding..." << endl;
        huff = new HuffmanEncoder(fin, fout);
        delete huff;
    }

    cout << "Done" << endl;
    return EXIT_SUCCESS;
}
```