Faculdade de Engenharia da Universidade do Porto

Compressão e Descompressão de Ficheiros

Concepção e Análise de Algoritmos

Mestrado Integrado em Engenharia Informática e Computação

Autores do projecto:

João Costa, ei09008@fe.up.pt
Nelson Oliveira, ei09027@fe.up.pt

Indice

Introdução

Para este segundoo trabalho prático, o grupo decidiu escolher o tema 2, que tinha como objectivo a compressão de ficheiros de texto. Estes ficheiros seriam então comprimidos usando os Algoritmos de Huffman e LZW (Lempel-Ziv-Welch), tendo como objectivo diminuir o tamanho de cada ficheiro. Ambos os algoritmos são diferentes, utilizando diferentes técnicas de compressão, como veremos à frente.

Entretanto, o grupo, como pedido nos objectivos do trabalho, realizou as seguintes tarefas:

- Compressão utilizando o Algoritmo de Huffman, com a ajuda de uma árvore binária;
- Compressão utilizando o Algoritmo LZW, com a ajuda de estruturas map para guardar o dicionário de compressão/descompressão.

Descrição dos algoritmos

Códigos de Huffman

O funcionamento deste algoritmo baseia-se na possibilidade de representar diferentes caracteres através de sequências de bits de diferentes tamanhos, ao invés de serem todos representados como sequências de 8 bit. No entanto, ao fazer isto, uns caracteres terão de ocupar mais de 8 bit.

Visto que o objectivo é comprimir o texto, os caracteres que aparecem com maior frequência ficarão com as representações de menor comprimento, ficando os caracteres com menor frequência com as representações de comprimento maior.

Ficheiro antes da compressão: Sherlock.txt – 6.2MB

Ficheiro após a conversão: Sherlock.huf – 3.5MB

Lempel-Ziv-Welch (LZW)

Ao contrário dos códigos de Huffman, o algoritmo LZW não se prende à frequência dos caracteres ou à possibilidade de os representar com menos de 8 bit, mas sim com a possibilidade de criar valores para novos caracteres que representem sequências de caracteres. Estas sequências vão sendo geradas conforme o ficheiro é carregado, sendo que se houver alguma sequência que seja repetida várias vezes, esta vai ser simplificada para um carácter só (se bem que, neste método, os caracteres são representados com mais de 8 bit).

Ficheiro antes da compressão: Sherlock.txt – 6.2MB

Ficheiro após a conversão: Sherlock.lzw – 1.9MB

Estruturas de Dados Criadas/Usadas

Para uma melhor estrutura do trabalho, o grupo dividiu tudo em certas classes. Vejamos então como estão divididas/detalhadas.

Classe Codec

A Classe Codec tem como objectivo criar objectos do tipo CodecHuffman e CodecLZW, que são subclasses de Codec, que contém membros virtuais como decomprimir(). A gestão da classe torna-se portanto mais fácil. No caso de LZW, o comprimir ficou bastante mais facil fazer com um vector de inteiros, para a sua rápida gravação no ficheiro binário.

Classe ArvoreBinaria

A Classe ArvoreBinaria é a classe que contém toda a informação acerca da Arvore Binária a usar na compressão/descompressão dos ficheiros de texto utilizados pelos códigos de Huffman, contendo Nós, Edges, entre outros. O código fonte do ficheiro ArvoreBinaria.h pode ser consultado nos Anexos que se encontram no fim do trabalho.

Dificuldades Encontradas

Durante a realização deste projecto foram encontradas várias dificuldades, tendo sido uma delas a compressão de dados de um ficheiro de texto e a sua escrita em binário correctamente para os ficheiros. Entretanto, a descompessão revelou-se ainda mais complicada, visto que ler os caracteres em binário estava a ser bastante complicado. Estes caracteres eram lidos, mas teriam valores na tabela ASCII entre 0 e 255 à mesma, o que dificultava a descompressão. Isto para ambos os algoritmos.

No entanto, nos Códigos de Huffman, o grupo conseguiu dar a volta usando uma tabela de frequências que ajudava bastante na descompressão/compressão do ficheiro, fazendo com que toda aquela arvore binária fosse mais fácil ler.

Já no algoritmo de LZW, a principal dificuldade foi sem duvida descomprimir quando o ficheiro estava em binário, fazendo com que tudo aquilo que estivesse comprimido fosse complicado de ler. Porquê? Isto porque quando está comprimido está representado como um caracter, e não como a string que supostamente foi comprimida. Exemplo: Foi guardado no dicionário a string “DE” na posição 256 do map. Entretanto, esta string esta definida como um caracter no ficheiro comprimido, não havendo acesso à string correspondente. Solução? Colocar o dicionário pós-compressão no ficheiro.

Conclusão

Com este trabalho, o grupo conseguiu entender um pouco mais acerca dos algoritmos de Huffman e LZW, e como estes são estruturados. Uma das conclusões principais, é que o algoritmo de LZW, comprime sem dúvida, muito mais que os Códigos de Huffman, embora seja muito mais lento. Entretanto não há muito mais a dizer sobre o projecto, esperando que tudo aquilo que foi feito pelo grupo esteja de certa forma, correcta.

Anexos – Todos os comentários foram retirados

Codec.h

```
#ifndef COMPRESSOR_H_
#define COMPRESSOR_H_

#include <string>
#include <vector>
#include <set>
#include <map>

using namespace std;
typedef unsigned char byte;

class Codec {
protected:
	string filename;
public:
	Codec(string file) {filename=file;}
	string virtual decomprimir() = 0;
};

class CodecHuffman: public Codec {
private:
	int tabelaFreq[256];
public:
	CodecHuffman(string file) : Codec(file) {};
	vector<byte> comprimir();
	string decomprimir();
};

class CodecLZW: public Codec {
protected:
	int size;
	map<string,int> dictionaryCompress;
	map<int,string> dictionaryDecompress;
public:
	CodecLZW(string file) : Codec(file)
	{
		int i;
		for (i = 0; i < 256; i++)
		{
		    dictionaryCompress[string(1, i)] = i;
		}
		size = 256;
	};
	vector<int> comprimir();
	string decomprimir();
	void setDicCompress(map<string,int> d)
	{dictionaryCompress = d;}
	map<string,int> getDicCompress()
	{return dictionaryCompress;}
};


#endif
```

ArvoreBinaria.h

```
#ifndef ARVOREBINARIA_H_
#define ARVOREBINARIA_H_

#include <string>
#include <vector>
using namespace std;

struct caracter {
	unsigned char c;
	string representacao;
};

class No {
	private:
		int peso;
		bool folha;
		unsigned char caracter;
		No* esquerda;
		No* direita;
	public:
		No();
		No(unsigned char c,int p);
		void addNo(No* n);
		No* getEsquerda();
		No* getDireita();
		int getPeso();
		unsigned char getChar();
		bool isFolha();
};

No juntaNos(No* n1,No* n2);
vector <caracter> getRepresentacoes(No &n);
void dfs(No* n,string s,vector <caracter> &r);
bool compareNoPtr(No* n1, No* n2);

#endif
```

CodecLZW.cpp (comprimir e decomprimir)

```
vector<int> CodecLZW::comprimir()
{
	vector<int> result;
	int dictSize = 256;
	fstream file;
	file.open(filename.c_str());
	string entry = "";
	string temp;

	while (getline(file,temp))
	{
		entry += temp;
		entry +='\n';
	}
	entry +='\0';
	string w;
	for (string::const_iterator it = entry.begin(); it != entry.end(); ++it)
	{
		char c = *it;
		string wc = w + c;
		if (dictionaryCompress.count(wc))
		  w = wc;
		else
		{
		  result.push_back(dictionaryCompress[w]);
		  dictionaryCompress[wc] = dictSize;
		  dictSize += 1;
		  w = string(1, c);
		}
	}

	if (!w.empty())
		result.push_back(dictionaryCompress[w]);

	size = dictSize;
	return result;
}

string CodecLZW::decomprimir()
{
	for (map<string,int>::iterator it = dictionaryCompress.begin(); it !=
dictionaryCompress.end(); it++)
	{
	    dictionaryDecompress[(*it).second] = (*it).first;
	}
	ifstream file;
	file.open(filename.c_str(),ios_base::binary);
	vector<unsigned int> aux;
	unsigned short int tmp = 1;
	while(tmp != 0)
	{
		file.read((char*)&tmp,sizeof(short int));
		aux.push_back(tmp);
	}

	string result = "";
```

```
	for (vector<unsigned int>::iterator it = aux.begin(); it != aux.end(); it+
+)
	{
		int index = (*it);
		string corres = dictionaryDecompress[index];
		result += corres;
	}

	return result;
}
```

CodeHuffman.cpp (comprimir e decomprimir)

```
vector<byte> CodecHuffman::comprimir() {
	vector <byte> ret;
	for (int n=0;n<256;n++) {tabelaFreq[n]=0;}
	ifstream file(filename.c_str(),ios_base::binary);
	char tempchar;
	while (file.read(&tempchar,1)) {
			tabelaFreq[(byte)tempchar]++;
	}
	tabelaFreq[0]++;
	for (int i=0;i<256;i++) {
		int freq=tabelaFreq[i];
		ret.push_back((char)(freq&0xFF));
		ret.push_back((char)((freq&0xFF00)>>8));
		ret.push_back((char)((freq&0xFF0000)>>16));
		ret.push_back((char)((freq&0xFF000000)>>24));
	}
	vector<No*> arvore;
	for (int n=0;n<256;n++) {
		if (tabelaFreq[n]>0) {arvore.push_back(new No((unsigned char)
n,tabelaFreq[n]));}
	}
	sort(arvore.begin(),arvore.end(),compareNoPtr);
	while (arvore.size()>1) {
		No* novoNo;
		novoNo=new No();
		novoNo->addNo(arvore[0]);
		novoNo->addNo(arvore[1]);
		arvore.push_back(novoNo);
		arvore.erase(arvore.begin());
		arvore.erase(arvore.begin());
		sort(arvore.begin(),arvore.end(),compareNoPtr);
	}

	vector <caracter> representacoes=getRepresentacoes(*arvore[0]);
	string repBuff[256];
	for (size_t n=0;n<representacoes.size();n++) {
		repBuff[(byte)representacoes[n].c]=representacoes[n].representacao;
	}

	string tempOutput;
	file.clear();
	file.seekg(0, ios::beg);
	while (file.read(&tempchar,1)) {
			tempOutput+=repBuff[(byte)tempchar];
			if (tempOutput.size()%8==0) {
				for (size_t i=0;i<tempOutput.size();i+=8) {
					byte newByte=0;
					for (int j=0;j<8;j++) {
						newByte=newByte<<1;
						if (i+j<tempOutput.size() &&
tempOutput[i+j]=='1') {newByte=newByte | 1;}
					}
					ret.push_back(newByte);
				}
				tempOutput.clear();
			}
	}
	file.close();
	tempOutput+=repBuff[0];
```

```
	for (size_t i=0;i<tempOutput.size();i+=8) {
		byte newByte=0;
		for (int j=0;j<8;j++) {
			newByte=newByte<<1;
			if (i+j<tempOutput.size() && tempOutput[i+j]=='1')
{newByte=newByte | 1;}
		}
		ret.push_back(newByte);
	}
	return ret;
}
string CodecHuffman::decomprimir() {
	string ret;
	ifstream file(filename.c_str(),ios_base::binary);
	for (int n=0;n<256;n++) {
		int freq;
		char temp;
		file.read((char*)&freq,sizeof(int));
		tabelaFreq[n]=freq;
	}
	vector<No*> arvore;
	for (int n=0;n<256;n++) {
		if (tabelaFreq[n]>0) {arvore.push_back(new No((unsigned char)
n,tabelaFreq[n]));}
	}

	sort(arvore.begin(),arvore.end(),compareNoPtr);
	while (arvore.size()>1) {
		No* novoNo;
		novoNo=new No();
		novoNo->addNo(arvore[0]);
		novoNo->addNo(arvore[1]);
		arvore.push_back(novoNo);
		arvore.erase(arvore.begin());
		arvore.erase(arvore.begin());
		sort(arvore.begin(),arvore.end(),compareNoPtr);
	}

	No* noActual=arvore[0];
	char decodedChar=-1;
	while (decodedChar!='\0' && !file.eof()) {
		char input;
		file.read((char*)&input,sizeof(char));
		for (int i=0;i<8;i++) {
			if ((input&0x80)==0x00) {noActual=noActual->getEsquerda();} //
0
			else if ((input&0x80)==0x80) {noActual=noActual-
>getDireita();} // 1
			input=input << 1; // shiftLeft
			if (noActual->isFolha()) {
				decodedChar=noActual->getChar();
				ret+=decodedChar;
				noActual=arvore[0];
			}
		}
	}
	file.close();

	return ret;
}
```