Faculdade de Engenharia da Universidade do Porto
Mestrado Integrado em Engenharia Informática e Computação

Compressão de texto

Relatório do Trabalho Prático de CPAL 2007/2008

Paulo André Teixeira Pinto 060509029
Rui Reis Costa Campos 060509080

2MIEIC5

21 de Abril de 2008

Compressão de Texto

Porto, Abril de 2008

Ano: 2º
Semestre: 2º
Grupo: 5.12
Turma: 2MIEIC5

Nº060509029	Paulo André Teixeira Pinto
Nº060509080	Rui Reis Costa Campos

Docente:

Dr. Eng.º João Carlos Pascoal de Faria

Relatório realizado no âmbito do trabalho de Compressão de Texto da disciplina de Complementos de Programação e Algoritmos do 2º Ano do 2º Semestre do Mestrado Integrado em Engenharia Informática e Computação.

Declaração de originalidade

Os autores declaram que o relatório e código fonte submetido é da sua autoria, excepto nas partes explicitamente assinaladas com referência à respectiva fonte. A utilização de uma fonte não referenciada implica a anulação do trabalho.

_____

_____

Índice

1 Introdução

Este trabalho realizado na disciplina de Complementos de Programação e Algoritmos tem como objectivo a realização de um programa em linguagem de programação Java, expandindo assim, as capacidades no desenvolvimento de programas orientados por objectos, sistematizando os conceitos relativos a técnicas de concepção de algoritmos, testes unitários e *"test-driven development*" e análise dinâmica de desempenho *("profiling").*

O objectivo do programa é a criação de software responsável pela compressão de texto. De modo a, implementar a compressão de texto sem perdas foram usados os códigos de Huffman e o algoritmo de Lempel-Ziv-Welch (LZW). O software desenvolvido tem mais especificamente a finalidade de compressão e a correspondente descompressão, comparando o ficheiro descomprimido e o original e verificando que se tratam de ficheiros idênticos, por esse motivo os algoritmos utilizados são mencionados como sem perdas. O programa desenvolvido interage com o utilizador perguntando qual o ficheiro que deseja comprimir e qual o nome do ficheiro que será criado pela compressão, caso o pretendido seja descomprimir recebe o nome do ficheiro comprimido e do novo ficheiro descomprimido.

2 Desenvolvimento

2.1 Algoritmos e estruturas de dados

O trabalho baseia se principalmente em dois algoritmos: Huffman e Lempel-Ziv-Welch (LZW). Ambos os algoritmos foram implementados com o objectivo de comprimir texto e descomprimir sem perdas.

Algoritmo LZW

O LZW (Lempel-Ziv-Welch) é um algoritmo de compressão de dados, derivado do algoritmo LZ78, baseado na localização e no registo dos padrões de uma estrutura. Foi desenvolvido e patenteado em 1984 por Terry Welch.

Para este efeito existe um dicionário que é inicializado com todos os símbolos do alfabeto (ao se usar codificação ASCII são 256 símbolos, de 0 a 255). A entrada é lida e acumulada numa cadeia de caracteres que vamos chamar de P. Sempre que a sequência contida em P não estiver presente no dicionário emitimos o código correspondente a versão anterior de P (ou seja, P sem o último carácter) e adicionamos P ao dicionário. P volta a ser inicializado com o último carácter lido (o seu último carácter) e o processo continua até que não hajam mais caracteres na entrada.

raiz……..... carácter individual
string……... uma sequência de um ou mais caracteres
palavra código…….valor associado a uma string
dicionário……tabela que relaciona palavras código e strings
P……string que representa um prefixo
C……... carácter
cW…….... palavra código
pW…… palavra código que representa um prefixo
X <= Y……..string X assume o valor da string Y
X+Y…….. concatenação das string X e Y
string(w)……...string correspondente à palavra código w

Codificação - Figura 1

1. No início o dicionário contém todas as raízes possíveis e P é vazio;
2. C <= próximo carácter da sequência de entrada;
3. A *string* P+C existe no dicionário?
   a. se *sim*,
      i. P <= P+C;
   b. se *não*,
      i. coloque a palavra código correspondente a P na sequência codificada;
      ii. adicione a *string* P+C ao dicionário;
      iii. P <= C;
4. Existem mais caracteres na sequência de entrada?
   a. se *sim*,

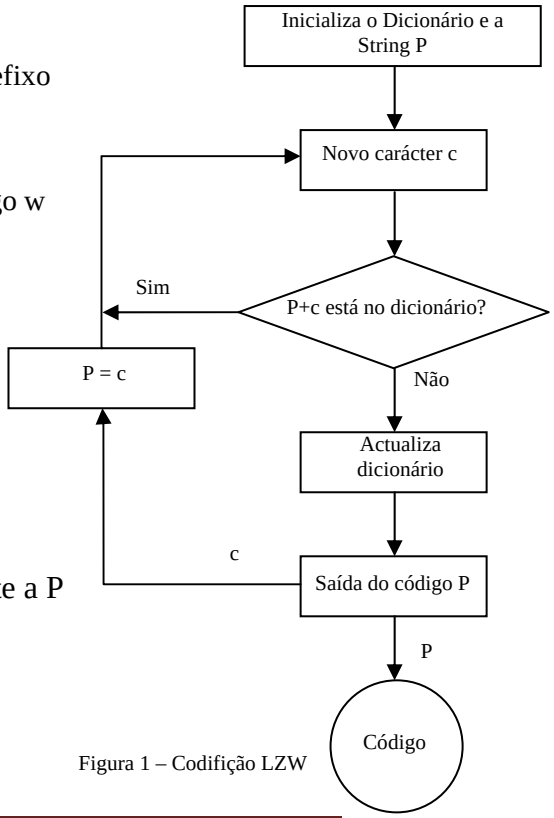

Figura 1 – Codifição LZW

   i. volte ao passo 2;
b. se *não*,
   i. coloque a palavra código correspondente a P na sequência codificada;
   ii. FIM.

Descodificação – Figura 2

1. No início o dicionário contém todas as raízes possíveis;
2. cW <= primeira palavra código na sequência codificada (sempre é uma raiz);
3. Coloque a *string*(cW) na sequência de saída;
4. pW <= cW;
5. cW <= próxima palavra código da sequência codificada;
6. A *string*(cW) existe no dicionário ?
   a. se sim,
      i. coloque a *string*(cW) na sequência de saída;
      ii. P <= *string*(pW);
      iii. C <= primeiro carácter da *string*(cW);
      iv. adicione a *string* P+C ao dicionário;
   b. se não,
      i. P <= *string*(pW);
      ii. C <= primeiro carácter da *string*(pW);
      iii. coloque a *string* P+C na sequência de saída e adicione-a ao dicionário;
7. Existem mais palavras código na sequência codificada?
   a. se sim,
      i. volte ao passo 4;
   b. se não,
      i. FIM.

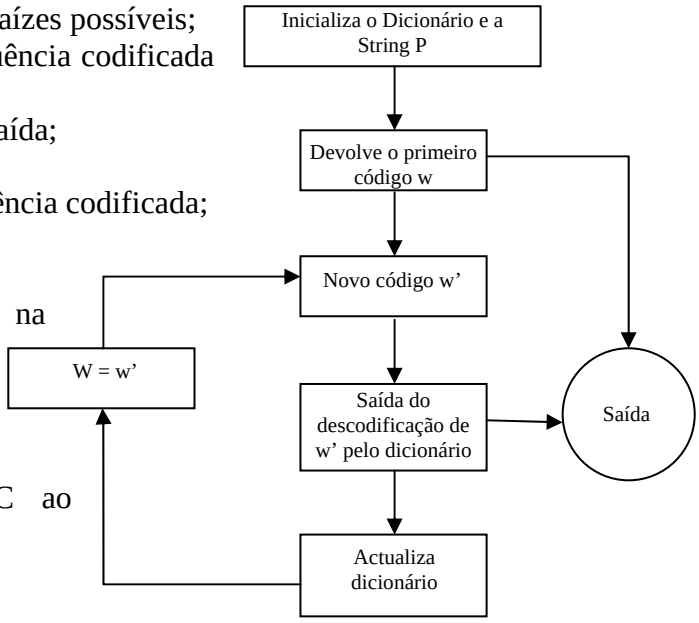

Figura 2 – Descodificação LZW

Estruturas de dados

Na implementação do algoritmo LZW de modo a tornar mais eficiente foram usadas duas tabelas de *Hash* para guardar o dicionário (Palavra>código ; código>Palavra), uma necessária para compressão e a outra para descompressão. Como as consultas e a inserções na tabela de *Hash* tem um peso O(1), a eficiência do algoritmo é da ordem de O(n).

Dependendo do tamanho do arquivo o dicionário pode crescer deliberadamente ocupando grande espaço de memória. Para solucionar isso, limitou-se o tamanho do dicionário usando palavras de código de 16bits ou seja no máximo 65536 palavras de código.

Algoritmo de Huffman

Uma aplicação interessante de árvores binárias em codificação de informação a ser transmitida e compactação de arquivos são os códigos de Huffman.

Para enviar uma mensagem através de um cabo de transmissão, os caracteres da mensagem são enviados um a um, modificados através de alguma tabela de codificação. Em geral, este código forma um número binário. Como a velocidade de transmissão é importante, é interessante tornar a mensagem tão curta quanto possível sem perder a capacidade de descodificar o texto enviado.

A ideia consiste em associar números binários com menos bits aos caracteres mais usados num texto.

O algoritmo de Huffman que será implementado para codificar um arquivo consiste em três fases:

1. Na primeira fase a frequência de cada carácter que ocorre no texto é calculada;
2. A segunda fase do algoritmo de Huffman consiste em construir uma arvore binária baseada na frequência de uso destas letras de modo que as mais frequentes apareçam mais perto da raiz que as menos frequentes. A esta árvore chamamos de árvore binária de Huffman.
   As folhas de cada uma destas árvores correspondem a um conjunto de caracteres que ocorrem no texto. A raiz de cada uma destas árvores será associado um número que corresponde á frequência com que os caracteres associados as folham desta árvore ocorrem no texto;
3. Finalmente na terceira fase a árvore de Huffman será usada para codificar e descodificar texto.

A árvore de Huffman construída durante a codificação será utilizada na descodificação (logo a arvore de Huffman deverá ser armazenada junto com o texto).

Suponhamos o seguinte exemplo:

Tabela 1 – Frequências de caracteres

Caracteres	***A***	***B***	***C***	***D***	***E***	***R***
Frequência	22	9	10	12	16	8

O algoritmo de Huffman constrói uma árvore baseada na frequência de cada carácter, deste modo as mais frequentes serão: A e E, e as que aparecem menos vezes: B e R.

A construção desta arvore e feita das folhas para a raiz, começando a partir das letras menos usadas até atingir a raiz. Nesta árvore, as letras serão representadas nas folhas e os seus vértices internos conterão um número correspondente á soma das frequências dos seus descendentes.

Inicialmente, cada uma das letras será uma arvore composta apenas pela raiz e cujo conteúdo é a frequência com que a letra ocorre no texto – Figura 3.

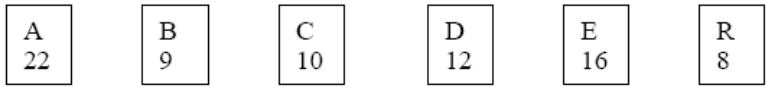

Figura 3 – Floresta no início do algoritmo

De seguida escolhe-se as duas raízes de menor frequência (Arvores com R e B com frequência 8 e 9, respectivamente) e junta-se essas duas árvores, formando uma árvore, que fará parte da colecção – Figura 4.

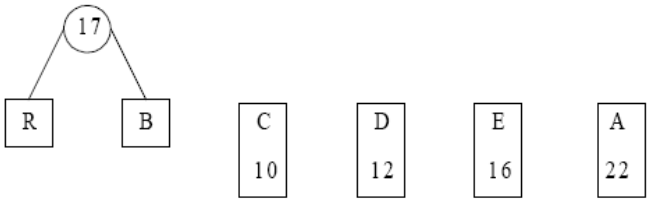

Figura 4 – Floresta após junção de duas árvores (das frequências 8 e 9)

Repete-se o processo, escolhe-se as árvores com raízes 10 e 12, e juntam-se. Segue-se o mesmo raciocino e conclui-se a construção da arvore de Huffman – Figura 5.

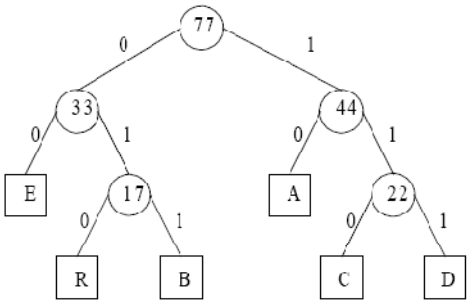

Figura 5 – Árvore de Huffman

Codificação:

Associa-se 0 às arestas da árvore de Huffman que ligam um nó com o seu filho esquerdo e 1, às arestas que ligam um nó com o seu filho direito. O código correspondente a cada letra será formado pelo número binário associado ao caminho da raiz ate a folha correspondente. Assim obtemos a tabela de códigos resultante da árvore de Huffman (da Figura 5):

Tabela 2 – Caminho na árvore de cada carácter

A	*B*	*C*	*D*	*E*	*R*
10	011	110	111	00	010

Descodificação:

Para descodificar uma mensagem obtida através da tabela acima, por exemplo:

10011010101101011111001101010,

basta utilizar cada bit da mensagem para percorrer a árvore de Huffman desde a raiz ate a folha ate obter o símbolo descodificado e assim sucessivamente. A sucessão de bits acima codifica o texto correspondente a ABRACADABRA. Pode-se observar que o texto foi compactado de 88 bits para 28 bits.

Estruturas de dados:

As estruturas de dados utilizadas foram:

- Na determinação das frequências de caracteres foi utilizada uma *hashtable* para que o acesso e inserção de novos dados seja feito em O(1).
- Para a criação da arvore foi criada uma *LinkedList* com nós que contêm o valor e frequência para que sejam feitas inserções e remoções em O(n).
- Para guardar a codificação foi utilizada uma *Hashtable*.

2.2 Estrutura de classes

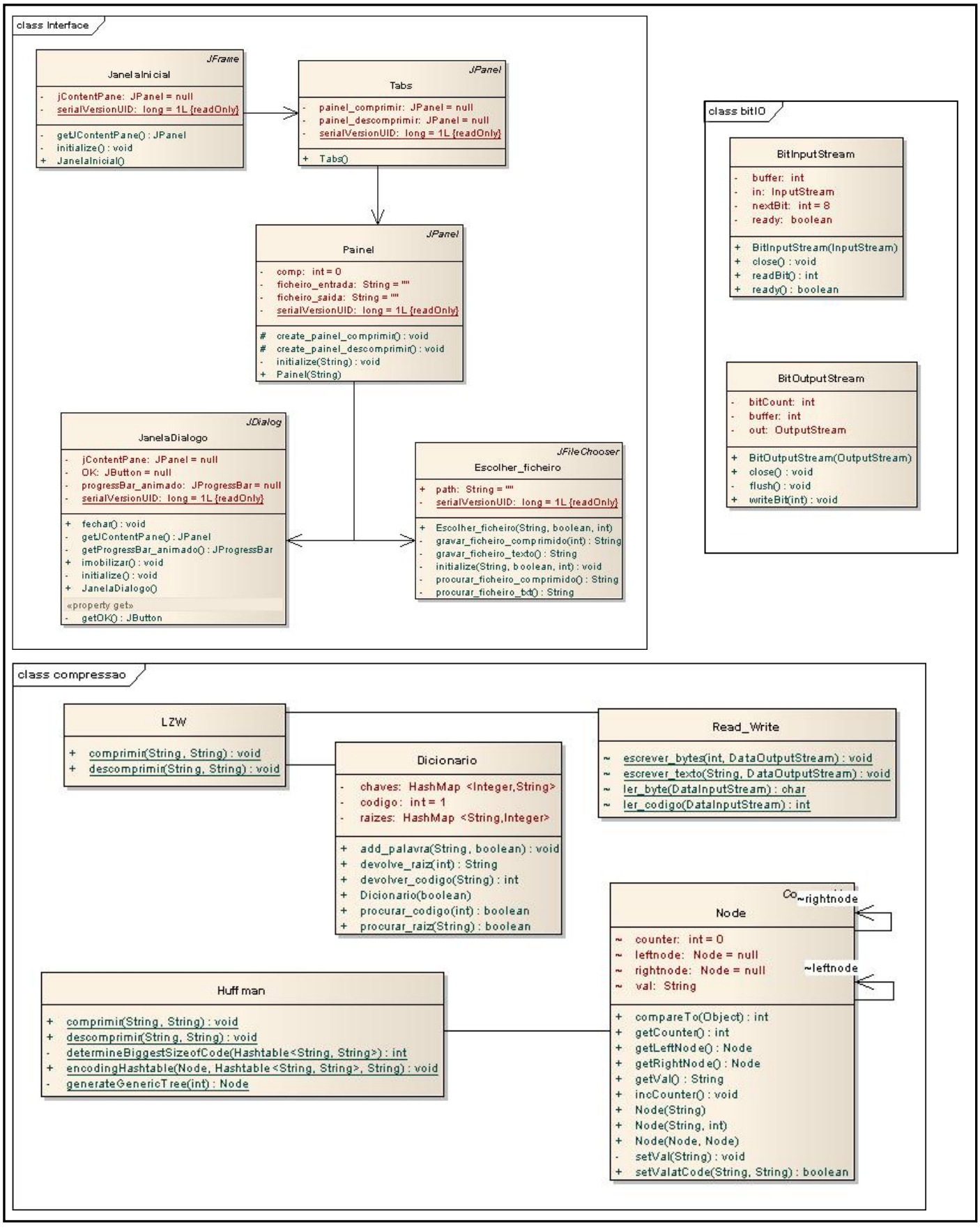

Figura 6 – Diagrama UML

- **LZW**: classe responsável pela implementação do algoritmo de LZW de forma a comprimir e descomprimir um ficheiro, através de dois métodos. Para cumprir esse objectivo é necessário a utilização da classe *Read_Write.*
- **Read_Write**: esta classe apresenta métodos necessários para ler e escrever bytes em ficheiros. A codificação realizada pelo algoritmo LZW é de 16 bits por esse motivo quando acede a ficheiros codificados lê/escreve 16bits.
- **Dicionario**: classe que apresenta vários métodos para manipular o dicionário necessário para implementar a compressão LZW. De modo a tornar mais eficiente os acessos ao dicionário foram implementadas duas *HashMap's* de modo a uma delas ter como *key* a palavra código e outra as *Strings* que são relacionadas com as palavras código.
- **Huffman**: esta classe contem dois métodos estáticos em que, um comprime com o algoritmo de Huffman e outro descomprime também com o algoritmo de Huffman.
- **BitOutputStream**: esta classe é utilizada para escrever bit a bit num ficheiro. Esta escrita é feita através de um buffer intermédio em que se vai escrevendo até 8 bits. Quando estes 8 bits são atingidos o byte é escrito no ficheiro e o buffer é apagado.
- **BitInputStream**: esta classe lê ficheiros bit a bit. Para o efectuar é utilizado um buffer de 8 bits que vai sendo lido bit a bit. Quando são lidos 8 bits o buffer é preenchido com um novo byte do ficheiro.
- **Node**: esta classe corresponde aos nós da árvore de codificação, sendo que cada nó contém um valor, um nó esquerdo, um nó direito e um contador.
- **JanelaInicial**: classe que tem como funcionalidade inicializar a janela principal do programa.
- **Tabs**: classe referente a interface gráfica que cria o JTabbedPane e inicializa com os parâmetros correspondentes.
- **Painel**: classe que esta encarregue de construir o painel (JPanel) referente a compressão e a descompressão do ficheiro com todos os seus componentes.
- **JanelaDialogo**: classe responsável pela janela de diálogo quando se realiza uma compressão ou descompressão de um ficheiro, contem a animação da barra de progresso.
- **Escolher_ficheiro**: classe que apresenta um construtor com vários parâmetros para apresentação de um janela de dialogo de abrir/guardar um ficheiro, ou seja um JFileChooser.

As principais bibliotecas que foram necessárias para o desenvolvimento do trabalho foram:

- java.io.DataInputStream;
- java.io.DataOutputStream;
- java.io.FileInputStream;
- java.io.FileOutputStream;
- java.io.IOException;
- java.io.BufferedReader;
- java.io.BufferedWriter;
- java.io.File;
- java.io.FileReader;
- java.io.FileWriter;
- java.io.IOException;
- java.io.Reader;
- java.util.Enumeration;
- java.util.Hashtable;

- java.util.LinkedList;
- java.util.ListIterator;
- java.util.HashMap;
- javax.swing.JFileChooser;
- javax.swing.JPanel;
- javax.swing.JDialog;
- javax.swing.JProgressBar;
- java.awt.Rectangle;
- javax.swing.JButton;
- java.awt.event.WindowAdapter;
- java.awt.event.WindowEvent;
- javax.swing.JPanel;
- javax.swing.JFrame;
- javax.swing.SwingWorker;
- javax.swing.JRadioButton;
- javax.swing.JTextField;
- javax.swing.JTabbedPane.

Excepcoes:

- IOException;

2.3 Interface gráfica

A interface gráfica foi pensada para ser o mais intuitiva possível, possibilitando assim uma utilização confortável pelo utilizador da mesma.

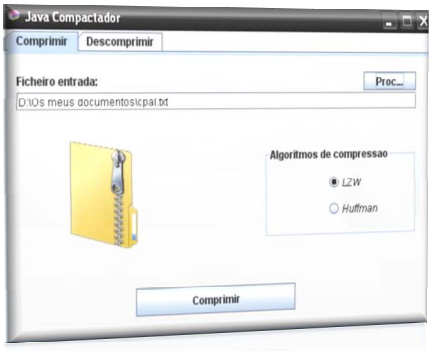

Figura 7 – Janela Inicial, visualizam-se as JTabbedPane

A parte gráfica do programa baseia-se numa frame inicial que possibilita ao utilizador comprimir ou descomprimir um ficheiro, para tal função foi implementada uma JTabbedPane – Figura 7.

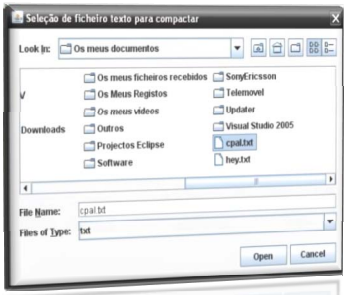

Figura 8 – JFileChooser

Na mesma JFrame existe a possibilidade de procurar um ficheiro através da utilização do componente JFileChooser – Figura 8 e de escolha do tipo de compressão se se verificar - JRaddionButton agrupadas num JButtonGroup – Figura 7. Depois de introduzir os parâmetros necessários aquando da compressão/descompressão o utilizador visualiza uma janela de diálogo com a informação que o programa se encontra em processamento através de uma barra de progresso - Figura 9.

Figura 9 – Janela de Dialogo com barra de progresso

Os principais eventos tratados na interface gráfica são referentes aos botões presentes. Cada um deles com uma função distinguível. O botão “procurar” tem como objectivo chamar a classe Escolher_ficheiro. Os botões comprimir/descomprimir chamam a classe referida anteriormente e visualizam a janela de diálogo com a barra de progresso enquanto o ficheiro se encontra em processamento. No fim do processamento o JButton designado “OK” fica activo possibilitando que a janela de diálogo seja fechada.

3 Utilização

O programa implementado suporta compressão e descompressão de ficheiros de texto (e outros – extensão ao enunciado [somente o algoritmo de LZW]) usando o algoritmo de LZW (Lempel-Ziv-Welch) ou os códigos de Huffman.

Figura 10 – Exemplo de ficheiro codificado

Os ficheiros de entrada do programa serão os ficheiros que se deseja comprimir e os ficheiros comprimidos que se deseja descomprimir. Os ficheiros comprimidos tem extensão *.lzw e *.huf (*.hufc árvore de codificação de Huffman), respectivamente comprimidos pelo método de LZW e pelos códigos de Huffman. Os ficheiros comprimidos encontram-se devidamente codificados consoante o método adoptado – Figura 10.

3.1 Utilização do programa (Figura 11 e 12)

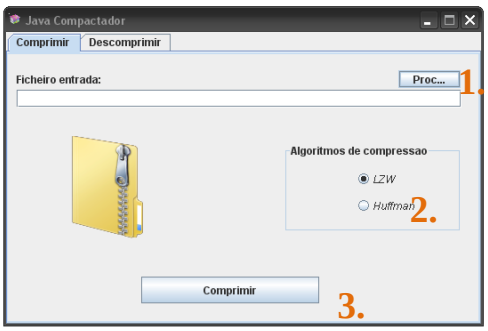

Figura 11 – Janela Principal

1. Clicar no Botão “Procurar” e escolher o ficheiro que se deseja comprimir/descomprimir através do JFileChooser;
2. Seleccionar o tipo de compressão a usar;
3. Clicar no botão “Comprimir”/”Descomprimir” e escolher a directoria do ficheiro a criar e o nome;
4. Esperar que o programa termine o processamento.

Figura 12 – Janela com barra de progresso

Foi exemplificado para comprimir para descomprimir o processo é o mesmo a excepção que não e necessário seleccionar o método de compressão usado.

De modo a confrontar os dois algoritmos utilizados para elaboração deste trabalho prático recorreu-se a compactação e descompactarão de 3 ficheiros de texto de tamanho diferentes, pode-se observar os resultados nos gráficos que se seguem.

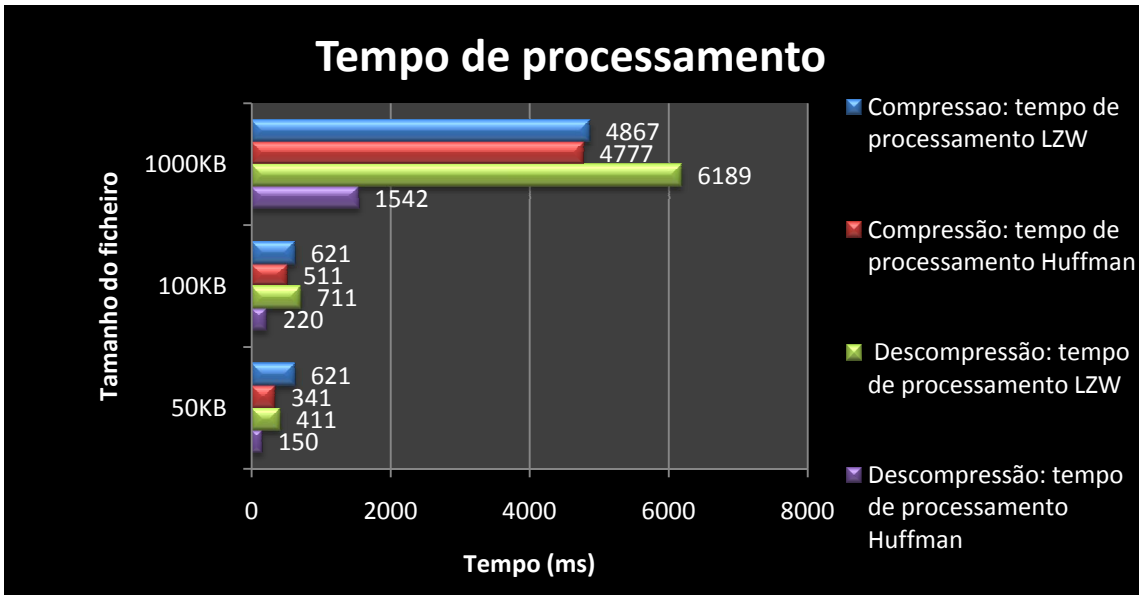

Figura 13 – Gráfico da relação do tempo de processamento de cada algoritmo

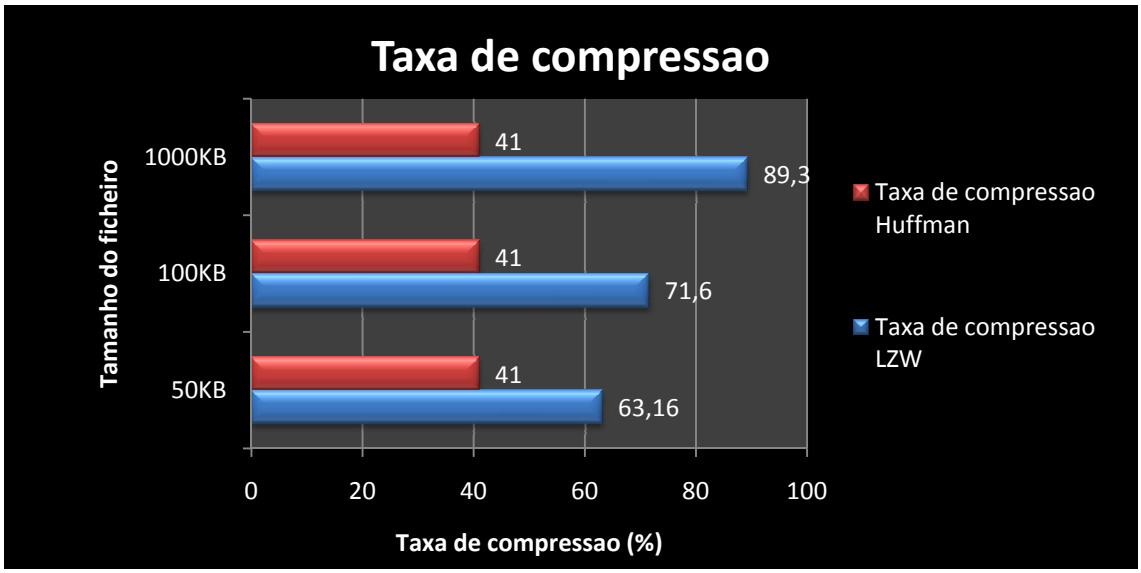

Figura 14 – Gráfico da relação da taxa de processamento de cada algoritmo

Como e possível observar nos gráficos, os dois algoritmos apresentam vantagens e desvantagens entre si. Verificamos que a taxa de compressão do algoritmo LZW é muito superior aos códigos de Huffman, que apresenta uma compressão constante de 41%, enquanto que o algoritmo LZW apresenta uma média de taxa de compressão de 74%. Quanto aos tempos de processamento os códigos de Huffman levam uma enorme vantagem pois a descodificação é muito mais rápida do que a do algoritmo adversário.

Pelos motivos mencionados, se se pretende maior rapidez e menor compressão dever-se-á utilizar códigos de Huffman, caso a taxa de compressão seja um factor importante o algoritmo LZW é mais eficaz.

Os testes efectuados foram realizados num computador com as seguintes características: processador Intel PentiumM 1.3Ghz, 512Mb de memória RAM.

4 Conclusões

O programa implementado suporta todas as funcionalidades pré-estabelecidas no enunciado. No enunciado o programa é específico para compressão de ficheiros de texto, contudo o algoritmo LZW desenvolvido pode ser utilizados para qualquer tipo de ficheiro, visto o mesmo utilizar a informação contida nos bits do ficheiro e não em caracteres ASCII. Contudo foi pensado para ficheiros de texto.

O código fonte a apresenta um tamanho de 1930 linhas.

4.1 Trabalho realizado por cada elemento do grupo

A solução para a resolução do problema proposto pela equipa de docentes da cadeira de Complementos de Programação e Algoritmos foi estruturada por ambos os elementos.

Tempo despendido por cada elemento do grupo:

- Paulo Pinto: 50horas;
- Rui Campos: 50horas.

5 Referências

- [1]Thinking in Java, 4th edition, Bruce Eckel, Prentice Hall, 2005
- [2]LZW – Wikipédia, a enciclopédia livre
  http://pt.wikipedia.org/wiki/LZW
- [3]Codificação de Huffman – Wikipédia, a enciclopédia livre
  http://pt.wikipedia.org/wiki/Codifica%C3%A7%C3%A3o_de_Huffman
- [4] HashMap(Java 2 Platform SE v1.4.2)
  http://java.sun.com/j2se/1.4.2/docs/api/java/util/HashMap.html
- [5] Improve Application Performance With SwingWorker in Java SE 6
  http://java.sun.com/developer/technicalArticles/javase/swingworker