# Projeto 1

Aluno: Eduardo Lemos Rocha Matrícula: 170009157

## Instruções

O projeto foi feito na linguagem Python. A seguir, as bibliotecas necessárias para a execução:

- *math*
- *heapq*
- *bitstring*

Para executar o programa, basta realizar o seguinte comando no diretório do projeto:

`python \.main.py` (Windows)

`python main.py` (Linux)

## Descrição do Programa

### Arquivo de Entrada

No início do programa, é capturado o nome do arquivo (**com extensão**) por meio da função **getFilename**. O arquivo é aberto e lido byte a byte e salvo em uma lista com o auxílio da função **readBinaryFile**.

### Probabilidades

Após a leitura do arquivo, calcula-se um dicionário de probabilidades (função **calculateProbability**). Como será necessário extrair os menores valores de probabilidade durante a execução do algoritmo, utilizou-se a estrutura **heapq**. Dessa forma, ao realizar as operações **pop** e **push** a estrutura já reordenará os elementos em função das probabilidades.

### Entropia

O passo seguinte é o cálculo da entropia (função **calculateEntropy**), utilizando-se das probabilidades para cada símbolo que foram calculadas anteriormente.

### Código de Huffman

Um dos passos principais do projeto é a correspondência dos símbolos do arquivo de entrada para com seus *bitstreams*, seguindo o algoritmo apresentado em sala de aula. A função **encode** é responsável por essa tarefa.

Inicialmente, utiliza-se do algoritmo explicado em sala: unir os dois símbolos com menor probabilidade, criando-se um novo símbolo. Esse procedimento se repete até que restem apenas dois símbolos. Determina-se um dos restantes como o bit 1 e o outro como bit 0. Finalmente, é determinado o valor de todos os símbolos criados durante a primeira etapa de maneira recursiva.

Ao final, o dicionário de Huffman, isto é, o dicionário que relaciona cada símbolo com seu respectivo *bitstream*, irá conter também os símbolos inexistentes no arquivo, criados apenas para a execução do algoritmo citado no parágrafo anterior. Dessa forma, realiza-se um filtro, de forma a eliminá-los do dicionário final da função **encode**.

### Comprimento Médio

A função **calculateAverageLength** calcula o comprimento médio do código gerado a partir do dicionário de Huffman criado.

### Compressão do Arquivo

Neste passo, realiza-se a compressão do arquivo de entrada, usufruindo-se do dicionário de Huffman gerado. É escrito um arquivo de saída em bytes a partir do *bitstream* gerado pela tradução.

A organização escolhida para o arquivo compactado foi:

$$árvoreHuffman + + bitsArtificiais + + arquivoTraduzido$$

A árvore de Huffman é organizada com o primeiro byte representando a quantidade de símbolos, seguido da relação entre símbolo e *bitstream*. Os bits artificiais indicam para o *decoder* quantos bits no final do arquivo foram adicionados para completar o último byte do arquivo. Isso é necessário, visto que não necessariamente o arquivo traduzido irá compor um conjunto de bits múltiplo de 8. Finalmente, insere-se o arquivo traduzido no arquivo compactado.

Como observação, ressalta-se que o nome do arquivo compactado é sempre o nome do arquivo original sem extensão, com a adição de ".du" no final do nome.

### Descompressão do Arquivo

O penúltimo passo consiste em descompactar o arquivo por meio da função **decode**. O primeiro passo consiste em recuperar o dicionário de Huffman que está contido no começo do arquivo compactado. Após isso, captura-se o byte que informa quantos bits foram artificialmente inseridos para formar o último byte. Finalmente, o restante do arquivo é lido bit a bit.

O processo acumula em um *buffer* os bits lidos. Ao encontrar a sequência de bits no dicionário de Huffman, realiza-se a tradução para o símbolo correspondente e limpa-se o *buffer*. Até que restem apenas os bits artificiais, esse procedimento é feito.

### Estatísticas

Nesta última etapa, é calculado e mostrado na tela o tamanho do arquivo original, do arquivo compactado e do arquivo descompactado (averiguar perdas durante após o processo). A função **calculateSize** é utilizada para calcular ambos os tamanhos de interesse. A seguir, uma lista que relaciona uma estatística e uma função responsável por mostrá-la para o usuário:

- **showEntropy**: mostrar a entropia calculada;
- **showAverageLength**: mostrar o comprimento médio do código gerado;
- **showFileSize**: mostrar o comprimento de um arquivo. Essa função também é capaz de mostrar o tamanho de um arquivo com e sem um *overhead* fornecido, na unidade bits;
- **showCompression**: mostrar a compressão (%) a partir de dois arquivos fornecidos.

## Resultados

Na tabela a seguir, consta os tamanhos em bytes dos arquivos testes fornecidos pelo professor:

| Arquivo | Tamanho Original (Bytes) | Tamanho Compressão (Bytes) | Tamanho Compressão Completo (Bytes) | Resultado Comercial 7Zip (Bytes) |
|---|---|---|---|---|
| dom_casmurro.txt | 389670 | 227985 | 229271 | 147281 |
| fonte.txt | 1000000 | 368694 | 368753 | 410398 |
| fonte0.txt | 1000 | 210 | 235 | 445 |
| fonte1.txt | 1000000 | 415794 | 415943 | 468252 |
| lena.bmp | 263290 | 247811 | 250743 | 216526 |
| TEncEntropy.txt | 19415 | 12883 | 13739 | 4307 |
| TEncSearch.txt | 253012 | 164197 | 165175 | 37668 |

A tabela abaixo mostra uma comparação entre os índices de compressão do projeto (*overhead* incluso) com os índices do compressor comercial 7Zip:

| Arquivo | Compressão (%) | Compressão Comercial 7Zip (%) |
|---|---|---|
| dom_casmurro.txt | 41.16 | 62.20 |
| fonte.txt | 62.12 | 58.96 |
| fonte0.txt | 76.50 | 55.50 |
| fonte1.txt | 58.41 | 53.17 |
| lena.bmp | 4.77 | 17.76 |
| TEncEntropy.txt | 29.24 | 77.81 |
| TEncSearch.txt | 34.72 | 85.11 |

Os resultados apresentados na coluna "Tamanho Compressão" corresponde ao tamanho do arquivo comprimido sem considerar a árvore de huffman inclusa (*overhead*). A coluna "Tamanho Compressão Completo" representa o tamanho em bytes levando-se em consideração a árvore.

## Conclusão

Nota-se que o programa apresentou baixa performance em comprimir arquivos compostos por símbolos dependentes entre si, como nos arquivos de código (TEncEntropy.txt e TEncSearch.txt) e de imagem (lena.bmp). Em contrapartida, o compressor comercial contornou essas situações.

Os arquivos compostos por símbolos mais independentes entre si (fonte.txt, fonte0.txt e fonte1.txt), apresentaram performances com um cenário invertido. O compressor comercial, por ser mais otimizado para arquivos com símbolos dependentes, apresentou um pior resultado quando comparado a implementação descrita.

Como melhorias futuras, destaca-se a análise do críterio de desempate para símbolos de mesma probabilidade durante a produção do código de Huffman. O algoritmo utilizado baseia-se na criação de símbolos fictícios, somando-se as probabilidades em um único símbolo. Alterar a ordenação de símbolos (reais e fictícios) de mesma probabilidade pode deixar o código final mais uniforme, potencialmente diminuindo a quantidade de bits dos símbolos úteis, isto é, que de fato são utilizados pelo arquivo que será comprimido.