L-EFLI NST-PROG 07 -Técnico de Programação

UFCD 10794 – Programação Avançada em Python

Relatório Realizado por:

Catarina Silva nº 7061800

Centro de Formação de Alcântara, 3/03/2022

Índice

Introdução

Este trabalho teve como objetivo consolidar os conhecimentos de programação avançada em Python, visando o desenvolvimento de um programa de compressão e descompressão como trabalho final da UFCD 10794, o PZYP, baseando-se no algoritmo LZSS.

No seguimento deste projeto, a finalidade passa por desenvolver um programa/aplicação que comprima e descomprima ficheiros, fazendo uso das bibliotecas necessárias para o seu desenvolvimento e terminar com a criação de uma interface gráfica da aplicação usando o QtDesigner.

De modo a trabalhar sobre este algoritmo e na sua implementação, foi feita uma pesquisa em torno do algoritmo em si e no que consistia o LZSS, assim como o algoritmo de compressão mais conhecido, o LZ77.

Lempel-Ziv-Storer-Szymanski, que chamamos de LZSS, é uma variação simples do algoritmo LZ77 comum. Usa o mesmo conceito de token com uma distância e comprimento para informar ao descompressor onde copiar o texto, exceto que só é colocado o token quando o token é menor que o texto que está a substituir.

A finalidade é que nunca aumente o tamanho de um arquivo adicionando tokens em todos os lugares para letras repetidas. Pode-se imaginar que o LZ77 aumentaria facilmente o tamanho do arquivo se simplesmente comprimisse cada letra repetida “e” ou “i” como um token, o que pode levar pelo menos 5 bytes dependendo do arquivo e implementação em vez de apenas 1 para LZSS.

Para isso, neste projeto foi estabelecido a implementação da compressão do algoritmo através de iterações nos caracteres, buffers de pesquisa (a nossa janela), verificação desses buffers de pesquisa para aceitar mais caracteres, comparação do tamanho dos tokens, a limitação do tamanho máximo da nossa janela. Por fim para se implementar a descompressão, recorremos a uma identificação desses tokens e tradução dos mesmos.

A elaboração deste projeto foi finalizado com a criação de uma interface gráfica da aplicação, usando a framework PyQt, onde através de um menu interativa podemos selecionar uma opção para escolher o documento ou ficheiro (Select Files), temos outra opção para comprimir (Encode), outra para descomprimir (Decode) e por fim uma opção para sair da aplicação (Exit).

Desenho e Estrutura

Como referido acima, foi pedido a realização de um programa de compressão e descompressão como trabalho final da UFCD 10794, o PZYP, baseando-se no algoritmo LZSS, para uma melhor compressão do projeto foram realizados dois Fluxogramas (compressão e descompressão), criados através do website *draw.io.*

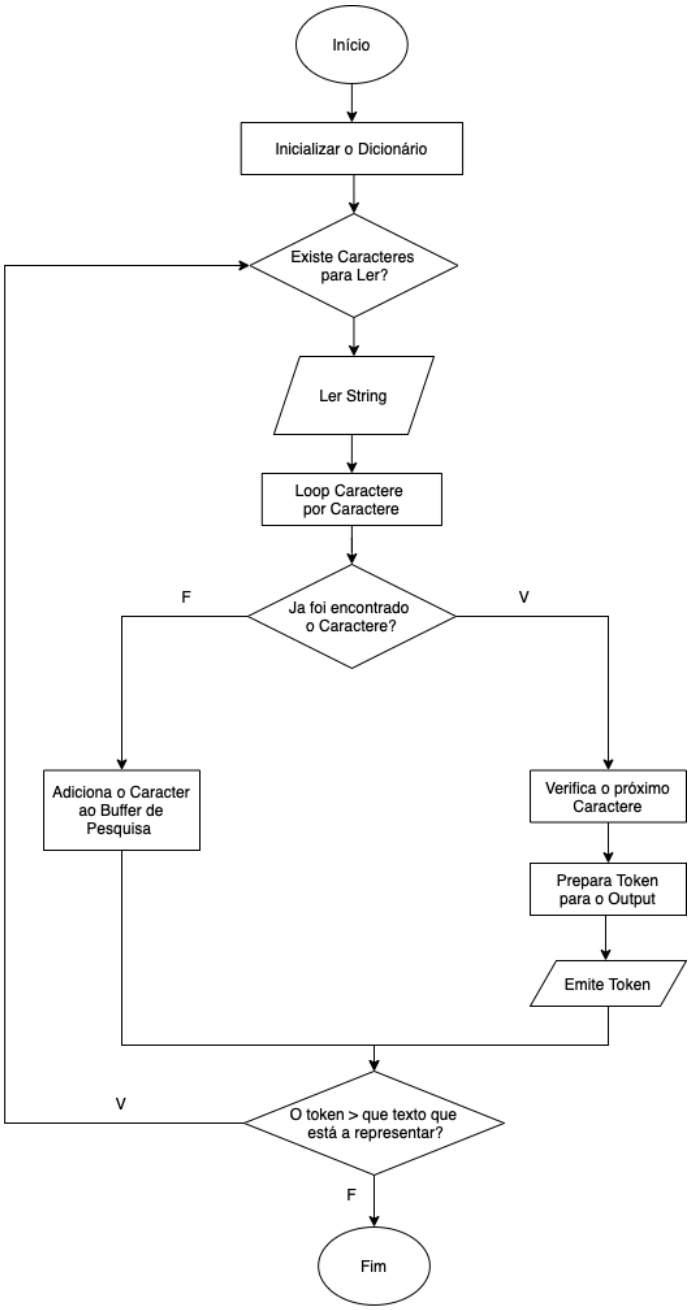


Figura 1 - Fluxograma de Compressão

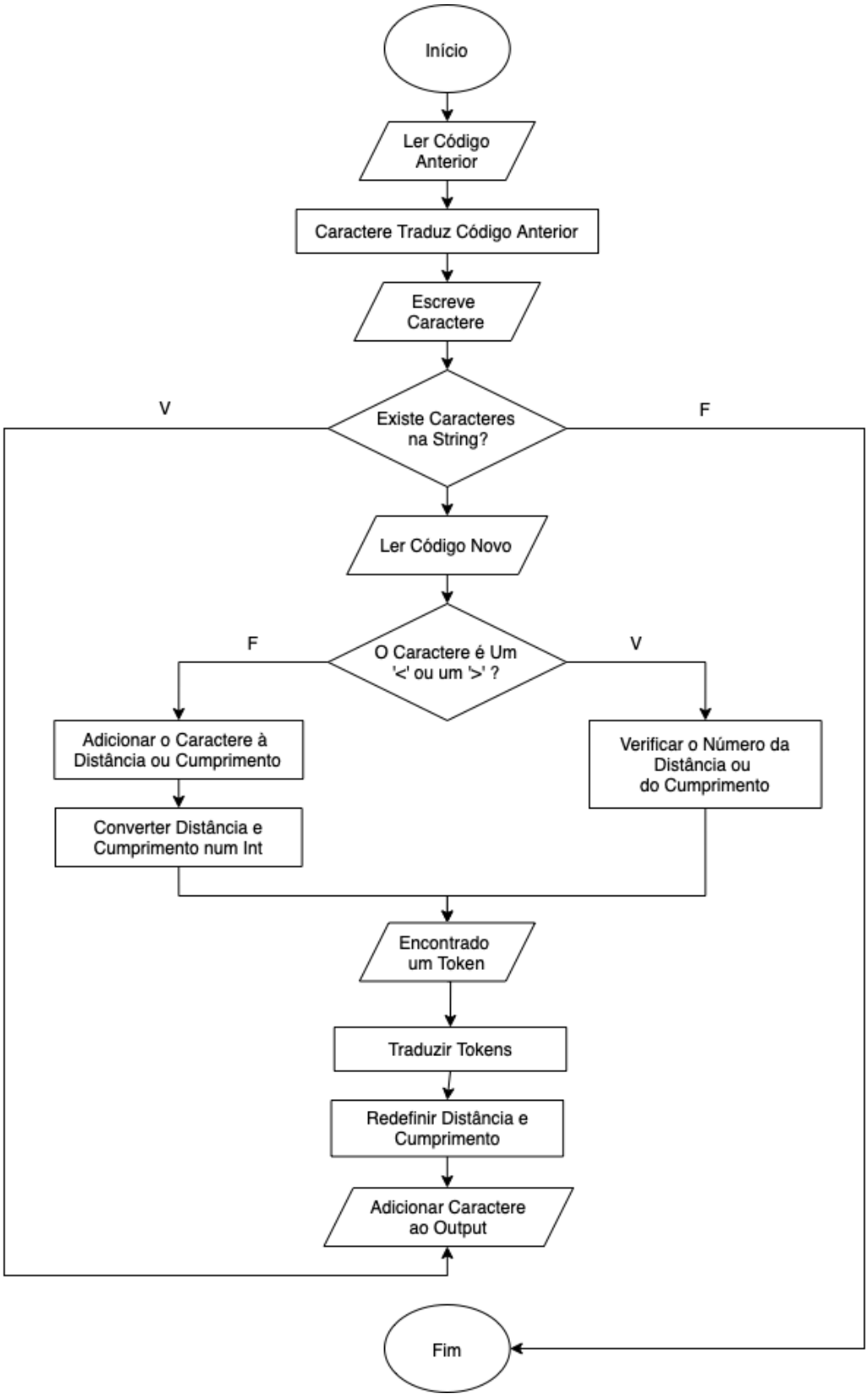


Figura 2 - Fluxograma de Descompressão

Implementação

De acordo com o exigido no enunciado, foi desenvolvida uma aplicação para a compressão e descompressão de ficheiros, utilizando como base o algoritmo LZSS. A solução implementada para o respetivo projeto foi desenvolvida no Visual Studio Code, com a versão 3.8.2.

Para inicializar o programa foi implementado um loop de caracteres, para que percorra cada caractere para comprimir. Um modelo bastante utilizado neste tipo de algoritmos de compressão.

No loop de caracteres, foi preciso comprimir caractere a caractere e depois fazer um loop sobre a compressão. Ou seja, ele converte a string numa matriz de bytes e desta forma será possível vermos o caractere quando o imprimimos.

```
for char in text_bytes:
    # O index onde os caracteres aparecem na nossa janela/buffer de pesquisa
    index = elements_in_array(check_characters, search_buffer)

    if elements_in_array(check_characters + [char], search_buffer) == -1 or i == len(text_bytes) - 1:
        if i == len(text_bytes) - 1 and elements_in_array(check_characters + [char], search_buffer) != -1:
            # Se for o ultimo caractere, adiciona o proximo caractere ao texto que o token representa
            check_characters.append(char)

        if len(check_characters) > 1:
            index = elements_in_array(check_characters, search_buffer)
            # Calcular a distância relativa
            offset = i - index - len(check_characters)
            # Definir o comprimento do token (Por quantos caracteres o representa)
            length = len(check_characters)

            token = f"<{offset},{length}>"  # Construir o nosso token

            if len(token) > length:
                # Comprimento do token é maior que o comprimento que o representa, por isso imprime os caracteres
                output.extend(check_characters)  # Imprime os caracteres
            else:
                # Imprime o nosso token
                output.extend(token.encode(encoding))

            # Adiciona os caracteres ao nosso buffer de pesquisa
            search_buffer.extend(check_characters)
        else:
            output.extend(check_characters)  # Imprime o caractere
            # Adiciona os caracteres ao nosso buffer de pesquisa
            search_buffer.extend(check_characters)

        check_characters = []

    check_characters.append(char)

    # Verifica se o search buffer está a exceder o tamanho maximo da janela
    if len(search_buffer) > max_sliding_window_size:
        # Remove o primeiro elemento do buffer de pesquisa
        search_buffer = search_buffer[1:]

    i += 1

return bytes(output)
```

De seguida, o que foi feito, foi criar a janela/buffer de pesquisa, implementando a "memória" para que o programa possa verificar se já viu o caractere em questão. Foi usado o método append, de forma a conseguir adicionar cada caractere na nossa janela de pesquisa.

Após criarmos este buffer de pesquisa, foi preciso verificá-lo e para isso, é preciso olhar para trás à procura de caracteres que já foram vistos. Isso pode ser conseguido, usando o método list.index. A partir daqui, foi criado um token e retornamo-lo assim que é encontrado um caractere. No entanto, esta implementação do token gerou um problema, que tem a ver com o facto de que se tivermos uma palavra que se repete duas vezes, ele copiará cada caractere em vez da palavra inteira. Então, para corrigir esta questão, verificou-se no buffer de pesquisa se há mais do que um caractere.

```
def elements_in_array(check_elements, elements):
    i = 0
    offset = 0
    for element in elements:
        if len(check_elements) <= offset:

            # Todos os elementos no check_elements estão nos elements
            return i - len(check_elements)

        if check_elements[offset] == element:
            offset += 1
        else:
            offset = 0

        i += 1
    return -1


encoding = "utf-8"

# compressor


def encode(text, max_sliding_window_size=4096):
    text_bytes = text.encode(encoding)

    search_buffer = []  # Array de numeros inteiros, representando bytes
    check_characters = []  # Array de numeros inteiros, representando bytes
    output = []  # Saída do array

    i = 0
    for char in text_bytes:
        # O index onde os caracteres aparecem na nossa janela/buffer de pesquisa
        index = elements_in_array(check_characters, search_buffer)
```

Depois de ter sido verificado no buffer de pesquisa a existência de mais do que um caractere, comparou-se os tamanhos do token, uma parte crucial neste processo, pois aqui estamos a dizer que, se o token ocupar mais espaço do que o texto que está a representar, então não vai imprimir um token, apenas imprimirá o texto.

```
        if len(token) > length:
            # Comprimento do token é maior que o comprimento que o representa, por isso imprime os caracteres
            output.extend(check_characters)  # Imprime os caracteres
        else:
            # Imprime o nosso token
            output.extend(token.encode(encoding))

        # Adiciona os caracteres ao nosso buffer de pesquisa
        search_buffer.extend(check_characters)
    else:
        output.extend(check_characters)  # Imprime o caractere
        # Adiciona os caracteres ao nosso buffer de pesquisa
        search_buffer.extend(check_characters)
```

Por fim para se ficar com o algoritmo de compressão a funcionar, foi preciso resolver a questão da janela deslizante. O nosso buffer de pesquisa, fica grande se tentarmos descomprimir um arquivo grande. Digamos que se está a descomprimir um arquivo de 1 Gb. Depois de se examinar cada caractere, adiciona-se ao buffer de pesquisa e continua-se, apesar de que a cada iteração temos que pesquisar também em todo o buffer de pesquisa por determinados caracteres.
Isso aumenta rapidamente para arquivos maiores. Neste cenário em que se tem um arquivo de 1 Gb, perto do final tem que se pesquisar quase 1 bilhão de bytes para codificar um único caractere.

Visto que isto se torna pouco eficaz, teve que se fazer uma troca. Com cada algoritmo de compressão tem que se decidir entre velocidade e taxa de compressão. Neste caso, ter criado a janela deslizante ajudou neste processo.

O que a “janela deslizante” faz é limitar o tamanho máximo do buffer de pesquisa. Quando se adiciona um caractere ao buffer de pesquisa que o torna maior que o tamanho máximo da janela deslizante, este remove automaticamente o primeiro caractere. Dessa forma, a janela está “deslizando” à medida que se percorre o arquivo, e o algoritmo não diminui assim a velocidade.

Daqui, partiu-se para a implementação do descompressor e chegou-se à conclusão que tudo o que o descompressor precisa de fazer é converter um token (“<5,2>”) no texto literal que ele representa. O descompressor não se importa com buffers de pesquisa, janelas deslizantes ou comprimentos de token, ele tem apenas um trabalho.

Para começar, descomprimiu-se caractere por caractere exatamente como no compressor, então começou-se com o loop principal dentro de uma função. Também foi preciso comprimir e descomprimir as strings para se manter o código 'encoding = "utf-8"'. O próximo passo foi identificar os tokens e começar a fazer a descompressão. O objetivo deste descompressor é converter um token em texto, então é preciso identificar primeiro um token e extrair a distância e comprimento antes de poder convertê-lo em texto.

Como se foi iterando caractere por caractere, pode-se verificar se o caractere era um caractere de abertura ou de fecho de token para saber se estávamos dentro de um token. Dentro do loop, verificou-se se o caractere era um <,, ou um > e modificou-se as variáveis de acordo para rastrear onde se estava. Se o caractere não fosse nenhum desses e estivéssemos dentro de um token, nesse caso iria se querer adicionar o caractere à distância ou ao comprimento porque isso significa que o caractere é uma distância ou comprimento.

Por fim, se o caractere for um >, isso significa que se está a sair do token, então vai se converter o comprimento e a distância num número inteiro. Tem que se fazer isso porque eles são representados atualmente como uma lista de bytes, então é preciso converter esses bytes numa string e converter essa string num número inteiro. Então, finalmente, imprime-se que foi encontrado um token. O último passo para a implementação deste descompressor foi fazer a tradução dos tokens para o texto que eles representam. Para calcular o pedaço de texto que um token está a referenciar, usou-se a distância e o comprimento para encontrar o texto da saída atual.

```
    for char in text_bytes:
        if char == "<".encode(encoding)[0]:
            inside_token = True  # Aqui estamos dentro de um token
            scanning_offset = True  # Aqui estamos a procurar pelo numero do comprimento
        elif char == ",".encode(encoding)[0] and inside_token:
            scanning_offset = False
        elif char == ">".encode(encoding)[0]:
            inside_token = False  # Já não estamos dentro do token

            # Converter comprimentos e distancias para um numero inteiro
            length_num = int(bytes(length).decode(encoding))
            offset_num = int(bytes(offset).decode(encoding))

            # Recebe o texto que o token representa
            referenced_text = output[-offset_num:][:length_num]

            # referenced_text é uma lista de bytes para que nós usemos o 'extend' para adicionar cada byte na saída
            output.extend(referenced_text)

            # Reset comprimento e distância
            length, offset = [], []
        elif inside_token:
            if scanning_offset:
                offset.append(char)
            else:
                length.append(char)
        else:
            output.append(char)  # Adiciona o caractere ao nosso output

    return bytes(output)
```

Com base naquilo que foi transmitido, para a otimização deste programa, sugeriu-se que fosse introduzido as respetivas bibliotecas: docopt, Union, BinaryIO, Tuple, math, bitarray e bitstruct.

O docopt é um módulo de descrição de interface de linha de comando, que auxilia os utilizadores a definir uma interface em linhas de comando e gera o seu próprio analisador.

```
'''
Usage:
    pzip [-c [-l LEVEL] | -d | -h] [-s] [-p PASSWORD] FILE
Operation:
    -c, --encode, --compress            Compress FILE whith PZYP
    -d, --decode, --decompress          Decompress FILE compressed with PZYP
Options:
    -l, --level                         Compression level [default: 2]
    -s, --summary                       Resume of compressed file
    -h, --help                          Shows this help message and exits.
    -p PASSWORD, --password=PASSWORD    An optional password to encrypt the file (compress only)
    FILE                                The path to the file to compress / decompress
'''
# imports
from docopt import docopt
from typing import Union, BinaryIO, Tuple
import math
import lzss_io
import bitstruct
from bitarray import bitarray
import pzypapp as mw
```

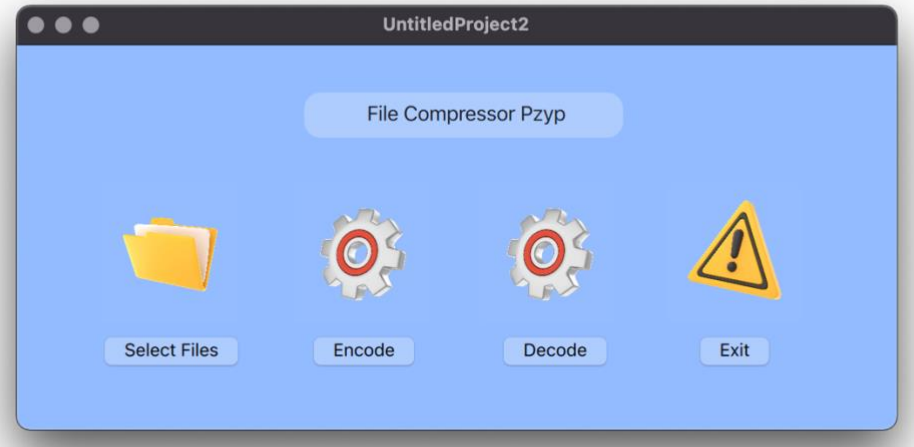
O docopt é uma das formas mais utilizadas para exibir mensagens de ajuda como por exemplo (a opção -h ou -help), permitindo assim iniciar diversas funcionalidades através da linha de comando como podemos visualizar no seguinte exemplo, onde mostra a finalização do programa.

```
def main():
    args = docopt(__doc__, version='0.1')
    file = args['FILE']

    if args['--compress']:
        encode(file, args["--encode"], args["--summary"])
    if args['--decompress']:
        decode(file, args["--decode"], args["--summary"])


if __name__ == '__main__':
    main()
```

Com a finalização do projeto foi criado a interface gráfica do projeto, utilizando a framework PySide6 e a ferramenta QtCreator, assim como a aplicação QtDesigner Studio.

```
from PySide6.QtCore import (QCoreApplication, QDate, QDateTime, QLocale,
    QMetaObject, QObject, QPoint, QRect,
    QSize, QTime, QUrl, Qt)
from PySide6.QtGui import (QBrush, QColor, QConicalGradient, QCursor,
    QFont, QFontDatabase, QGradient, QIcon,
    QImage, QKeySequence, QLinearGradient, QPainter,
    QPalette, QPixmap, QRadialGradient, QTransform)
from PySide6.QtWidgets import (QApplication, QGraphicsView, QMainWindow, QPlainTextEdit,
    QPushButton, QSizePolicy, QStatusBar, QWidget)

class Ui_MainWindow(object):
    def setupUi(self, MainWindow):
        if not MainWindow.objectName():
            MainWindow.setObjectName(u"MainWindow")
        MainWindow.resize(670, 460)
```

Por fim, depois da criação da interface gráfica foi possível visualizar o protótipo da aplicação na ferramenta QtDesigner Studio, onde é possível observar os diversos comandos que permitem ao usuário selecionar ficheiros, comprimir ficheiros, descomprimir ficheiros e sair do menu.



Conclusão

Na realização deste trabalho foi encontrado algumas dificuldades, em primeiro lugar na realização do programa de compressão e descompressão, pois a compreensão do algoritmo tornou-se difícil face à complexidade do mesmo, mas depois de compreendido o desafio foi representá-lo em código, onde foi dispensado mais tempo deste projeto.

Entre a criação do programa de compressão e descompressão, a integração de bibliotecas como docopt para ler as opções da linha de comandos, bitarray e bitconstruct foi algo que levou a um processo demorado e trabalhoso.

Depois de superadas as dificuldades sentidas em todo este processo, avançou-se para o desenvolvimento da interface gráfica do projeto, utilizando a framework PySide6 e a ferramenta QtCreator, assim como a aplicação QtDesigner Studio.

Toda a parte de desenvolvimento criativa deste projeto, pode-se concluir que foi a parte mais interessante e desafiante, pois a integração gráfica com a interação produzida através da linguagem de Python foi algo inovador e nunca antes realizado ao longo deste curso. As ferramentas utilizadas permitiram a criação de uma aplicação funcional de modo a que o utilizador perceba rapidamente a eficiência e o seu potencial, que permite ao usuário selecionar ficheiros e comprimi-los ou descomprimi-los de uma forma simples e eficaz, melhorando assim a experiência do usuário na sua utilização.

Concluiu-se deste modo que o presente trabalho foi um desafio considerável face às adversidades sentidas, mas também se pode afirmar que a realização e a conclusão deste projeto excederam as expectativas esperadas.

Webgrafia

- https://en.wikipedia.org/wiki/Lempel–Ziv–Storer–Szymanski
- https://stackoverflow.com/questions/61774916/lzss-vs-lz77-compression-difference
- https://docs.rs/lzss/latest/lzss/
- https://www.researchgate.net/figure/LZSS-Encoding-Example_fig1_224264427
- https://moddingwiki.shikadi.net/wiki/LZSS_compression
- https://www.programiz.com/article/flowchart-programming
- https://www.programiz.com/python-programming
- http://docopt.org
- https://towardsdatascience.com/using-docopt-in-python-the-most-user-friendly-command-line-parsing-library-5c6aeac14deb
- https://docopt.readthedocs.io/en/latest/
- https://pypi.org/project/PySide/
- https://wiki.qt.io/Qt_for_Python
- https://www.qt.io/design
- https://www.qt.io/product/ui-design-tools