Centro de Formação de Alcântara - IEFP

Projecto 3 - PZYP: Compressão LZSS

(Relatório)

Carlos Mendes | Filipe Cavaco | Maria João Claro

L-EFLI NST-PROG07 - Técnico de Programação

UFCD 10794 - Programação Avançada com Python

Formador: João Galamba

3 de março de 2022

Introdução e Objectivos

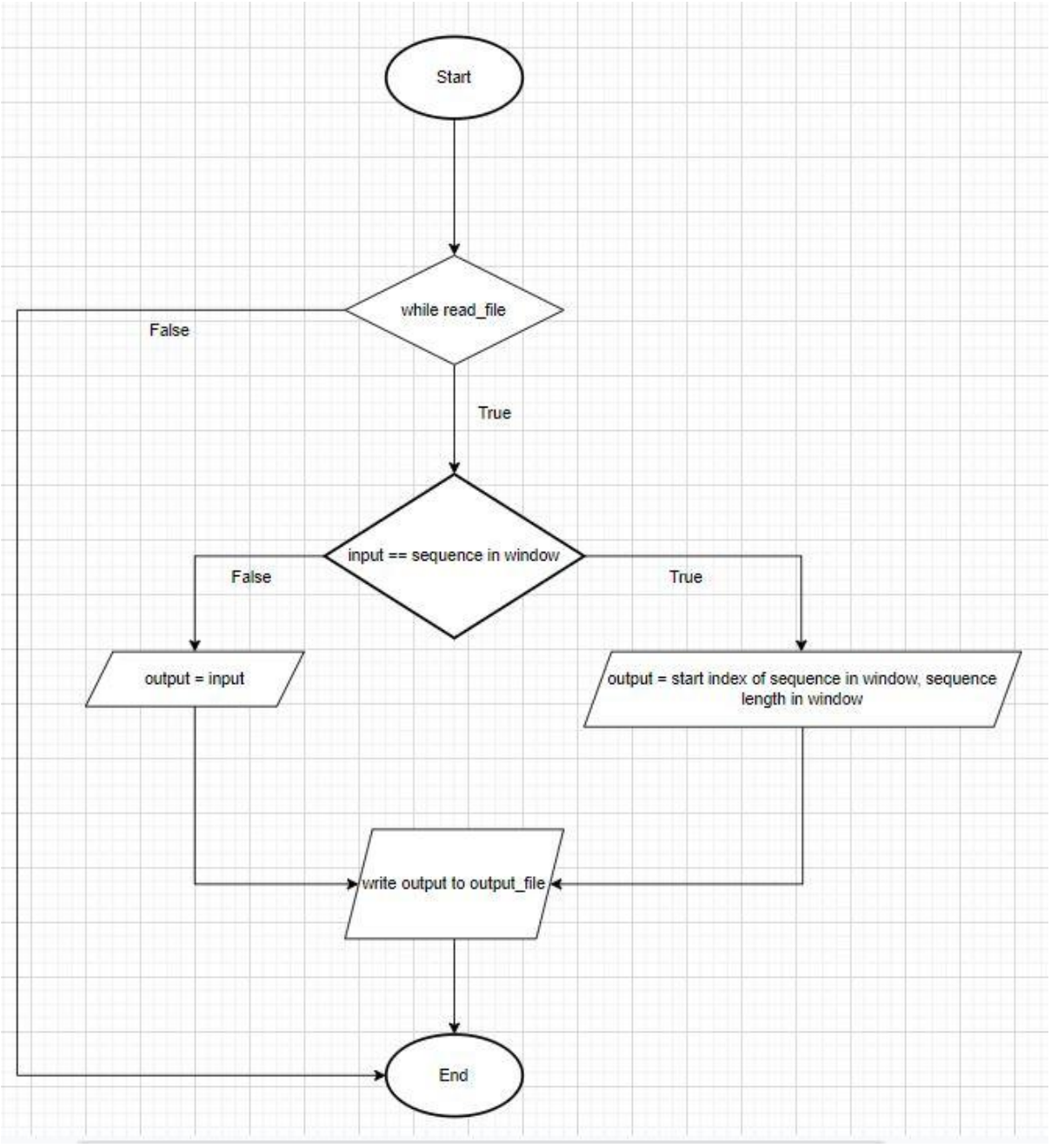
Neste relatório, apresenta-se o desenvolvimento de uma aplicação de compressão baseada no algoritmo LZSS, método de compressão bastante popular. O programa destina-se, assim, a realizar a compressão de ficheiros de quaisquer formatos e a respectiva descompressão.

O projecto proposto terá sido, porventura, no contexto do curso, o mais exigente em termos de organização do código. Nessa medida, constituía um desafio exigente para a aplicação de conhecimentos acumulados ao longo do curso. Da mesma forma, tendo em conta as pesquisas extensas na documentação oficial de Python e outras fontes na internet a que obrigava para a resolução de problemas colocados pelo enunciado, estimulava também a aquisição de conhecimentos novos.

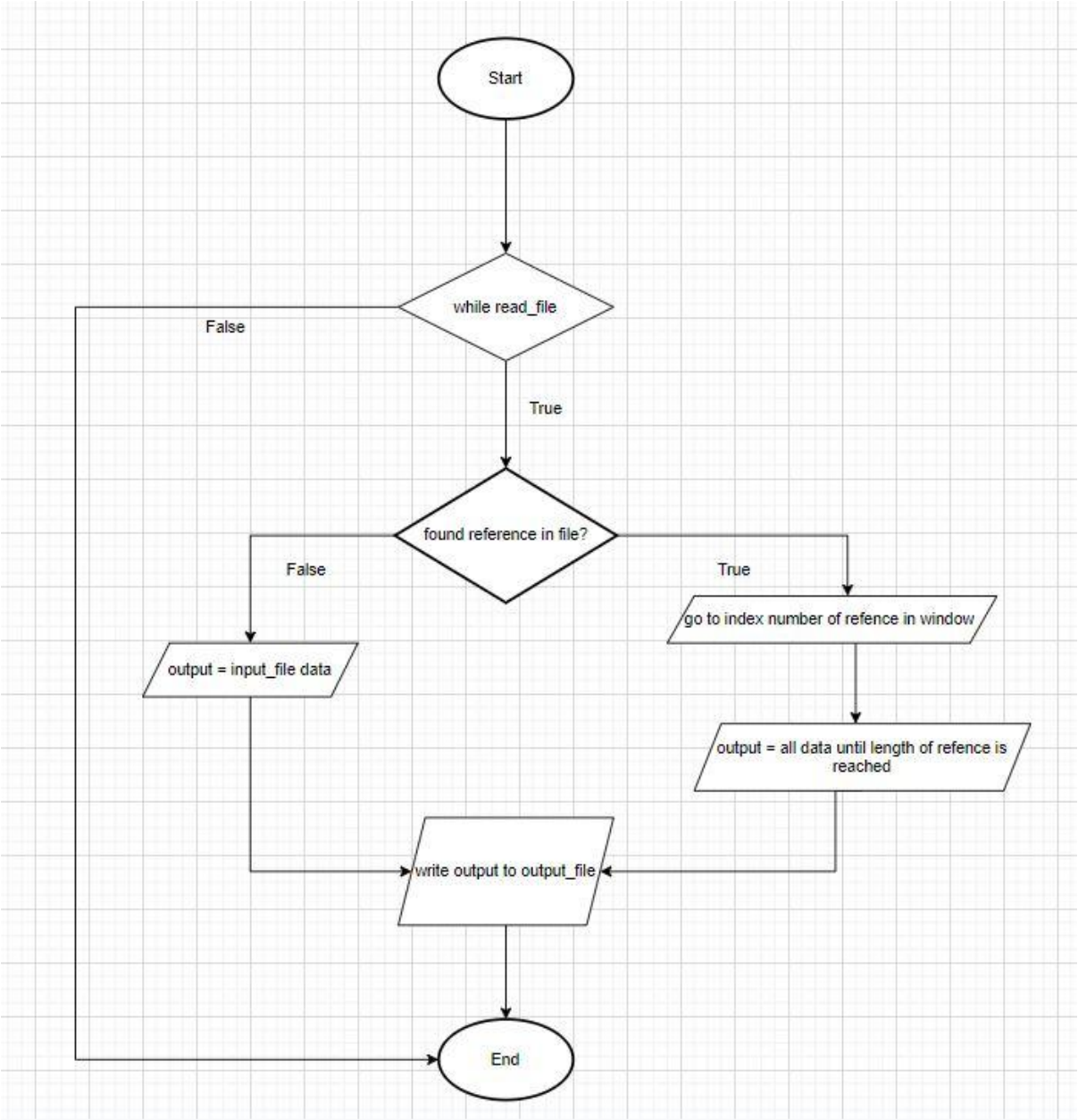
Na secção “Desenho e Estrutura”, a primeira do relatório, apresentam-se dois fluxogramas, onde se encontram representados graficamente os processos principais do compressor e do descompressor. Na segunda secção, relativa à implementação, descrevem-se de forma sucinta soluções empregues no desenvolvimento do compressor e também os módulos mais importantes utilizados.

Desenho e Estrutura

Compressor:



Descompressor:



Implementação

O algoritmo LZSS, em que se baseia o programa desenvolvido neste projecto, pertence à categoria de "algoritmos *de dicionário*". Esta classe de algoritmos caracteriza-se pela centralidade do conceito de "janela deslizante", um *buffer* ou zona de memória temporária que "desliza" pelos dados de entrada.

Na implementação deste conceito, optou-se pela criação de uma classe *Window*, em cujo construtor se inicializou uma variável do tipo deque. Esta servirá de *buffer* onde se armazenam os dados - uma espécie de zona de memória temporária, com uma dimensão fixa, garantindo melhor performance do programa - e que "deslizará" pelos dados de entrada, à procura de repetições.

Como, à medida que o *buffer* avança, "entram" nele os bytes mais recentes e "saem" os bytes mais antigos, um deque é uma **estrutura de dados** particularmente adequada, porquanto num deque com tamanho fixo, ao contrário do que acontece numa lista, os dados mais recentes acrescentados "empurram" para fora os dados mais antigos. Esta janela é instanciada nas **duas funções principais** que estruturaram o projecto desde o seu início - **as funções *encode* e *decode***.

Em relação à implementação da **função *encode***, começa-se por escrever o cabeçalho com as informações para usar no *decode*. Depois, lê-se o texto para memória e, por cada iteração do ciclo for, adiciona-se cada carácter (byte) a uma lista. Chama-se, então, a função *find* da classe *Window* que vai confirmar se os caracteres na lista estão na janela. Nessa altura, se sim - isto é, se os caracteres estiverem na lista -, escrevem-se as referências <distância, comprimento> no ficheiro de saída que substituem as repetições e indicarão ao descompressor, quando este for invocado, a localização dos dados que este deve repor. Se não estiverem, escreve-se o carácter (byte literal) no ficheiro. Finalmente, faz-se um *reset* à lista e adiciona-se o carácter à janela, retomando-se o ciclo.

Quanto à **função *decode***, antes de ser chamada, lê-se o cabeçalho no *main* e extrai-se de lá a informação (o tamanho da janela, o comprimento máximo de

caracteres e o nome do ficheiro sem a extensão .lzs). Quando a função *decode* é chamada, já o ficheiro certo se encontra aberto em modo de escrita, pondo-se o tamanho da janela e o comprimento máximo como argumentos da função, além dos dois ficheiros. A função propriamente dita chama então o método *get_dict* da janela que devolve o deque como lista. Num ciclo for, verifica-se então se cada elemento (byte) lido do ficheiro de entrada tem ou não tem um bit a antecedê-lo que serve para sinalizar se o byte está ou não comprimido. Se sim, o byte a seguir tem uma referência <distância, comprimento> que indica onde ir buscar os dados. Então, vai-se buscá-los à lista, juntando-se esses dados numa string de bytes, e, com outro ciclo for, colocam-se os bytes na janela, fazendo-se um *reset* da lista. Se não - ou seja, se não houver esse bit -, colocamos o elemento na string de bytes e na janela, fazendo-se um *reset* à lista. No fim, escreve-se a string de bytes no ficheiro.

Um dos **principais módulos utilizados** neste projecto é o próprio script fornecido pelo formador (**lzss_io**). Deste módulo, importaram-se as classes *PZYPContext, LZSSWriter* e *LZSSReader*. A classe *PZYPContext* fornece o conjunto de parâmetros que definirão a dimensão da janela em bits, ao passo que as classes *LZSSWriter* e *LZSSReader* permitem escrever e ler em bits, respectivamente.

O módulo **docopt**, importado para se ler as opções da linha de comandos, é outro módulo fundamental utilizado neste projecto. No script, definiu-se uma docstring com as diferentes opções de invocação do programa na linha de comandos. Estas opções podem ser argumentos em letra maiúscula ou entre os símbolos <>; opções começadas por “-” ou “--”; e ainda comandos entre parênteses curvos (significando que são obrigatórios) ou entre parênteses rectos (que são opcionais).

O módulo docopt transforma a docstring num dicionário que reflecte estas opções. Deste modo, quando se invoca o programa, “ARGS=docopt” torna a variável ARGS nesse dicionário. Se imprimissemos esse dicionário, encontraríamos várias chaves com valores booleanos “True” ou “False”. Assim, quando uma opção se encontra escrita na invocação do programa, essa opção/comando passa a “True” e, quanto às variáveis, assumem os valores dados. No caso deste programa, o nível de compressão tem “2” como valor por defeito na docstring (ou seja, se não se alterar

explicitamente esse valor na linha de comandos quando o programa é invocado - por exemplo, com “-l 4”, de acordo com um dos **extras** implementados neste projecto -, o valor de “l”, do nível de compressão, será sempre 2).

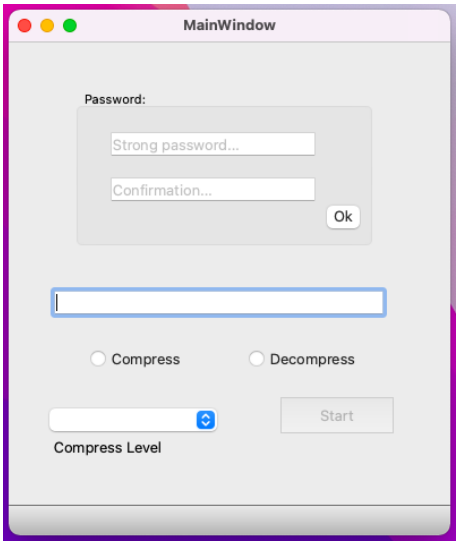
Para a geração de um cabeçalho, utilizou-se um outro módulo importante neste projecto, o módulo **struct**. No caso deste projecto, usaram-se as funções *pack* e *unpack* do módulo. Como a própria designação do módulo o indica, a função *struct.pack* converte os dados de python numa *struct* da linguagem C. Ou seja, no caso deste programa, converte a string de bytes numa struct. Então, há que indicar na função (através da string inicial) que tipo de dados vão estar representados (o formato) e, depois, os dados propriamente ditos. Isto significa que, se tivermos, por exemplo, “var = struct.pack('iii', 10, 20, 30)”, esta expressão quererá dizer que existem três inteiros (int’s) na struct (os valores 10, 20 e 30 indicados logo a seguir ao formato). Quando se utiliza a função *struct.unpack*, também tem de se declarar uma string igual, porque será isso que informará a função de quantos bytes estão na struct a "desempacotar/empacotar".

O módulo **time**, por sua vez, que fornece diferentes funções relacionadas com medidas temporais, foi importado para formatar a informação nesse cabeçalho sobre a data e a hora de compressão de um arquivo. Neste caso, antes de se escrever o cabeçalho, usa-se a função *time.ctime* para se extraírem a hora e a data em segundos, sendo que, depois, quando se lê o cabeçalho utilizando a opção -s (um dos três **extras** do projecto), convertem-se os segundos num formato de data.

Para encriptar estes ficheiros comprimidos com a introdução de uma palavra-passe, caso o utilizador tome essa opção, importou-se a classe *Fernet* da biblioteca **criptography**. Vale a pena ainda referir que a própria utilização de um deque, a estrutura de dados usada na janela, só é possível por ser importado do módulo **collections** que disponibiliza estruturas de dados alternativas àquelas integradas no Python 3.

Sempre que o programa é invocado sem opções, abre-se a interface gráfica do programa. Esta interface foi criada com a aplicação **Qt Designer**, a ferramenta para desenho de interfaces gráficas da biblioteca e framework Qt, disponibilizando ao

utilizador, através de dois *radio buttons*, a opção entre comprimir ou descomprimir um ficheiro carregado para a aplicação. Como acontece com a interface da linha de comandos, ao utilizador é também possível encriptar o ficheiro de saída, atribuindo-lhe uma palavra-passe. As caixas para introdução da password (*line edits*) estão agrupadas no interior de uma *group box*. É ainda possível definir o nível de compressão desejado, optando por um dos quatro valores disponibilizados através de uma *combo box*. As opções do utilizador são confirmadas com cliques em *push buttons* e as informações ao utilizador comunicadas por via de *message boxes*. Para o texto incluído na interface, são usadas *labels*.



Por fim, para a criação de ficheiros executáveis que permitam executar o programa em ambientes Windows e macOS, um **extra** implementado no projecto, recorreu-se ao pacote **PyInstaller**. Este, quando o comando “pyinstaller --onefile pzyp.py” é executado, identifica, por via da análise do script, os módulos e as bibliotecas indispensáveis à execução do script, reunindo-os num único pacote, executável sem necessidade de se instalarem um interpretador de python ou quaisquer módulos. Os executáveis correm apenas a versão gráfica do programa.

Conclusões

Neste projecto, desenvolvemos uma aplicação de compressão, em que se implementaram diversas funcionalidades, tais como: uma interface da linha de comandos e uma interface gráfica; uma opção que permite definir o nível de compressão; um cabeçalho; opções para se obter um resumo sobre o ficheiro comprimido e a ajuda do comando; por fim, a possibilidade de o utilizador definir uma palavra-passe usada para encriptar os ficheiros comprimidos.

O desenvolvimento deste programa aprofundou aspectos já trabalhados anteriormente em python no curso, como a implementação de funcionalidades com recurso a bibliotecas e módulos, e também de uma interface da linha de comandos.

Da mesma forma, permitiu praticar a escrita de código modular e desenvolver uma interface gráfica para a aplicação. Mais além, o projecto serviu também para estimular a integração de um sistema de controlo de versão nas nossas práticas de programação, no caso mediante o uso da popular plataforma GitHub.