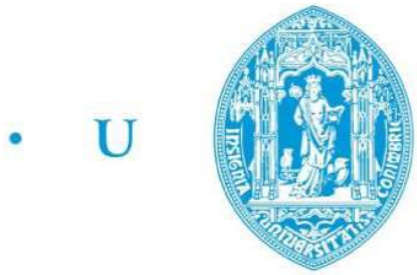

FCTUC FACULDADE DE CIÊNCIAS
E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Teoria da Informação

João Macedo: 2021220627

Johnny Fernandes: 2021190668

Miguel Leopoldo: 2021225940

18 de dezembro de 2022

Índice

Introdução

O presente relatório tem o objetivo de explicar o funcionamento do ficheiro *gzip.py* constante em anexo, e que diz respeito à implementação do algoritmo ***DEFLATE*** utilizado no software de compressão de ficheiros *gzip* e que é também ele composto pelo algoritmo *LZ77* e pelo sistema de códigos de *Huffman.* Em particular, o que se pretende com o projeto é, mais do que a implementação completa de todo o algoritmo DEFLATE, a criação e desenvolvimento de métodos na linguagem de programação *Python*, que permita o processo inverso, ou seja, a descompressão dos dados, para se fazer a descompressão do arquivo FAQ.txt.gz associado ao enunciado do projeto, ou seja, o *inflate* de dados.

É assim alvo deste relatório todo o processo utilizado na criação dos métodos e a sua lógica inerente para levar a cabo, com bom rigor, o objetivo do projeto.

O presente relatório dividir-se-á pelas diversas etapas do desenvolvimento dos métodos do ficheiro gzip.py, culminando em algumas considerações finais a levar em consideração pelo professor a quem o presente relatório se destina.

Breve explicação do algoritmo DEFLATE

O algoritmo DEFLATE é um método de compressão de dados que combina duas técnicas: o algoritmo *Lempel-Ziv 77* e codificação de *Huffman*.

O algoritmo LZ77 baseia-se na procura de padrões repetidos de dados e fazer a substituição dos ditos padrões por referências de outras repetições. Isto é feito através da utilização de uma "janela" (doravante chamada de *window*) deslizante, que faz a procura por padrões repetidos dentro de um determinado tamanho da *window*. Quando um padrão é encontrado, é substituído por uma referência que inclui o comprimento do padrão e a distância até a última ocorrência desse padrão na *window*.

A codificação de *Huffman*, por outro lado, é um método de compressão de dados que utiliza uma tabela de códigos para representar cada símbolo de dados de forma mais curta possível. Isso é feito atribuindo códigos mais curtos para os símbolos mais comuns e códigos mais longos para os símbolos menos comuns. No caso do nosso projeto esses códigos serão em binário a ser explicado com maior detalhe à frente no relatório.

Por fim, o algoritmo *DEFLATE* combina estas duas técnicas, primeiro utilizando o algoritmo *LZ77* para encontrar padrões repetidos de dados e substituí-los por referências, e depois utilizando a codificação de *Huffman* para codificar essas referências de maneira mais compacta possível. O resultado é uma representação compacta dos dados originais, que pode ser descompactada novamente para recuperar os dados originais. Essa descompactação será o trabalho no qual o presente projeto incide.

É importante notar que o algoritmo *DEFLATE* é apenas um dos muitos métodos de compressão de dados disponíveis, e pode não ser o mais eficiente em todos os casos. No entanto, é amplamente utilizado devido à sua eficiência e simplicidade de implementação.

gzip.py

Função ***decompress***

A função *decompress*, existente no ficheiro original *gzip.py* é a principal função para o desenvolvimento do programa e do projeto propriamente dito. É nesta função onde ocorre a descompressão dos blocos existentes no arquivo e onde, por conseguinte, são colocadas todas as chamadas a funções posteriormente criadas pelo grupo de trabalho e a serem explicadas abaixo. Convém ainda notar que o ficheiro *gzip.py* já inclui toda a leitura de cabeçalhos e parâmetros iniciais do formato *gzip*, terminando no byte a zero posterior ao nome do arquivo *FAQ.txt.gz*. É após este *null-terminator* que faremos a leitura de bytes para analisar os dados associados aos códigos de *Huffman* e ao algoritmo *LZ77* para a posterior descompressão dos dados.

Apesar do arquivo *FAQ.txt.gz* ter apenas um único bloco, dado o tamanho reduzido do ficheiro original, o desenvolvimento do código feito pelo grupo de trabalho faz também a descompressão de arquivos com um número arbitrário de blocos.

Na função *decompress*, a parte essencial do desenvolvimento tem a ver como mencionado anteriormente, com a descompressão dos blocos. A função possui um ciclo *while* que começa por verificar se último bloco analisado é ou não um bloco final. Se for um bloco final, termina assim a descompressão dos dados. Caso não seja, o ciclo irá continuar a desenrolar-se até que os blocos estejam todos eles descompressos.

Durante uma descompressão de bloco, é importante ter em conta os seguintes aspectos, segundo o *RFC 1951* (cujo conteúdo será mencionado ao longo deste relatório):

- Cada bloco possui conjunto de três árvores de *Huffman*: uma primeira árvore para a descompressão das outras duas, nomeadamente, a árvore dos literais/comprimentos, e a árvore das distâncias.

- Cada bloco possui um conjunto de dados (após os dados referentes às árvores de *Huffman*) os quais, com apoio às árvores de *Huffman,* dará origem a dados descompressos.

- Cada bloco poderá ter um tamanho arbitrário, estando apenas blocos não comprimíveis limitados a um total de 65536 bytes.

Assim, conforme a necessidade de leitura de dados para a descompressão dos blocos, são chamadas nesta função outras funções criadas pelo grupo de trabalho, nomeadamente:

- Chamada de função para a leitura dos primeiros *HCLEN+4* conjuntos de 3 bits, que seguirá como índice a ordem enunciada no *RFC 1951* (3.2.7). É com estes dados que iremos construir a primeira árvore de *Huffman* que dará origem às outras duas, através da função *get_alphabet_code_len* e da função *create_huffman_tree*.

- Chamada de função para a leitura dos *HLIT+257* elementos (leitura de bits arbitrária pois depende dos *match* da árvore de *Huffman*), com recurso à nossa árvore de *Huffman* anteriormente criada. Assim, é feita a leitura bit a bit até se encontrar uma cópia do código *Huffman* na primeira árvore criada. É então guardado o valor da folha da árvore numa lista. Recomendamos verificar em detalhe a função *get_code_len* para melhor compreensão do processo. Esta secção irá dar origem à árvore de *Huffman* dos literais/comprimentos.

- Chamada de função para a leitura dos *HDIST+1* elementos (leitura de bits arbitrária pois depende dos *match* da árvore de *Huffman*), com recurso também à primeira árvore de *Huffman*. Em processo similar, recomendamos verificar a função *get_code_len* para melhor compreensão. Esta secção irá dar origem à árvore de *Huffman* das distâncias.

- Chamada de função *decode*, passando como argumento as duas árvores mencionadas supra e uma *window*, limitada a 32768 últimos elementos do bloco anterior (caso seja o primeiro bloco, naturalmente, não haverá valores na *window*).

- Por fim, chamada de função para escrever os dados no ficheiro destino (o nome do ficheiro destino utiliza o parâmetro *self.gzh.fName* para recuperar o nome do ficheiro original).

As funções seguintes foram desenvolvidas na totalidade pelo grupo de trabalho para contemplar as diversas etapas necessárias à descompressão de dados, começada nesta função *decompress*.

Função ***get_alphabet_code_len***

Esta função diz respeito à leitura dos *HCLEN+4* conjuntos de 3 bits, guardando pois, o seu valor decimal numa lista que será posteriormente passada à função *code_len_to_huffman_code*, juntamente com a ordem mencionada na função anterior, constante no ponto *3.2.7 do RFC*, para serem gerados os seus valores binários e poder ser criada a árvore de *Huffman* que será então utilizada para a criação das árvores dos literais/comprimentos e a árvore das distâncias.

Função ***get_code_len***

É nesta função que vamos buscar os valores dos comprimentos das folhas nas árvores de *Huffman* dos literais/comprimentos e das distâncias. Fazemos isso num ciclo que se vai repetir um número diferente de vezes, dependendo do *HLIT* e do *HDIST*, em que em cada iteração vamos buscar o valor do *node* da árvore de *Huffman* usada para descodificar as outras duas.

Após ler o valor, fazemos uma verificação para saber se ele está entre 0 e 15, inclusive. Se estiver, este valor é interpretado como um literal e adicionado à lista das distâncias dos códigos. Se não estiver, procedemos como está mencionado no *slide* 37 do *Doc1* que nos foi fornecido. Quando todas as distâncias dos códigos forem descodificadas, essa lista será devolvida e usada para criar uma árvore de *Huffman*.

Função ***code_len_to_huffman_code***

A função code_len_to_huffman_code é uma função essencial no nosso programa, uma vez que é esta a função que codifica as distâncias dos códigos em códigos de Huffman, para posterior criação da árvore de Huffman com a função create_huffman_tree. Esta função recebe como parâmetros os códigos dos comprimentos e os seus literais. Tome-se como exemplo a primeira árvore de Huffman, cujos valores são os valores decimais dos HCLEN+4 conjuntos de 3 bits e cujos valores literais vão de 0 a 18 pela ordem constante no RFC. Para cada valor literal, verifica-se se o seu código do comprimento é ou não zero. Se for, removem-se estes valores, uma vez que os zeros não têm qualquer tipo de utilidade neste caso. Assim, a função reordena estes valores por ordem crescente por forma a que

possamos então passar à criação dos códigos de Huffman. Assim, para cada valor, se o comprimento for igual ao comprimento do código de comprimento anterior, incrementamos um. Caso haja uma alteração de comprimento (crescente, lembre-se que já houve uma reordenação dos valores), é feito um shift à esquerda e adiciona-se mais um. Desta forma, o que se garante são códigos de Huffman com existência de um prefixo ótimo e sem existência de duplicados, que são posteriormente adicionados à árvore de Huffman para poderem ser pesquisados durante a criação das próximas duas árvores, dos literais/comprimentos e das distâncias. O código de comprimento de menor valor é também o número de bits que possui o primeiro elemento da árvore.

Função ***create_huffman_tree***

A função *create_huffman_tree* faz uso do *import Huffmantree*, um ficheiro *Python* fornecido para o desenvolvimento do projeto e que possui duas funções cruciais usadas no nosso programa: *addNode* e *nextNode*. É com base nestas funções e no ficheiro *Huffmantree* que são criadas as árvores de *Huffman* ao longo do projeto. É assim passado como argumento uma lista que contém duas listas, uma relativa aos códigos que darão origem aos ramos da nossa árvore, e outra lista que contém os valores dos *nodes*/folhas. Assim, quando feita a pesquisa nas árvores de *Huffman*, é passado como argumento à função *nextNode* o código de *Huffman* (caminhos pelos ramos da árvore) e é devolvido o valor do *node*/folha.

Função ***decode***

A função *decode* é onde vamos buscar a última parte da informação do bloco. Com a ajuda das árvores de *Huffman* dos literais/comprimentos e das distâncias, que foram descodificadas anteriormente, conseguimos descodificar o resto da informação.

Nesta função vamos entrar num ciclo que se vai repetir até chegarmos ao final do bloco. A cada iteração vamos procurar um *node* da árvore dos literais/comprimentos com a ajuda da função *search_bit_by_bit* e dependendo do seu valor, vamos trabalhar de forma diferente. Se o valor do node estiver entre 0 e 255, inclusive, interpretamos o valor como um literal e guardamos diretamente na nossa *window* de informação. Caso seja 256 significa que chegamos ao final do bloco, portanto saímos do ciclo e não lemos mais

informação, pois esta pertence ao próximo bloco. Se for entre 257 e 264, interpretamos como um comprimento.

O valor do comprimento vai ser calculado segundo o ponto 3.2.5 do *RFC*. Como foi interpretad um comprimento, precisamos também de descodificar uma distância. Fazemos isso ao procurar um *node*, desta vez na árvore das distâncias. Quando é encontrado o *node*, outra vez com a ajuda da função s*earch_bit_by_bit*, interpretamos o valor lido como uma distância de acordo com a segunda tabela no ponto 3.2.5 do *RFC*.

Agora que temos um comprimento e uma distância, recuamos na *window* a distância e copiamos os valores com o comprimento calculado para o final da *window*. Como a *window* pode entrar na função já com valores, pois podem ser precisos em casos que há mais do que um bloco, temos de os remover antes de devolver a *window* para não haver repetição de informação. Após isso é devolvida a *window* de informação.

Função ***write_data***

A função *write_data* é bastante simples no entanto crucial, pois é a função que escreve os dados descomprimidos no ficheiro de output. É assim feita a leitura dos dados que lhe são passados e, a cada caracter *ASCII* obtido, é então escrito no ficheiro de texto destino. Convém indicar que a cada bloco descomprimido, é chamada a função *write_data* para escrever os dados obtidos pela descompressão do bloco. Assim, para cada bloco que se lhe some, os dados são concatenados no ficheiro, aberto no modo *append* com acesso de escrita.

Função ***search_bit_by_bit***

A função search_*bit_by_bit* declarada no ficheiro gzip.py vai encontrar uma folha da árvore de *Huffman* passada como parâmetro. Na função temos um ciclo que vai chamar a função *nextNode* do ficheiro *Huffmantree*, cuja função é verificar se para um determinado número de bits pesquisado, existe ou não uma correspondência de código na árvore, que leva a um node. Caso não seja encontrado um node cuja ramificação é a do código binário que obtivemos, continuamos a pesquisar próximos bits até formar o código de *Huffman* existente na árvore.

Existe total garantia da existência de uma correspondência entre a procura de bits e o código de *Huffman* na árvore, uma vez que a codificação e a descodificação usam o mesmo processo. Assim, a pesquisa de bits é permanente até se encontrar uma correspondência.

Por fim, essa correspondência é devolvida (o valor do *node*/folha encontrado).

Considerações finais

Por fim, em formato de conclusão, o grupo decidiu que seria importante ter as seguintes informações em conta. Primeiramente, apesar do arquivo *FAQ.txt.gz* ser um arquivo de texto com um tamanho extremamente reduzido e apenas um bloco de dados comprimidos, decidimos elaborar o programa por forma a ser possível descomprimir qualquer arquivo de texto com um número arbitrário de blocos. Assim, decidimos incluir nos nossos ficheiros um ficheiro de texto com o nome *bible.txt.gz* com 67 blocos de dados comprimidos.

Por outra via, é também essencial assinalar o facto de que o programa apenas aceita ficheiros de texto comprimidos em formato *gz* com o algoritmo *DEFLATE* que use o método de códigos de *Huffman* dinâmicos, sendo que o programa não está preparado para qualquer outro tipo de arquivo.

Todo o código foi comentado para uma maior legibilidade e compreensão por parte do professor tendo em conta que a sua defesa será feita de forma não presencial.

Consideramos ter atingido os resultados necessários à conclusão do enunciado, tendo desenvolvido todas as partes das diferentes etapas do trabalho delegado para cada semana de aula.

Esperamos que o resultado seja apreciado pela equipa docente e que o trabalho tenha sido desenvolvido com distinção.