BZIP2

O algoritmo de compressão utilizado neste trabalho é o Bzip2. Foi crido em 1996 por Julian Seward e disponibilizado gratuitamente em formato open-source. Este algoritmo combina diversas técnicas de compressão, tais como Run Length Encoding, Códigos de Huffman e o algoritmo Burrows–Wheeler, permitindo-lhe assim obter maiores taxas de compressão que algoritmos mais antigos, como o Deflate e o LZMA. Foi implementado no nosso programa através da livraria bz2 do Python.

O módulo bz2 contém:

-a função open () e a classe BZ2File para ler e gravar arquivos comprimidos;

-as classes BZ2Compressor e BZ2Decompressor para (des)compressão incremental;

- as funções compress () e decompress () para comprimir/descomprimir;

A função **open()** permite abrir um ficheiro comprimido com bzip2 em modo binário ou de texto, retornando um ficheiro de objeto. Esta função pede como argumento um ficheiro, podendo ser dado o nome de ficheiro real (uma string), ou um ficheiro de objeto existente para ler ou escrever. O argumento de modo pode ser qualquer um de 'r', 'rb', 'w', 'wb', 'x', 'xb', 'a' ou 'ab' para o modo binário, ou 'rt', 'wt' , 'xt' ou 'at' para modo de texto. O padrão é 'rb'.

Também é possível definir o nível de compressão pretendido, através de um inteiro entre 1 e 9.

A função **compress()** recebe dados para comprimir e devolve-os em bloco, ou então devolve uma string vazia. Quando termina, chama o método flush(), para concluir o processo de compressão.

Fornece dados para o objeto compressor. Retorna um bloco de dados compactados, se possível, ou uma string de byte vazia, caso contrário. Quando terminar de fornecer dados ao compressor, chame o método flush () para finalizar o processo de compressão.

A função **flush()** conclui o processo de compressão. Retorna os dados comprimidos deixados em buffers internos. O objeto compressor não pode ser usado após este método ter sido chamado.

A função **decompress()** recebe dados (um objeto semelhante a bytes), devolvendo-os não compactados na forma de bytes. Alguns dos dados podem ser armazenados internamente, para uso em chamadas posteriores para decompress (). Os dados retornados devem ser concatenados com a saída de quaisquer chamadas anteriores deste método. A tentativa de descomprimir os dados após o final do fluxo ser atingido gera um EOFError. Quaisquer dados encontrados após o final do fluxo são ignorados e salvos no atributo unused\_data.

A função **eof()** devolve **True** se o marcador de fim do fluxo de dados tiver sido alcançado.

Etapas do BZIP2

1. **Run-length encoding** (RLE) nos dados iniciais: a ideia basica por tras do RLE (Run length Encoding) é mapear uma sequéncia de numeros numa sequência de pares de simbolos (n° de ocorréncias contiguas do valor, valor), contando cada par como um simbolo.



1. O **Método de Burrows-Wheeler** é um processamento estatístico de um bloco de dados que aumenta a redundância espacial, facilitando a aplicação de técnicas de compressão de dados. Neste caso específico, os dados são divididos em blocos com um tamanho máximo de 900 kB. O bloco é totalmente autocontido, com buffers de entrada e saída permanecendo do mesmo tamanho. Para a ordenação em bloco, uma matriz (nocional) é criada, na qual a linha i contém todo o buffer, girada para começar a partir do i-ésimo símbolo. Após a rotação, as linhas da matriz são classificadas em ordem alfabética (numérica). Um ponteiro de 24 bits é armazenado marcando a posição inicial para quando o bloco não é transformado. Na prática, não é necessário construir a matriz completa; em vez disso, a classificação é realizada usando ponteiros para cada posição no buffer. O buffer de saída é a última coluna da matriz; ele contém todo o buffer, mas reordenado para que provavelmente contenha grandes séries de símbolos idênticos.
2. Com a **transformação Move-to-front**, o tamanho do bloco a ser processado não é alterado. Cada um dos símbolos em uso no documento é colocado numa matriz. Quando um símbolo é processado, ele é substituído pela sua localização (índice) na matriz, sendo o símbolo colocado no início da matriz. Isto faz com que os símbolos imediatamente recorrentes sejam substituídos por símbolos zero (séries longas de qualquer símbolo arbitrário tornam-se séries de símbolos zero), enquanto outros símbolos são remapeados de acordo com a sua frequência local. Muitos dados "naturais" contêm símbolos idênticos que se repetem dentro de um intervalo limitado (o texto é um bom exemplo). A transformação MTF atribui valores baixos aos símbolos que aparecem com frequência, muitos deles sendo idênticos (diferentes símbolos de entrada recorrentes podem mapear para o mesmo símbolo de saída). Estes dados podem posteriormente ser codificados de forma muito eficiente por qualquer método de compressão.
3. **Run-length encoding (RLE)** no resultado do passo 3.
4. **Código de Huffman**: este processo substitui símbolos de comprimento fixo no intervalo de 0 a 258 por códigos de comprimento variável com base na frequência de uso. Códigos usados ​​com mais frequência acabam mais curtos (2–3 bits), enquanto códigos raros podem ser representados em até 20 bits. Os códigos são selecionados cuidadosamente para que nenhuma sequência de bits possa ser confundida com um código diferente (códigos de prefixo).

Diagram

Description automatically generated

1. **Seleção de tabelas de Huffman**: várias tabelas de tamanho idêntico podem ser usadas com um bloco se o ganho de as usar for maior do que o custo de incluir a tabela extra. Podem estar presentes pelo menos 2 e até 6 tabelas, com a tabela mais apropriada sendo selecionada novamente antes de cada 50 símbolos processados, eliminando a necessidade de fornecer continuamente novas tabelas.
2. **Codificação unária da seleção das tabelas de Huffman**: se várias tabelas de Huffman estão em uso, a seleção de cada tabela (numeradas de 0 a 5) é feita a partir de uma lista por um bit de terminação zero, que tem entre 1 e 6 bits de comprimento. A seleção está numa lista MTF das tabelas. No caso comum de se continuar a usar a mesma tabela Huffman, a seleção é feita com um único bit. Esta codificação unária pode ser vista como uma árvore de Huffman levada ao extremo, em que cada código tem metade da probabilidade do código anterior.
3. Os comprimentos de bits do código de Huffman são necessários para reconstruir cada uma das tabelas de Huffman canônicas usadas. Cada comprimento de bit é armazenado como uma diferença codificada em relação ao comprimento de bit do código anterior. Um bit zero (0) significa que o comprimento do bit anterior deve ser duplicado para o código atual, enquanto um bit (1) significa que um bit adicional deve ser lido e o comprimento do bit deve ser aumentado ou diminuído com base nesse valor. No caso comum, um único bit é usado por símbolo por tabela e o pior caso - indo do comprimento 1 ao comprimento 20 - exigiria aproximadamente 37 bits. Como resultado da codificação MTF anterior, os comprimentos de código começariam com 2–3 bits (códigos usados com frequência) e aumentariam gradualmente, o que significa que o formato delta é bastante eficiente, exigindo cerca de 300 bits (38 bytes) por tabela Huffman completa .
4. **Sparse bit array**: um bitmap é usado para mostrar quais os símbolos usados dentro do bloco e que devem ser incluídos nas árvores de Huffman. Os dados binários provavelmente usarão todos os 256 símbolos representáveis por um byte, ao passo que os dados textuais podem usar apenas um pequeno subconjunto de valores disponíveis, talvez cobrindo a faixa ASCII entre 32 e 126. Armazenar 256 bits zero seria ineficiente se eles estivessem quase todos sem uso. Os 256 símbolos são divididos em 16 blocos, e somente se os símbolos desse bloco são usados é que se inclui uma matriz de 16 bits. A presença de cada um desses 16 intervalos é indicada por uma matriz de bits adicional de 16 bits no início. O bitmap total usa entre 32 e 272 bits de armazenamento (4–34 bytes).

Observação dos resultados

Como podemos observar na tabela, as imagens comprimidas com BZIP2 ficaram com tamanhos muito próximos das imagens em PNG, sendo que no caso da imagem “landscape.bmp” consegue um tamanho de compressão menor que o PNG.

|  |  |  |  |
| --- | --- | --- | --- |
| Nome do ficheiro | Tamanho original | Tamanho PNG | Tamanho BZIP2 |
| egg.bmp | 17.75 | 4.63 | 4.75 |
| landscape.bmp | 11.01 | 3.35 | 3.49 |
| pattern.bmp | 48.01 | 2.28 | 1.81 |
| zebra.bmp | 16.74 | 5.47 | 5.63 |

Fonte: <https://docs.python.org/3/library/bz2.html>, https://en.wikipedia.org/wiki/Bzip2.