Guia Apresentação TI

INTRODUÇÃO

ETAPAS DA CODIFICAÇÃO SEM PERDAS

Transformação: Esta etapa aplica uma transformação reversível aos dados da imagem de entrada. Por norma, filtra-se a distribuição estatística dos dados e/ou agrupa-se uma grande quantidade de informações em poucas amostras de dados.

Mapeamento de dados para símbolo: Esta etapa converte os dados da imagem em símbolos que podem ser codificados com eficiência na etapa final. Os dados de imagem podem ser divididos em blocos de amostras de dados vizinhas (cada bloco é um símbolo). O agrupamento de várias unidades de dados pode resultar em taxas de compressão mais altas à custa do aumento da complexidade da codificação. A ideia básica por trás do RLE (Run-length Encoding) é mapear uma sequência de números numa sequência de pares de símbolos (nº de ocorrências contíguas do valor, valor), contando cada par como um símbolo.

Codificação de símbolo sem perdas: Esta etapa gera um fluxo de bits binário atribuindo palavras de código binárias aos símbolos de entrada.

CRITÉRIOS DE SELEÇÃO

1. Eficiência de compressão: A taxa de compressão corresponde ao número de vezes em que se diminui o tamanho da fonte. Quanto maior for esta, mais eficaz é o algoritmo.

2. Atraso de codificação: o tempo mínimo necessário para codificar e decodificar uma amostra de dados de entrada. O atraso de codificação aumenta com o número total de operações aritméticas necessárias. Geralmente também aumenta com o aumento dos requisitos de memória. Minimizar o atraso de codificação é importante para aplicações em tempo real.

3. Complexidade da implementação: medida em termos do número total de operações aritméticas necessárias e em termos dos requisitos de memória. Uma maior eficiência de compressão geralmente pode ser alcançada aumentando a complexidade da implementação, o que levaria a um aumento do atraso de codificação. Na prática, deve-se otimizar a eficiência de compressão, mantendo os requisitos de implementação tão simples quanto possível.

4. Robustez: Para aplicações que requerem transmissão do fluxo de bits comprimido em ambientes sujeitos a erros, a robustez do método de codificação para erros de transmissão torna-se uma consideração importante.

5. Escalabilidade: os codificadores escaláveis geram um fluxo de bits em camadas que incorpora uma representação hierárquica dos dados da imagem de entrada. Desta forma, os dados de entrada podem ser recuperados em diferentes resoluções de forma hierárquica e a taxa de bits pode ser variada dependendo dos recursos disponíveis.

BZIP2

Desenvolvido por Julian Seward, BZIP2 usa RLE em conjunto com método de Burrows-Wheeler e a técnica de move-to-front. Em geral a técnica de RLE é uma técnica auxiliar que não gera uma compressão muito grande sozinha, mas que aliada a outras técnicas é um instrumento simples e poderoso de compressão. BZIP2 não é apenas um algoritmo e dá nome a um software de compressão de arquivos. A sua licença é livre e de código aberto (*open source*) e comprime a maioria dos formatos de arquivos de maneira mais eficiente do que o tradicional gzip, de Jean-loup Gailly e Mark Adler. Desta maneira é claramente similar aos algoritmos de compressão de gerações recentes.

BIBLIOTECA BZIP2 EM PYTHON (BZ2)

A função **open()** permite abrir um ficheiro comprimido com a codificação bzip2 (".bz2"), em modo binário ou de texto, e dá retorno de 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'.

A função **compress()** recebe dados para comprimir e devolve-os em bloco, ou então devolve uma string vazia. Fornece dados para o objeto que comprime. Retorna um bloco de dados compactados, se possível, ou uma string de byte vazia, caso contrário. Quando terminar de fornecer dados ao compressor e chama o método flush () para finalizar o processo de compressão. Também é possível definir o nível de compressão pretendido, através de um inteiro entre 1 e 9, sendo que, quanto mais perto de '1', menos etapas de compressão são executadas e/ou são executadas de uma maneira menos exaustiva e, quanto mais perto de '9', o contrário.

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 resultante de uma compressão), 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 de saída 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 DA COMPRESSÃO BZIP2

1. Run-length encoding (RLE): esta técnica é aplicada nos dados iniciais, após a sua leitura. A ideia por de trás deste método é mapear uma sequência de números numa sequência de pares de símbolos (n° de ocorrências contíguas do valor, valor), contando cada par como um símbolo. É uma técnica normalmente utilizada como auxiliar e que não gera uma compressão muito grande sozinha, mas que é muito eficaz para fontes de informação com grande repetição de valores e que se torna poderosa aliada a outras técnicas. No contexto da TP2, as imagens fornecidas (monocromáticas) não podem assumir uma grande variedade de valores, uma vez que estão limitadas a uma escala de cinzentos.

2. Método de Burrows-Wheeler: 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 auto-contido com *buffers* de entrada e saída, permanecendo do mesmo tamanho. Para a ordenação em bloco, é criada uma matriz, na qual a linha i contém todo o *buffer*, sofrendo uma rotação para começar no i-ésimo símbolo. Após a rotação, as linhas da matriz são ordenadas alfabeticamente (numericamente). É armazenado um ponteiro de 24 bits que marca a posição inicial, caso o bloco não seja transformado. BZIP2, em vez de construir uma matriz completa (demorado e ocupa espaço), realiza a classificação usando ponteiros para cada posição no *buffer*. O *buffer* de saída é a última coluna da matriz, contendo grandes séries de símbolos idênticos.

3. Move-to-front: cada um dos símbolos em uso é colocado numa matriz. O tamanho do bloco a ser processado não é alterado. Quando um símbolo é processado, é substituído pela sua localização na matriz (índice), sendo o símbolo colocado no início da matriz (*Move-to-front*). 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 mapeados novamente de acordo com a sua frequência local. A transformação MTF (*Move-to-front*) atribui valores baixos aos símbolos que aparecem com frequência, muitos deles sendo idênticos (diferentes símbolos de entrada recorrentes podem ser mapeados para o mesmo símbolo de saída). Estes dados podem posteriormente ser codificados de forma muito eficiente por qualquer método de compressão.

4. Run-length encoding (RLE) no resultado do passo 3.

5. 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. Símbolos usados com mais frequência são codificados com códigos mais curtos (2–3 bits), enquanto que símbolos raros podem ser representados com códigos com até 20 bits de informação. Os códigos são selecionados cuidadosamente para que nenhuma sequência de bits possa ser confundida com um código diferente (códigos de prefixo).

6. Seleção de tabelas de Huffman: várias tabelas de tamanho idêntico podem ser usadas com um só bloco se o ganho (processamento, memória ou tempo) for maior do que o custo de incluir uma tabela extra. Podem estar presentes pelo menos 2 e até 6 tabelas. A tabela mais apropriada é selecionada novamente antes de cada 50 símbolos processados, eliminando a necessidade de fornecer continuamente novas tabelas.

7. Codificação unária da seleção das tabelas de Huffman: se várias tabelas de Huffman estão em uso, a seleção de cada tabela (no máximo 6, numeradas de 0 a 5) é feita por um bit de terminação zero numa lista (*Move-to-front* das tabelas) que tem entre 1 e 6 bits de comprimento. No caso mais comum, onde 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.

8. Codificação Delta: os comprimentos de bits do código de Huffman são necessários para reconstruir cada uma das tabelas de Huffman usadas. Cada comprimento de bit é armazenado como a codificação da diferença 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 mais comum, é usado um único bit por símbolo por tabela e, no pior caso, indo do comprimento 1 ao comprimento 20, exigiria aproximadamente 37 bits. Como resultado da codificação MTF anterior, os comprimentos de código começam com 2–3 bits (códigos usados com frequência) e aumentariam gradualmente. O formato *Delta* é bastante eficiente, exigindo cerca de 300 bits (38 bytes) por tabela *Huffman* completa.

9. Sparse bit array: é usado um bitmap 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, enquanto que os dados textuais podem usar apenas um pequeno subconjunto dos 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).

ANÁLISE DE DADOS

Como é possível observar na tabela acima, as imagens comprimidas com o método “BZIP2” ficaram com tamanhos muito próximos das imagens em “PNG”, sendo que, no caso do ficheiro “pattern.bmp”, o ficheiro “.bz2” correspondente consegue um tamanho menor que o “PNG”, para a mesma imagem. O processo de compressão é feito de uma maneira extremamente rápida, bem como o de descompressão, que acaba por ser executado de uma forma ainda mais breve que o primeiro. Estes resultados são interessantes na medida em que BZIP não é um codec específico para compressão de imagens, mas sim para uma grande variedade de ficheiros. O bom desempenho observado pode ser justificado pela escolha inteligente das técnicas de compressão e, sobretudo, da ordem em que são aplicadas aos dados. Experimentalmente, pôde-se demonstrar que o método “BZIP2” foi extremamente eficaz na compressão das imagens em estudo na TP2, todas monocromáticas (escala de cinzentos). Analisando os histogramas e a tabela do ponto V, conclui-se, como previsto, que fontes com maior redundância de informação, ou seja, com maior ocorrência de símbolos repetidos, alcançam mais facilmente elevadas taxas de compressão do que as fontes com menor redundância.