Compresão de arquivos com Deflate (HuffmanCode + LZ77)

Autor

Nome : Lucas Silva Amorim **RA** : 11201720968 **Github** : lucas170198 **Video Youtube**: https://youtu.be/NoIF2FrbOw0

O que este projeto faz?

Aplicação CLI de uma versão simplificada do GZIP (sem todos os headers e a formatação convencionada para o arquivo) utilizando o Deflate, algorítmo de compressão de dados que une a codificação de Huffman e o LZ77. A ideia é trazer uma interpretação funcional ao algorítmo (que em grande parte das vezes é descrito e implementado em termos imperativos).

Motivação

Este projeto implementa uma versão de um dos métodos mais conhecidos de compressão sem perda de dados atualmente. Comprimir um arquivo sem perda de dados, significa diminuir o espaço em memória por ele (o transformando em outro arquivo), porém sem haver perda de informação.

O **GZIP** é um programa de compressão/descompressão de arquivo escrito por Jean-loup Gailly para o projeto GNU, este programa é largamente utilizado por servidores WEB na transferência de arquivos, já que, ele é extramente eficiente para reduzir o tamanho de arquivos que contém muita repetição em seu conteúdo (o que é o caso das tags HTML e XML dos arquivos transferidos na WEB). Por baixo dos panos, ele utiliza um algorítimo chamado **DEFLATE** que basicamente combina outros dois algorítmos clássicos de compressão (a **codificação de Huffman** e o **LZ77**).

A ideia do **LZ77** é basicamente substituir partes do arquivo que já foram lidas anteriormente (que estão no dicionário) por "ponteiros" que apontam para a última ocorrência do texto encontrado. Para isso, o algoritmo divide o texto entre o **dicionário** (informações já processadas pelo algorítimo) e o **lookahead buffer** (informações que serão processadas). - A cada iteração, o algorítmo busca a maior string possível presente no buffer que combina com algum dos registros presentes no dicionário. - Caso encontre esse registro, substitui o texto por uma tupla <offset, length> onde offset representa a quantas posições atrás estava o registro encontrado, e length contém o tamanho da string - Se o registro não for encontrado, um novo registro é gravado com a tripla de valores <0, 0, caractere>.

Ok, ficou claro que este método reduzirá o tamanho do arquivo (principalmente para os arquivos com muita repetição de caractéres), porém a estrutura de tuplas e triplas não é tão econômica em questão de espaço, dado que tanto offset e length precisam de alguns bytes para ser guardados (imagine um arquivo muito grande, e que no meio deste arquivo um string que se repetiu no começo é encontrado. Precisariamos de um espaço grande para guardar o offset). Outro problem é que, o tamanho dos registros é variável (pode ser uma tupla ou tripla, dependendo se é a primeira vez ou não que o caractere em encontrado). É ai que surge a **codificação de Hamming**!

O método de codificação de Hamming é também muito eficiente para comprimir arquivos com muita repetição. Além disso, uma das coisas que o levaram a ser criado foi conseguir armazenar valores utilizando tamanhos diferentes, o que se encaixa perfeitamente no probelma do LZ77 descrito a cima! O algoritmo constroi uma árvore binária com os caracteres, baseado na frequência com que eles aparecem e usa essa arvore para codificar a informação (fazendo o passeio na árvore até a folha que representa aquele caractere e escrevendo 0 ou 1 para "marcar o caminho tomado até lá").

O **DEFLATE** junta o melhor dos dois mudos, o texto é utilizado como entrada para o LZ77, e a saida do LZ77 serve como entrada para a codificação de Hamming.

Modulos da aplicação

- **Main** : Aplicação cli e algumas funções básicas para auxiliar na leitura e escrita de arquivos
- **Compression.HuffmanTree** : Estruturas de dados e funções que para a implementação da codificação de Huffman
- **Compression.LZ77** : Estruturas de dados e funções que para a implementação do LZ77
- **Compression.SimplifiedDeflate** : Modulo que "une" os dois algorítmos de compressão.

Dificuldades, surpresas e desafios

Certamente uma das maiores **dificuldades** para implementar estes algorítmos (que conheço desde de sempre em termos imperativos) de uma maneira realmente funcional. E com realmente funcional quero dizer, se utilizando das vantagens que o paradigma nos dá ao permitir escrever códigos com um maior nível de abstração. Outro ponto que eu senti bastante dificuldade, foi lidar com leitura e escrita de binários, por ser uma linguagem de alto nível, "escovar" bits com Haskell se mostrou uma difícil tarefa.

Fiquei muito **supreso** como as abstrações com implementações de instancias de classes de tipo (como o **Monoid** implementado para o data type **HuffmanTree**) deixaram o código muito mais simples e legível (se comparado com as implementações em C por exemplo). Isso mostra o poder que o paradigma traz de escrever programas muito mais complexos, com menos linhas de códigos.

Entre os **destaques** que mais me "orgulharam" neste projeto, esta o uso pragmático do paradigma e linguagem ao resolver probelmas que originalmente seriam muito mais complexos de resolver por laços iterativos, mas foram resolvidos com alguns **folds** e **<>**. Por exemplo, a função que constroe a árvore de huffman:

```
buildHuffmanTree :: FrequencyArr -> HuffmanTree
buildHuffmanTree freq = foldl (<>) Empty leafArr
  where
    leafArr = map (uncurry Leaf) freq
```

A implementação em linguagens imperativas dessa função envolve criação de uma arvóre utilizando heap, um código que não seria nada amigável, e sertamente não possuíria somente 3 linhas!

Outro destaque importante do projeto, é uso de **QuickChecks** para validar uma propriedade excensial em qualquer algoritmo de compressão de arquivos, `decode.encode == id`

Uso

Para rodar a aplicação execute o seguinte comando

```
$ stack run [operation-mode] resources/<some-file-input>
```

Modos de operação

- **zip** : comprime o arquivo desejado, gerando dois arquivos de saída ("nome-do-arquivo".hzip, "nome-do-arquivo".hzip.meta)
- **unzip** : descomprime o arquivo, e printa o seu conteúdo decodificado na tela. O arquivo passado deve ser da extensão .**hzip**, com o arquivo correspondente de metadados na mesma página

---

Examplo

zip

```
$ stack run zip resources/teste.txt
```

unzip

```
$ stack run unzip resources/teste.hzip
```

Referências

https://www.ietf.org/rfc/rfc1951.txt

https://www.youtube.com/watch?v=Jqc418tQDkg&t=465s

https://www.youtube.com/watch?v=goOa3DGezUA&t=220s

https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-wusp/fb98aa28-5cd7-407f-8869-a6cef1ff1ccb

http://multimedia.ufp.pt/codecs/compressao-sem-perdas/codificacao-baseada-em-dicionarios/lz77/