

EP 03

Ronaldo Fumio Hashimoto

Data de Entrega: 19 de junho de 2010

Tries

1 Introdução

Em 1977 e 1978, Jacob Ziv e Abraham Lempel propuseram dois métodos de compressão de textos inovadores [1, 2]. Existem diversas variantes desses métodos implementados em programas conhecidos de compressão, como o *zip/unzip*, o *gzip*, o *compress* e outros compressores do UNIX.

2 Compressor de Ziv e Lempel

Neste EP estamos interessados no método que foi proposto em 1978, conhecido como LZ78. Ilustramos a técnica básica deste método por meio de um exemplo. Considere um alfabeto com apenas duas letras, 0 e 1, e um texto nesse alfabeto:

000101110001000000010011

A idéia é particionar o texto em pedaços de maneira que cada pedaço é o menor trecho que não apareceu entre os pedaços anteriormente definidos. O texto acima seria quebrado da seguinte maneira:

0|00|1|01|11|000|10|0000|001|0011|

O primeiro pedaço é sempre simplesmente o primeiro símbolo do texto. No caso do exemplo acima, como 0 é o primeiro símbolo e ele vem seguido de dois outros 0's, então o segundo pedaço é 00. O terceiro pedaço é apenas 1, já que o 1 nunca tinha aparecido no texto antes. E assim por diante. Por exemplo, o pedaço 0000 aparece pois 000 está entre os pedaços anteriores, porém não há entre os anteriores um pedaço 0000.

Todo compressor na verdade consiste de dois algoritmos: o compressor e o descompressor.

3 O Compressor

Para a compressão, determina-se os pedaços do texto conforme a regra descrita acima, e estes são numerados sequencialmente. No exemplo acima, a cadeia vazia (começo do texto) recebe o índice 0, o pedaço 0 recebe índice 1, o pedaço 00 recebe índice 2, e assim por diante.

Cada pedaço é então associado a um par, composto por um índice e um símbolo do alfabeto. Por exemplo, o pedaço indexado por 1, ou seja, a cadeia 0, é associada ao par 0 0, pois é a concatenação da cadeia vazia (indexada por 0) e da letra 0. O pedaço 00, que é indexado por 2, é associado ao par 1 0, pois é a concatenação do pedaço indexado por 1 (que é a cadeia 0) e o símbolo 0. Mais adiante, o pedaço 0000, por exemplo, é associado ao par 6 0, pois é a concatenação do pedaço indexado por 6 (cadeia 000) e o símbolo 0. Abaixo mostramos os pares associados a cada pedaço do exemplo acima.

pedaços →		0	00	1	01	11	000	10	0000	001	0011
índice →	0	1	2	3	4	5	6	7	8	9	10
pares →		0 0	1 0	0 1	1 1	3 1	2 0	3 0	6 0	2 1	9 1

Basicamente, a sequência de pares obtida é a codificação do texto (ou seja, a informação codificada)! Veja que, à medida que avançamos no texto, os pares representarão subsequências do texto cada vez mais longas. Inteiros relativamente pequenos (a primeira coordenada do par) substituirão uma longa sequência de caracteres. A sequência de pares é então codificada em um arquivo binário.

3.1 Representação binária da informação codificada

A informação codificada corresponde à sequência de pares obtida no passo anterior. Veremos nesta subseção uma forma de representar esta codificação.

Primeiramente queremos calcular o número de bits necessários para representar essa informação codificada, ou seja, a sequência de pares.

Na verdade cada par será representado por uma sequência de bits. Para isso, como adotamos o alfabeto binário, basta que a primeira coordenada do par, ou seja, o índice, seja escrito em binário. Para que a decodificação seja simples, o número de bits usados ao se escrever o índice do n -ésimo par em binário é o número de bits necessários para se escrever $n - 1$ (que é o maior valor que o índice do n -ésimo par pode assumir) em binário. Abaixo mostramos o número de bits usados para representar cada par do exemplo acima, bem como a sequência de bits da codificação de cada um desses pares.

índice →	0	1	2	3	4	5	6	7	8	9	10
pares →		0 0	1 0	0 1	1 1	3 1	2 0	3 0	6 0	2 1	9 1
# de bits →		1	2	3	3	4	4	4	4	5	5
binária →		0	10	001	011	0111	0100	0110	1100	00101	10011

Ou seja, a codificação desse trecho do texto ficaria

01000101101110100011011000010110011.

Se o texto de entrada for longo, ficará evidente a economia desse processo de codificação.

3.2 Estrutura de dados

Para implementar o algoritmo de codificação LZ78, você deve usar uma (variante da) estrutura de dados conhecida como *trie* (que se pronuncia como a palavra *try* em inglês, apesar do nome ser derivado da palavra *retrieval*).

No caso da nossa aplicação, a *trie* será uma árvore binária, pois adotamos o alfabeto binário, e a subárvore esquerda de cada nó corresponderá a um

bit 0, enquanto que a direita corresponderá a um bit 1. Cada nó estará associado a um dos pedaços do texto descritos na seção anterior, e será rotulado com o índice desse pedaço.

Veja a *trie* correspondente aos 6 primeiros pedaços no item (a) da Figura 3.2. A *trie* correspondente aos 11 símbolos do texto do nosso exemplo no item (b) desta figura.

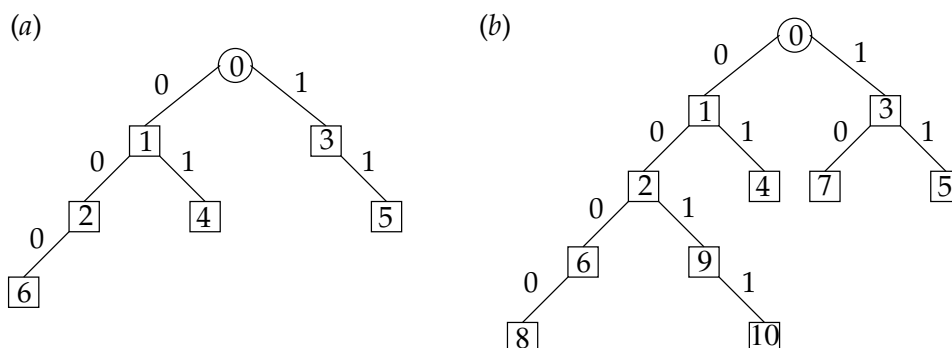


Figura 1: Trie com os (a) 6 primeiros pedaços e (b) os 11 símbolos do texto.

Observe que, percorrendo-se o caminho de um nó até a raiz, determina-se o pedaço cujo índice rotula o nó concatenando-se os “rótulos” das arestas atravessadas no **sentido inverso**, onde o rótulo de uma aresta é 0 ou 1 dependendo de se ela vai para a esquerda ou para a direita na árvore.

A **utilidade** dessa *trie* é determinar facilmente o próximo pedaço do texto (ou seja, descobrir o próximo pedaço que não apareceu no texto anteriormente – e por isso, ela deverá ser usada para encontrar a sequência de pares descrita anteriormente): basta percorrer a *trie* usando os próximos símbolos do texto para decidir qual dos ramos da *trie* seguir. Se chegarmos a um símbolo para o qual não existe um ramo correspondente, então esse é o último símbolo do próximo pedaço, e é fácil atualizar a *trie* para incorporar o novo pedaço. No nosso exemplo, percorrendo a *trie* a partir do 6º símbolo do texto, chega-se ao nó rotulado por 3 e não há nesse nó uma aresta para a esquerda. Portanto inserimos um novo nó (de rótulo 7) na *trie* como filho esquerdo deste. Repetindo o processo para os demais símbolos do texto, chegamos à *trie* exibida no item (b) da Figura 3.2.

A *trie* deve ser construída à medida que o texto é percorrido, ao mesmo tempo que se constroi a sequência de pares da codificação (ou diretamente

o arquivo binário resultante da compactação do texto).

4 O Descompressor

Uma vez que o número de bits para representar cada par na sequência codificada está claro e independe dos valores codificados, o descompressor consegue facilmente obter os pares a partir do arquivo binário. A partir dos pares, é fácil reconstruir a *trie* e por conseguinte obter o texto. O ideal é descomprimir o texto em uma única passada. Observe que a *trie* é uma maneira compacta de armazenar os pedaços do texto.

5 O que deve ser feito

Você deve escrever um programa que comprime e descomprime arquivos texto de acordo com o algoritmo LZ78. O nome do arquivo a ser comprimido/descomprimido deve ser dado na linha de comando. A opção `-x` indica que o arquivo dado deve ser descomprimido. A ausência dela indica que o arquivo dado deve ser comprimido. Ao comprimir um arquivo de nome `abacaxi.xxx`, seu programa deve gerar um arquivo de nome `abacaxi.xxx.cod`. Quando a opção `-x` é dada, o arquivo dado na linha de comando deve ter a extensão `.cod`, por exemplo, `abacaxi.xxx.cod`, e a saída do seu programa será um arquivo cujo nome substitui a extensão `.cod` por `.dec`. Ou seja, para o arquivo `abacaxi.xxx.cod`, o arquivo gerado com a opção `-x` seria `abacaxi.xxx.dec`. Repare que seu programa não deve destruir os arquivos dados como entrada, pois isso vai facilitar a correção do EP.

É preciso tomar um cuidado especial no final do texto a ser comprimido, do contrário o pedaço final do texto pode vir a ser um prefixo de um pedaço anterior. Falaremos mais sobre isso numa seção mais adiante.

Exemplo: O comando

```
ep3 carta.tex
```

deve gerar um arquivo de nome `carta.tex.cod`, enquanto que o comando

`ep3 -x carta.tex.cod`

deve gerar um arquivo chamado `carta.tex.dec`.

Para evitar problemas na codificação do pedaço final do texto, seu compressor automaticamente deve adicionar algumas cópias de um símbolo especial que não deve aparecer no texto (por exemplo, o símbolo fim de arquivo) ao final do arquivo. O número de cópias deve ser tal que o pedaço final seja um pedaço “normal”, ou seja, não seja um prefixo de um pedaço anterior. O decodificador deve agir de acordo, e interpretar que o texto terminou assim que decodificar um caracter de fim de arquivo (mesmo que a sequência de bits continue um pouco além deste).

Referências

- [1] J. Ziv and A. Lempel, “A universal algorithm for sequential data compression,” *IEEE Trans. Information Theory*, IT-23:337–343, 1977.
- [2] J. Ziv and A. Lempel, “Compression of individual sequences via variable-rate coding,” *IEEE Trans. Information Theory*, IT-24:530–536, 1978.