Algoritmos e Estruturas de Dados I
Prof. Alexandre Freire
Segundo Exercício Programa
Data de Entrega: 8 de Julho de 2018

Compressor de Ziv e Lempel

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

Neste EP estamos interessados no método proposto em 1978, que é conhecido como LZ78. Ilustramos a técnica básica deste método por meio de um exemplo. Considere um alfabeto com apenas dois símbolos (0 e 1) e o seguinte texto nesse alfabeto:

000101110001000000010011

Basicamente, a idéia é particionar o texto em pedaços de maneira que cada pedaço seja 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 0s, 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.

O método LZ78 é composto por dois algoritmos: o compressor e o descompressor.

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.

	0	00	1	01	11	000	10	0000	001	0011
0	1	2	3	4	5	6	7	8	9	10
	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! Veja que, a 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 decaracteres. A sequência de pares é então codificada em um arquivo binário.

Representação binária da informação codificada

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.

0	1	2	3	4	5	6	7	8	9	10
	0 0	1 0	0 1	1 1	3 1	2 0	3 0	6 0	2 1	9 1
	1	2	3	3	4	4	4	4	5	5
	0	10	001	011	0111	0100	0110	1100	00101	10011

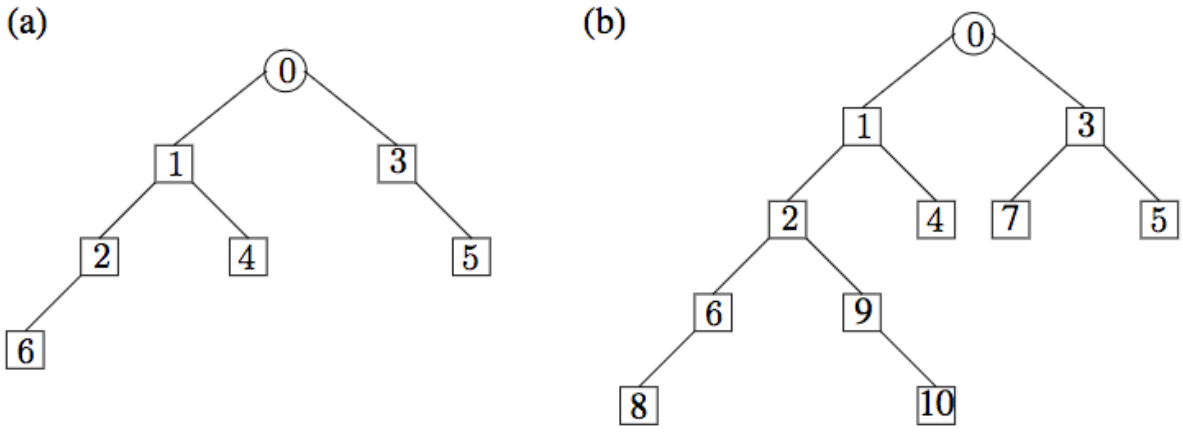
Ou seja, a codificação desse trecho do texto ficaria assim:

0100010110111010001101100010110011

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

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*). Uma *trie* é uma árvore (não necessariamente binária) onde, em cada subárvore, estão armazenadas as chaves que têm um certo "prefixo" comum. 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 11 primeiros símbolos do texto do nosso exemplo (os 6 primeiros pedaços) no item (a) da figura abaixo.



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 se ela vai para a esquerda ou para a direita na árvore. A partir dessa *trie*, é fácil determinar o próximo pedaço do texto: 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 12º 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 acima.

A trie deve ser construída à medida que o texto é percorrido, ao mesmo tempo que se constrói a sequência de pares da codifição (ou diretamente o arquivo binário resultante da compactação do texto).

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.

O descompressor

Uma vez que o número de bits para representar cada par na sequência codificada está claro e independe dos valores codificados, o decompressor 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. Como um teste, veja se você consegue decodificar o seguinte texto codificado usando o esquema acima. Após decodificá-lo, interprete cada 8 bits da sequência como o código US ASCII de um caracter e leia a mensagem até o símbolo $ aparecer, que indica que a mensagem terminou.

```
00101011110000011010101001000001110100110011011001110110101010110010111001100
10010001100001001010101101101001000110100000101100100110011011000001110011101
01001011000000000110110000110101111001111110010000111000010000001010100
```

A mensagem acima foi codificada manualmente, e pode estar sujeita a erros...

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.

Exemplo:

O comando `java EP2 carta.tex` deve gerar um arquivo de nome `carta.tex.cod`, enquanto que o comando `java EP2 -x carta.tex.cod` deve gerar um arquivo chamado `carta.tex.dec`. Lembre-se de que para obter os parâmetros passados em linha de comando, basta acessar o vetor de strings declarado como parâmetro do método `public static void main (String[] args)`, onde `args[0]` contém o primeiro parâmetro, `args[1]` o segundo, e assim por diante...

Atenção: 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).