TP0: LZ77

Giovani Amaral Barcelos

6 de abril de 2016

1 Introdução

Esse trabalho prático objetiva a compressão e descompressão de arquivos utilizando o algoritimo LZ77, desenvolvido por Abraham Lempel e Jacob Ziv em 1977.

No nosso dia a dia compartilhamos muitos arquivos via web, e isso tem um custo de banda para transferência. Se os arquivos podem ter o mesmo conteúdo em um tamanho menor, isso deve ser feito para economizar banda e armazenamento.

Para entender o algoritimo e construir o compressor e o descompressor, deve-se ter um entendimento de algorítmos de busca e manipulação de bits, para codificar corretamente a saída.

Esse método de compressão dá-se tão somente pela substituição conjuntos de 3 ou mais bytes por ponteiros que apontam para a maior ocorrência anterior dos respectivos conjuntos. Analogamente, na descompressão, lemos os bytes comprimidos e re-escrevemos os ponteiros substituindo-os pelas ocorrencias para as quais eles apontam.

2 Solução do problema

2.1 Compressor

Temos um vetor contendo todos os bytes lidos do arquivo original. Andamos pelo vetor através de conjuntos de 3, e ao encontrar ocorrências anteriores destes conjuntos, escrevemos uma referência que aponta para a maior ocorrência possível. Se não encontramos uma ocorrência para um determinado conjunto, escrevemos o primeiro byte dele na forma de um literal. O fluxo principal do compressor pode ser aproximado pelo seguinte pseudo código:

Código 1: Fluxo principal compressor

```
1 para cada conjunto de 3 bytes no arquivo original:
2     para cada ocorrencia anterior deste conjunto:
3         tentamos expandir para os caracteres seguintes a ocorrencia e ao conjunto
4                         escrevemos um ponteiro para a da maior ocorrencia
5     se nao tiver nenhuma ocorrencia anterior:
6                         escrevemos o primeiro caracter literal deste conjunto
```

Para realizar as buscas de ocorrências dos trios no vetor de bytes, foi usado o algorítmo KMP adaptado do livro "Introduction to Algorithms"(Cormen, Leiserson, Rivest and Stein). No algoritmo adptado, retornamos um array com todas as posições de matching, e só é considerado um match aquela ocorrência que já foi avaliada como um conjunto anteriomente, através de um array de flags. Isso pode ser melhor explicado no seguinte código:

Código 2: KMP Busca em strings

```
ao encontrar uma ocorrencia anterior
                        se esta ocorrencia foi avaliada anteriormente como um conjunto
                                se a distancia ate essa ocorrencia e menor ou igual a 3278
                                        colocamos a posicao atual no vetor de posicoes de ocorrencia
```

A grande dificuldade da solução está na manipulação dos bits para escrever no arquivo final, pois os bytes não estão juntos. Podemos ter os dois primeiros bits em um byte e o resto no byte à direita. Isso ocorre pois os literais não são representados com 1 byte e sim 9 bits, sendo este a mais o identificador que nos mostra se aquele byte seguinte é um literal ou um ponteiro. No caso dos ponteiros, além do identificados temo 1 byte significando o comprimento e 15 bits representando o offset.

Desta forma, deve-se trabalhar o tempo todo com o shift de bits, para que a codificação de saída seja idêntica à especificada pelo documento.

2.2 Descompressor

Já o fluxo do descompressor é tangenciado pelo seguinte código:

Código 3: Fluxo principal descompressor

```
para cada bit lido do arquivo:
        se for igual a 0:
                escrevemos o byte imediatamente posterior no arquivo de saida
        se for igual a 1:
                traduzimos o byte posterior para a variavel de comprimento, os proximos 15 bits para
                     o offset e escrevemos na saida o valor para o qual este ponteiro aponta
```

Já tendo a estrutura e a prática de manipulação de bits, o descompressor fica bem mais simples, uma vez que é só a interpretação de decodificação de bytes sobre os quais sabemos a organização.

3 Análise de complexidade

Para as seguintes análises, vamos considerar T o tamanho do arquivo em bytes, N o tamanho do texto e M número máximo de combinações possíveis para um dado match.

3.1 Análise de complexidade de tempo

void prefixo(unsigned char* p, int m, int *pi) - Função que cálcula os prefixos para um padrão. Complexidade O(m) onde m é o tamanho do padrão, que no nosso caso é sempre 3, ou seja, **O(1)**.

int kmp(unsigned char* t, int n, unsigned char* p, int m, int* posicoes, int* byteFlags) - Esta função percorre nossa string de busca, tendo a complexidade O(n), onde n é o tamanho do texto. Além disso ela chama a função acima, então sua complexidade total é O(n + 3), ou seja **O(n)**.

int lz77_compress(char* source, char* target) - Função responsável pela compressão do arquivo. Temos um for que vai de 3 até o tamanho em bytes, então vamos chamar **T** o tamanho em bytes de um arquivo. Dentro deste for, temos um outro que checa cada posição de matching e tenta expandi-las com um while. Seja **M** o número máximo de matchings possíveis para um determinamo conjunto de bytes a ser validado. Como sabemos que a maior expansão possível é 255, temos a certeza que nosso while rodará no máximo 255 vezes. Juntando as complexidades acima, temos ainda nosso algoritmo KMP, que receberá, na última iteração e no pior caso, um texto de tamanho 32768. Vamos denotar o produtório de todas as iterações por por K.

$$S = \prod_{i=1}^{32768} i$$

Sabemos que K é a mesma coisa que o fatorial de 32768. Logo, a complexidade do nosso compressor é O(T*M*255*32768!) que equivale a **O(T*M)**.

int lz77_decompress(char* source, char* target) - Temos um laço while que irá percorrer o arquivo interpretando-o até sobrarem 8 bytes ou menos. Entretanto, nosso índice I não caminha de 1 em 1, mas sim pode sofrer pulos de 9 ou 24, que são as quantidades de bits que podem ser escritas e, no pior caso de complexidade, ele saltará de 9 em 9.
Dentro deste while, temos um laço for que no pior caso irá até o comprimento máximo de uma ocorrencia, ou seja, 258. Logo, nossa complexidade é O((T/9)*258) que equivale a dizer **O(T)**.

3.2 Análise de complexidade de espaço

3.2.1 Compressor

Tanto nosso vetor que conterá os bytes de saída quanto os vetor de flags para os bytes serão alocados com o tamanho de bytes do arquivo original, pois é o pior caso, onde não teremos nenhuma compressão e todos os bytes serão literais.
Dito isso, nosso kmp irá alocar um vetor para o padrão de 3 para cada vez que for executado. Então nosso complexidade é O(2T*3) ou **O(T)**.
Quanto ao **throughput**, fica claro que para arquivos maiores o tempo de compressão é muito maior, sendo o compressor mais indicado para arquivos menores, como arquivos de texto, e deve ser evitado para imagens e vídeos.

3.2.2 Descompressor

No nosso descompressor temos um vetor para guardar todos os bytes lidos do arquivo original (Tamanho T) e um que irá armazenar os bytes da saída. Nosso vetor de saída dobra de tamanho sempre que atinge o tamanho T. Como visto em sala anteriomente, sabemos que essa ideia de dobrar quando chegar no limite representar o custo de O(1) amortizado, então nossa complexidade é **O(T)**.

4 Análise experimental

Quanta a taxa de compressão: Para um arquivo de texto com 55kB, a saída foi 30kB, uma compressão de 45%. Para uma imagem de 2mB, a saída de compressão foi 1.7mB, logo a taxa de compressão foi de apenas 15%. Para uma outra imagem de 187kB, a saída foi de 210kB, logo tivemos um **aumento** de 12%, ficando com 112% do arquivo original.

Quanto ao tempo de execução:

Tamanho do arquivo	Tempo execução
1.1kB	0.002s
10.5kB	0.076s
356kB	1m5.739s

Quanto a complexidade de espaço e bytes alocados:

Tamanho do arquivo	Bytes alocados
1.1kB	141,882
10.5kB	218,004
356kB	2,745,954

5 EXTRA: Melhoria no algoritmo

Alguns arquivos não podem ser comprimidos, e devido ao fato que o LZ77 adiciona um bit identificador, ele acabam ficando muito maiores do que o arquivo original. Desta forma, para evitar ficar com um arquivo maior, poderíamos implementar uma alteração que colocaria o byte 00000000 no começo do algorítmo, aí saberíamos que todos os bytes seguintes são literais, sem o indicador. Desta forma, nosso arquivo aumentaria apenas em 1 byte caso não pudesse ser compactado.

6 Conclusão

A grande dificuldade deste trabalho foi lidar com arquitetura de baixo nível, pois eu não possuía muita experiência com operações bit a bit. Escolhi o algoritmo KMP do Cormen por ser um algoritmo relativamente fácil de entender e simples de implementar, além de ser uma escolha rápida.
Meu programa, entretanto, apresentou comportamento inesperado para arquivos muito grandes, onde ele demorou muito tempo para terminar a execução, e em algumas das execuções finalizou corretamente, em outras finalizou apresentando SIGNAL ABORT porém comprimindo corretamente.
Em alguns dos casos, ainda, pode-se perceber que o arquivo **aumentou** de tamanho. Isso se deu pois não foi possível a criação de nenhum ponteiro, portanto foram escritos apenas literais, e como nosso algoritmo escreve um bit identificador a mais para cada byte, é esperado que o tamanho do arquivo de saída seja maior que o de entrada. O compressor é mais indicado para arquivos pequenos, como arquivos de texto, pois a execução é rápida e o resultado eficiente.

7 Referencias

T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, 3rd edition, MIT Press, 2009