Trabalho Prático 0: LZ77

Alef Henrique de Castro Monteiro

6 de abril de 2016

1 Introdução

O trabalho prático 0 tem por objetivo a criação de um compactador e de um descompactador de arquivos. A implementação dos dois programas deve tomar por base o LZ77, algoritmo de compressão de dados criado por Abraham Lempel e Jacob Ziv em 1977. Esse algoritmo realiza a compressão dos dados por meio da redução de redundâncias, ou seja, ele substitui ocorrências de uma sequência de bytes por referências à primeira ocorrência dessa sequência.

Para a realização da tarefa, é necessário a implementação de algoritmos que realizam casamento de strings, ou seja, algoritmos que, dado um padrão, identifica as ocorrências deste padrão no texto. No caso deste trabalho, o algoritmo usado para essa finalidade foi o KMP(Knuth-Morris-Pratt)[1].Além de lidar com o casamento de padrões, o trabalho prático exige habilidades para manipular uma sequência de bits de um grande vetor, o que vem a ser o grande desafio desse trabalho.

O problema de compactar e de descompactar arquivos é extremamente comum no universo da computação. O algoritmo LZ77, especificamente, é um modelo clássico, além de ser utilizado em formatos como GIF, PNG e GZIP. Atualmente, a compressão de dados está presente em diversos meios, e a implementação de um compactador torna-se um desafio com imensa aplicabilidade no mundo real.

2 Solução do Problema

2.1 Compressor

No caso do compactador, a solução do problema envolve 3 grandes etapas: armazenar o conteúdo do arquivo em uma string, utilizar o algoritmo LZ77 para identificar as múltiplas ocorrências de uma cadeia e compactar o arquivo e, por fim, imprimir a string resultante no arquivo de saída. A primeira etapa foi feita de forma trivial, com a utilização de funções comuns na linguagem. O funcionamento geral do programa segue o seguinte pseudo-código:

Código 1: LZ77

```
1  Enquanto a string inicial nao tiver sido lida por completo:
2          Colocar os 3 proximos caracteres na CS
3          Atualizar o valor do SB com os caracteres CS[1] e C[2]
4      Se a CS tiver tamanho 3:
5          Buscar uma ocorrencia anterior da sequencia no SB, com o algoritmo KMP.
6          Se houve matching:
7          verifica o matching de maior comprimento
8          Imprime o ponteiro relativo ao matching com a funcao EmpurraBitPonteiro
9      Se nao houve matching:
10         Imprime o primeiro caractere da CS com a funcao EmpurraBitLiteral
11 Imprime String final no arquivo de saida
```

O algoritmo KMP utilizado pelo LZ77 teve algumas modificações, para que fossem encontrados apenas os "matchings"válidos, ou seja, os "matchings"em que os 3 caracteres do texto foram Current Sunbstring(CS) juntos. O algoritmo modificado foi implementado da seguinte forma:

Código 2: KMP(CS, SB)

```
Prefixo = FUNCAO PREFIXO
n=SB.comprimento
m=CS.comprimento
posicoes= vetor com as posicoes iniciais de cada matching encontrado
j=0
q=0
for i=1 to n
    while q>0 e CS[q+1]!=SB[i]  //caractere do padrao e diferente do proximo caractere do texto
        q= Prefixo[q]
    if CS[q+1]==SB[i]                   //Caractere do padrao e igual ao caractere do texto
        q++
    if q==m                                             //O padrao todo foi encontrado
        if SB[i-2].flag==1              //Verificar se o matching e valido
                posicoes[j]= i-2        //Armazenar a posicao onde ocorreu o matching
                j++
        q= Prefixo[q]                   //Procurar pela proxima ocorrencia do padrao no texto
```

Apos o realizar algortimo KMP, a função obtém um vetor com todas as posições onde ocorreram *matching*. Dessa forma, é possível verificar cada um dos "matchings"e encontrar aquele com o maior comprimento, ou seja, aquele que será convertido em um ponteiro.

Além do algoritmo KMP, o LZ77 utiliza duas funções: *EmpurraBitLiteral* e *EmpurraBitPonteiro*. Essas funções são essenciais para o funcionamento do programa, pois elas realizam a manipuação dos bits, de forma que um caractere seja impresso com 9 bits e um ponteiro seja impresso com 24 bits. As duas funções são descritas a seguir.

Código 3: EmpurraBitLiteral

```
Verifica o byte do vetor que deve ser escrito
Verifica o numero de bits empurrados no byte
Coloca o 0 na proxima posicao
Imprime os bits iniciais do literal apos o zero
Imprime o restante dos bits do literal no proximo byte
Atualiza o numero de bits empurrados
Atualiza o byte do vetor em que parou de escrever os bits
```

Código 4: EmpurraBitPonteiro

```
Verifica o byte do vetor que deve ser escrito
Verifica o numero de bits empurrados no byte
Verifica os valores do comprimento e do r_offset do ponteiro
Coloca o 1 na proxima posicao
Imprime os bits iniciais do comprimento apos o 1
Imprime o restante dos bits do comprimento no proximo byte
Imprime os bits iniciais do offset apos o comprimento
Se restarem menos de 8 bits do offset para serem escritos:
        Imprime o restante dos bits do offset no proximo byte
Se restarem mais de 8 bits:
        Imprime os 8 proximos bits no proximo byte
        Imprime o restante dos bits do offset no byte seguinte
Atualiza o byte do vetor em que parou de escrever os bits
```

Quando o KMP não encontra nenhuma ocorrência da Current Substring(CS), o programa utiliza a função *EmpurraBitLiteral* e escreve o primeiro caractere da CS no vetor de saída como um literal(9 bits). Por outro lado, se houve *matching*, o programa verifica o comprimento e o offset do *matching* e utiliza a função *EmpurraBitPonteiro* para imprimir os 24 bits do ponteiro no vetor de saída.

O programa realiza todo esse procedimento até que a "janela"da Current Substring percorra todo o vetor que detém o conteúdo da entrada.

2.2 Descompressor

O Descompressor, por sua vez, transfere o conteúdo do arquivo comprimido para um vetor e percorre todo o vetor identificando os bits que representam ponteiros e os bits que representam literais. O descompressor segue o seguite fluxo:

Código 5: DESCOMPRESSOR

```
1  Enquanto o vetor de entrada nao tiver sido lido por completo:
2          Verifica o proximo bit do vetor
3      Se o bit e 0:
4          Chama a funcao DesempurraBitLiteral, que retira o 0 e imprime os proximos 8 bits no vetor de
               saida
5      Se o bit e 1:
6          Chama a funcao DesempurraBitPonteiro, que retira o 1 e verifica os valores do comprimento e
               do offset do ponteiro
7          Verifica a string que o ponteiro corresponde
8          Imprime a string no vetor de saida
9  Imprime String final no arquivo de saida
```

Para realizar a descompressão dos dados, o programa utiliza as funções DesempurraBitLiteral e DesempurraBitPonteiro. Essas funções fazem a manipulação dos bits, de forma que o programa possa identificar os literais e os ponteiros. O funcionamento das funções é mostrado a seguir.

Código 6: DesempurraBitLiteral

```
1  Verifica o byte do vetor que deve ser escrito
2  Verifica o numero de bits empurrados no byte
3  Retira o 0 e identifica os 8 bits do literal
4  Imprime os bits iniciais do literal na proxima posicao do byte
5  Imprime o restante dos bits do literal no proximo byte
6  Atualiza o numero de bits empurrados
7  Atualiza o byte do vetor em que parou de ler os bits
```

Código 7: DesempurraBitPonteiro

```
1  Verifica o byte do vetor que deve ser escrito
2  Verifica o numero de bits empurrados no byte
3  Retira o 1 e identifica os 8 bits do comprimento
4  Segue para o proximo byte e identifica os bits iniciais do r_offset
5  Vai mais um byte adiante e identifica os bits finais do r_offset
6  Atualiza o byte do vetor em que ler de escrever os bits
7  Retorna os valores do comprimento e do r_offset do ponteiro.
```

Assim, o descompressor substitui os ponteiros pelas strings correspondentes e imprime a string gerada após a descompressão no arquivo de saída, obtendo o arquivo original.

3 Análise de Complexidade

O trabalho prático é composto dos arquivos compress.c, decompress.c e mais 3 TAD's: Lz77, KMP e binário. Nesta análise, cada um dos tipos abstratos de dados(TAD's) será avaliado separadamente e, por fim, serão analisados os programas principais. Durante as análises a seguir, $\boldsymbol{N}$ é o número de bytes do arquivo de entrada, $\boldsymbol{T}$ é o tamanho em bytes do SB e $\boldsymbol{M}$ é o número total de *matchings* encontrados pelo algoritmo KMP.

3.1 TAD KMP

int* prefixo(int* prefix, unsigned char* padrao, int tam-padrao) - A função depende apenas do tamanho do padrão. Como o padrão tem tamanho único igual a 3, o custo da função é **O(1)**.

int KMP(int* posicoes, int* prefix, unsigned char* padrao, buffer* texto, int tam-padrao, int tam-texto, int SB-posicao) - A função KMP chama a função prefixo- O(1). Além disso, KMP contém um loop principal, que é executado de acordo com o tamanho do texto contido no SB - O(T). Dentro deste loop existem apenas operações O(1). Portanto, o custo da função KMP é **O(T)**.

3.2 TAD Binário

void empurra-bit-literal(unsigned char* saida, int* bytes-pos, unsigned char literal, int* bits-empurrados) - A função tem apenas operações O(1) e, portanto, tem custo final **O(1)**.

void empurra-bit-ponteiro(unsigned char* saida, int* bytes-pos, unsigned char comprimento, unsigned short r-offset, int* bits-empurrados)- A função tem apenas operações O(1) e, portanto, tem custo final **O(1)**.

void desempurra-bit-literal(unsigned char* saida, int* bytes-pos, unsigned char* string, int* string-pos, int* bits-empurrados) - A função tem apenas operações O(1) e, portanto, tem custo final **O(1)**.

void desempurra-bit-ponteiro(unsigned char* comprimento, unsigned short* r-offset, unsigned char* string, int* string-pos, int* bits-empurrados) - A função tem apenas operações O(1) e, portanto, tem custo final **O(1)**.

3.3 TAD LZ77

void LZ77(int* posicoes, int* prefix, unsigned char *CS, unsigned char *string, buffer *SB, int* SB-posicao, FILE* output, int string-length, unsigned char* saida, int* bytes-pos) - Além das operações de custo constante O(1), a função tem um loop principal que depende apenas do tamanho em bytes da entrada- O(N). Dentro deste loop existem diversas operações de custo constante O(1). Além disso, no pior caso, a função chama o algoritmo LZ77- O(T) e realiza um loop que depende do número de mathcings encontrados no texto- O(M). Portanto, o custo final da função é O(N*(T+M)), que equivale a **O(N)**, já que (T+M) tem valor máximo igual a (32768+258).

void descompactador-LZ77(unsigned char *string, int string-length, unsigned char* saida, int* bytes-pos, FILE* output) - A função descompactador realiza apenas um loop principal que depende do tamanho do arquivo de entrada- O(N). Dentro deste loop existem apenas operações de custo constante O(1). Portanto, o custo total da função é **O(N)**.

3.4 MAIN

Compactador(compress.c) - A função main do compactador realiza diversas operações de custo constante O(1), um loop que depende do tamanho do arquivo de entrada-O(N) e chama a função LZ77- O(N). Portanto, a complexidade final do compactador é O(N+N), que equivale a **O(N)**.

Descompactador(decompress.c) - A função main do descompactador, por sua vez, realiza diversas operações de custo constante O(1) e chama a função descompactador-LZ77 - O(N). Portanto, a complexidade final do compactador é **O(N)**.

3.5 Análise de complexidade do espaço

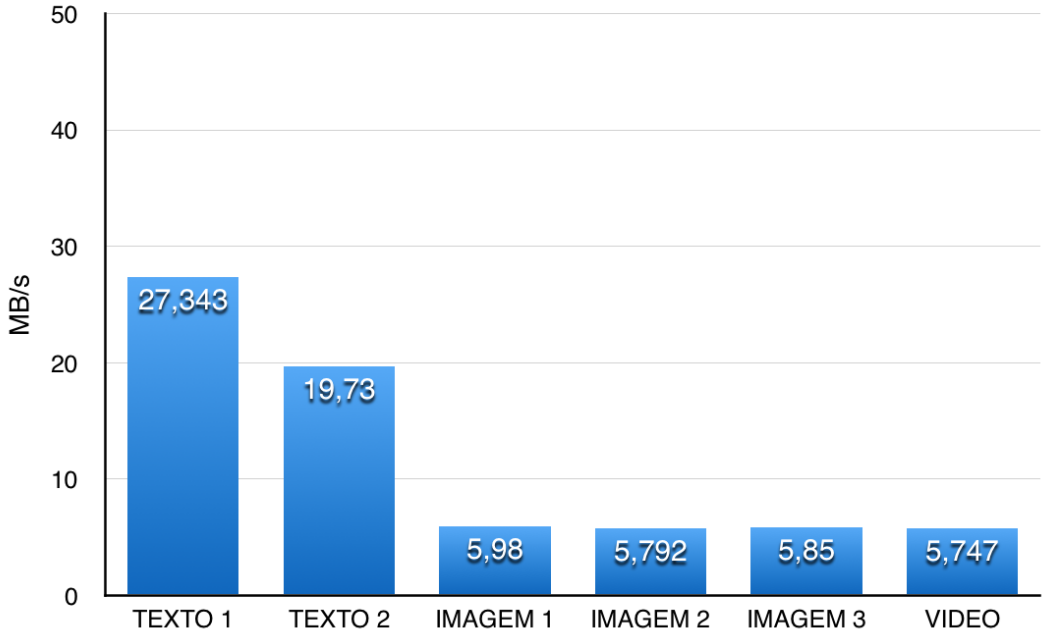
Compactador - Além das alocações de memória de custo constante- O(1), o compactador aloca memória para os vetores "string"e "saida", que armazanam o conteúdo da entrada e da saída, respectivamente. Cada um dos vetores é alocado com o dobro do tamanho do arquivo de entrada. Portanto, o custo do espaço no compactador é **O(N)**.

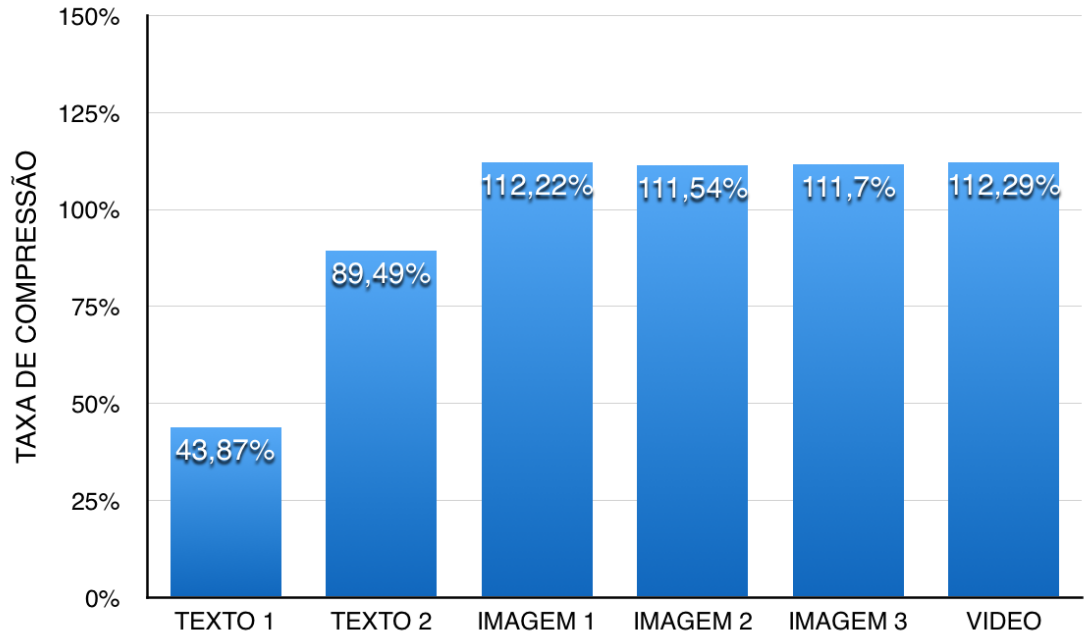
Descompactador - O descompactador realiza duas alocações de memória, uma no vetor "string", que recebe o conteúdo da entrada, e outra no vetor "saida", que armazena o conteúdo que será impresso no arquivo de saída. O vetor string tem alocação com custo igual a O(N+258). Já o vetor de saída tem alocação de 24 vezes o tamanho da entrada, uma vez que o pior caso teria um arquivo com 3 literais e (N-3) ponteiros. Portanto, o custo final do descompactador é O(24*N), equivalente a **O(N)**.

4 Análise Experimental

Com a finalidade de realizar um estudo experimental a respeito da taxa de compressão, foram realizados 6 testes. Nos 6 testes foram utilizados 1 vídeo(*.m4v*), 3 imagens(*.jpg*) e 2 textos(*.txt*). Os detalhes do teste e os resultados estão apresentados na tabela e nos gráficos a seguir.

	TEXTO 1	TEXTO 2	IMAGEM 1	IMAGEM 2	IMAGEM 3	VIDEO
TAMANHO (bytes)	196	990	187.761	1.441.469	1.106.530	2.343.002
TAMANHO APÓS COMPRESSÃO	86	886	210.715	1.607.877	1.235.997	2.631.009
TEMPO (segundos)	0,007	0,049	30,658	243,004	184,713	398,122
MB/s	27,343	19,730	5,980	5,792	5,850	5,747
TAXA COMPRESSÃO	43,87%	89,49%	112,22%	111,54%	111,70%	112,29%





A partir dos dados obtidos com o experimento, pode-se confirmar o resultado obtido na análise de complexidade do compressor, onde foi observado que o tempo de execução é O(N). Além de verificar a complexidade linear do tempo, fica evidente no experimento que o tamanho do arquivo afeta o *throughput* de compressão (MB/s). De acordo com os experimentos, quanto maior é o tamanho do arquivo, menor é *throughput* de compressão.

Em relação à taxa de compressão, o resultado obtido foi surpreendente. Ao analisar o tamanho dos arquivos de teste, o comportamento da taxa de compressão é exponencial: quanto menor um arquivo, maior é a difereça que 1 byte pode fazer na taxa de compressão. De acordo com os experimentos, a taxa de compressão para os arquivos maiores tende a ficar próxima de 112%. Paralelamente, ao analisar o tipo dos arquivos de teste, percebe-se que o compactador tem um comportamento melhor em arquivos de texto, enquanto o comportamento em arquivos de imagem e vídeo é pior.Portanto, conclui-se que o algoritmo de compressão gerado tende a ser bom para arquivos pequenos e/ou arquivos de texto.

5 Extra - Sugestão de Mundança

A grande desvantagem apresentada pelo algoritmo implementado neste trabalho é que ele pode, nos piores casos, aumentar o tamanho dos arquivos. De acordo com os experimentos realizados neste trabalho prático, arquivos grandes (maiores que 1Mb) e arquivos que não são de texto tendem a aumentar o seu tamanho com a execução do compressor. Isso ocorre porque, caso um arquivo não tenha nenhum subsequência repetida, o compressor aumenta em um bit cada caractere do arquivo. Outro fato observado é que os ponteiros que representam um matching de comprimento 3, além de serem os ponteiros mais frequentes, eles tem os mesmos 24 bits que os 3 caracteres representados pelo ponteiro. Ou seja, esses ponteiros não diminuem o tamanho da sequência.

Dessa forma, um novo método para diferenciar ponteiros e caracteres seria interessante. Outro fator que poderia ser alterado era permitir que os ponteiros tenham tamanho flexível. Assim, um ponteiro que representa um matching de comprimento 3 não precisaria de 8 bits para representar seu comprimento, mas apenas 2 bits. Com a flexibilidade no tamanho dos ponteiros e com literais representados por 8 bits apenas, a eficiência do programa iria aumentar drasticamente, além de evitar as taxas de compressão maiores que 100% .

6 Conclusão

A implementação de um compressor de dados em C é uma tarefa simples, porém desgastante. O grande desafio trazido pelo trabalho é a manipulacão de bits, essencial para execução do trabalho. Essa manipulação deve ser feita cuidadosamente, uma vez que existem muitas situações específicas para lidar. Com a compreensão do funcionamento dos bits, após a implementação do compacctador, a realização do descompressor fica mais fácil.

Os resultados obtidos pelos experimentos mostram que o compressor baseado no algoritmo LZ77 funciona bem, porém não é eficiente em alguns casos. A conclusão dos testes é que o algoritmo é muito bom para arquivos de texto. No entanto, em imagens e vídeos, que costumam já ser comprimidos, o resultado foi péssimo: os arquivos aumentam o seu tamanho.

Por fim, a execução do Trabalho Prático 0 é de grande valia para o entendimento de operações com bits. Além disso, o trabalho permite uma maior compreensão dos algoritmos que realizam "casamento de padrões", como o KMP, por exemplo.

Referências

[1] Thomas H Cormen. *Introduction to algorithms*. MIT press, 2009.