TP0: LZ77

Vitor Cláudio Chaves de Aguiar

6 de abril de 2016

1 Introdução

O trabalho se baseia na implementação de um algorítimo de compressão de dados, mais especificamente o LZ77. É esperado que ao final desse trabalho seja possível compactar e descompactar quaisquer arquivos desejados.

O motivacional observado nesse trabalho é o quanto pode ser útil esse tipo de algoritmo. Muitas vezes é necessário diminuir um tamanho de um arquivo com a finalidade de economizar espaço em disco ou para enviar um e-mail sem que o tamanho máximo do servidor de e-mail seja ultrapassado ou para uma transferência de arquivos mais rápida. Outro fator interessante é que esse algoritmo não causa perda de dados no processo.

O algoritmo funciona de tal maneira que todas as redundâncias são buscadas no arquivo e representadas de uma forma diferente, desta forma o tamanho do arquivo final é reduzido. É necessário entender no algoritmo que ele procura em uma janela de opções de caracteres aqueles que são identificados como redundância, ou seja, que se repetem.

A solução do trabalho está em entender como as redundâncias são identificadas e selecionadas para fazerem parte da saída final. Tendo em mãos essa informação é necessário saber representar esses dados da forma especificada.

2 Solução do Problema

A solução do problema pode ser dividido em duas etapas. A primeira consiste em olhar no texto todas as possíveis redundâncias com o seu comprimento e offset. Essa busca é realizada à partir do `CS(Current Substring)` que seria o padrão a ser buscado no texto e que é modificado a todo o momento. Tomadas de decisões são necessárias à partir do momento que temos várias ocorrências. Se as ocorrências tiverem um comprimento diferente uma das outras, a que tiver maior comprimento deve ser selecionada. Caso contrário, a ocorrência mais recente (cujo offset é menor) deve ser selecionada. A verificação das redundâncias foi realizada com o algoritmo KMP.

A segunda parte implica em como os valores literais e os ponteiros vão ser representados e escritos no arquivo de saída.Para adaptar esses valores para serem escritos na forma binária foi necessário fazer diversas operações com bits, bytes e o vetor de saída. Dentre as operações utilizadas estão: operações lógicas, mod, divisão e shift.

Código 1: Primeira parte

```
1 para cada CS novo gerado:
2     para cada CS com tamanho 3:
3         executa o algoritmo KMP para buscar o padrao no SB
4         seleciona o match mais adequado
5         adiciona o match no vetor de saida com a representacao de um ponteiro
6         se nao retornar nenhum match do KMP:
7                 adiciona o dado no vetor de saida com a representacao de um literal
8     para cada CS com tamanho menor que 3:
9         adiciona os valores dele no vetor de saida com a representacao de um literal
```

Código 2: Segunda parte

```
para cada literal:
        coloca o bit identificador 0
        adiciona o caracter na posicao correta no byte
para cada ponteiro:
        coloca o bit identificador 1
        adiciona o caracter do comprimento na posicao correta no byte
    adiciona o caracter do offset na posicao correta no byte
```

Código 3: KMP

```
calcula o vetor de prefixos
i <- 0
q <- -1
countMatch <- 0
for i ate n-1
        while q > -1 e CS[q+1] != SB[i]
        q <- vetorPrefixos[q]
    if CS[q+1] = SB[i]
        q <- q + 1
    if q = m-1
        if indicesFalg[i-q] == 1
                indicesMatch[countMatch] = i-q;
                        countMatch <- countMatch + 1;
        q <- vetorPrefixos[q]
return countMatch
```

O algoritmo do KMP teve como base o pseudocódigo do livro do Cormen. Porém foi adaptado para que os matchs ocorram apenas em conjuntos que foram CS. Para verificar isso, toda vez que um conjunto de caracteres vira CS um vetor de flags coloca uma flag na mesma posição onde se encontra o primeiro caracter do CS.

3 Análise de Complexidade

3.1 Tempo

A função aonde tudo acontece é na `comprimirEntrada`.

Esta função precisa percorrer todo o vetor de entrada para fazer todas as operações necessárias, diante disso, o loop mais externo na linha 58 do LZ77.c possui complexidade $O(n)$ sendo n o tamanho da entrada. Entretanto, em cada iteração desse loop mais externo, chama-se a função `KMP matcher`.

A função `KMP matcher` depende do tamanho do SB. Na minha implementação o SB aumenta cada vez que um CS foi processado e trocado os seus valores. Cada iteração do loop mais externo coloca no final novos valores pra CS e altera o SB. Portanto, o loop mais externo com essa função deixam a complexidade na ordem de $O(n^2)$.

Existem mais dois loops que possuem a finalidade de verificar o match mais adequado perante o seu comprimento e offset. Considerando o pior caso do mais externo na linha 93 do LZ77.c, ele não entraria no mais interno na linha 100 do LZ77.c e, portanto, apresentaria no máximo 256 iterações que é a capacidade de representação com 8 bits, tornando a ordem desse loop constante $O(1)$. O mesmo vale pro pior caso do mais interno.

Pior caso loop linha 93 do LZ77.c: todos os matches devem possuir apenas comprimento 3. Para isso ocorrer eles não podem estar se tangenciando, diante disso é necessário um caracter diferente entre cada um deles. Por exemplo: abc**D**abc**E**abc**F**abc**G**abc**H** Desta forma um match irá ocorrer a cada 4 bytes. Ou seja, um total de 256 matches

É possível observar que todas as outras instruções e funções tem ordem constante e, portanto, a ordem geral do programa é baseado no loop na linha 58 do LZ77.c com o `KMP matcher`. Ou seja, complexidade de tempo final é $O(n^2)$.

3.2 Espaço

Toda alocação de memória no programa ou é realizada por um constante ou pelo tamanho da entrada. Ou seja, o espaço ocupado vai ser diretamente ligado ao tamanho do arquivo/entrada. Permitindo a conclusão da análise de complexidade final de espaço ser $O(n)$.

4 Análise Experimental

Podemos fazer uma análise experimental apresentando as taxas de compressão

Arquivos muitos dados repetidos

Exemplo 1: entrada com tamanho de 9,05KB gerou saída de 2,32KB (redução de 90,2% do espaço ocupado)

Exemplo 2: entrada com tamanho de 42,2KB gerou saída de 4,13KB (redução de 74,4% do espaço ocupado)

Arquivo poucos dados repetidos

Exemplo 3: entrada com tamanho de 110KB gerou saída de 39,2KB (redução de 64,4% do espaço ocupado)

Testes com a finalidade de ver o tempo e o espaço

Tamanho	Tempo
9,05 KB	0.052s
36,7 KB	0.171s
86,4 KB	0.727s
208 KB	1.879s

Tamanho	Espaço ocupado em bytes
9,05 KB	121.050
36,7 KB	444.611
86,4 KB	1.036.220
208 KB	2.433.226

5 Conclusão

É possível observar que para implementar esse algoritmo de compressão de dados temos alguns desafios. É necessário compreender como vai ser comparado um padrão (CS) com um texto. Assim como saber manipular bits usando operações lógicas e shift para colocar os dados em binário.

Para verificar as redundâncias e saber qual redundância selecionar foi necessário utilizar um algoritmo de processamento de cadeia de caracteres (no caso o KMP). Com o uso KMP foi possível identificar as redundâncias do arquivo na maneira especificada e, desta forma, representá-las de uma forma em que ocupe menos espaço no arquivo de saída.