Trabalho Prático 0: LZ77

Algoritmos e Estruturas de Dados III – 2016/1
Entrega: 05/04/2016

1 Introdução

Algoritmos de compressão têm como objetivo codificar informação, como vídeo, música, texto, utilizando menos bits que a representação original. De forma bem ampla, existem dois tipos de compressão: com e sem perdas. Na compressão sem perdas (*lossless compression*), o algoritmo procura por redundâncias nos dados e as remove, dessa forma reduzindo a quantidade de bits necessários para representar o dado original. Como o nome sugere, em tais métodos não há perda alguma de informação, ou seja, um arquivo descompactado a partir de uma compressão sem perdas é idêntico ao arquivo original, bit por bit. Por outro lado, em métodos de compressão com perda (*lossy compression*), o algoritmo identifica informações não essenciais e as remove. Tais algoritmos são utilizados em contextos onde é aceitável a perda de uma pequena quantidade de informação em favor de uma taxa de compressão maior, como por exemplo em arquivos de mídia: fotos, vídeos, áudios. Neste TP você irá implementar um compactador e um descompactador que fazem uso do algoritmo de compressão sem perdas LZ77.

2 LZ77

O algoritmo LZ77 é um algoritmo de compressão sem perdas publicado por Abraham Lempel e Jacob Ziv em 1977. Este algoritmo é de extrema importância, pois além de ser a base para diversos algoritmos de compressão, ele também é utilizado em vários formatos de arquivo: GIF, PNG e GZIP.

O LZ77 comprime dados por meio da redução de redundâncias, substituindo múltiplas ocorrências de uma sequência de bytes δ por referências à uma ocorrência anterior da sequência: δ_0. Damos o nome de *matching* às ocorrências posteriores de δ. Dessa forma, a saída do algoritmo é composta por dois tipos de símbolos:

1. **Literais**: bytes ou sequência de bytes para os quais não foi encontrada uma ocorrência anterior de comprimento 3 ou maior.

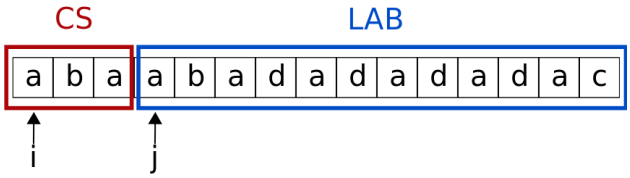
2. **Ponteiros**: tuplas no formato $\langle$`comprimento`, `r_offset`$\rangle$, onde `comprimento` é o tamanho em bytes do *matching* encontrado e `r_offset` é a distância relativa entre a posição do byte inicial do substring δ para a posição do byte inicial de δ_0. Note que as posições mencionadas se referem ao conteúdo antes da compactação.

Diferentes implementações do LZ77 estabelecem diferentes limites para os valores de `comprimento` e `r_offset`. Para este TP, `comprimento` pode assumir valores no intervalo [3, 258] e `r_offset` valores no intervalo [1, 32768]. Assim, o tamanho mínimo para um *matching* ser substituído por um ponteiro é 3.

Exemplo 1

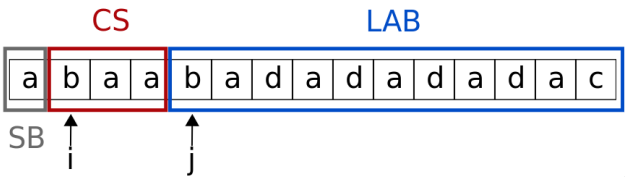
Para melhor ilustrarmos como o LZ77 funciona, iremos utilizá-lo para comprimir a string de entrada *abaabadadadadac*. Seja o *Search Buffer* (SB) a porção da entrada já processada. Esta é a região onde iremos procurar por δ_0. *Current Substring* (CS), como o nome diz, é a substring corrente sendo considerada para o *matching*, e o *Lookahead Buffer* (LAB) é a porção ainda não lida da entrada, o qual pode fazer parte do *matching*.

Passo 1 O algoritmo começa com $i = 0$, início da string atual, e $j = 3$, índice do próximo byte. Então, ele tenta encontrar uma ocorrência anterior da substring *aba* (pois o *matching* mínimo é 3), porém não encontra. Logo, o primeiro byte do CS, *a*, é impresso como um literal e i e j são incrementados em 1.
Saída S = *a*

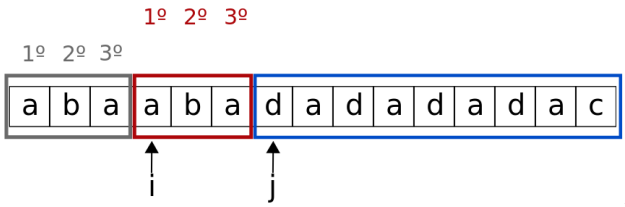


Passos 2 e 3 Com $i = 1$, $j = 4$ e CS = *baa*, o algoritmo tenta encontrar uma ocorrência anterior de *baa*, porém não encontra. Portanto, o algoritmo imprime o primeiro byte do CS, *b*, e muda o CS para o próximo substring da entrada com comprimento 3: *aab*. Novamente, não há nenhuma ocorrência

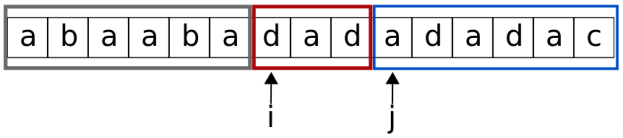
anterior desta sequência de bytes no SB e, então, o algoritmo imprime o literal a.
Saída S = aba



Passo 4 Com $i = 3$ e $j = 6$, o algoritmo procura por uma ocorrência anterior de aba e acha-a começando 3 posições atrás. Como o comprimento do *matching* é 3 e a distância relativa também é 3, o algoritmo imprime o ponteiro $\langle 3, 3\rangle$. i e j avançam 3 posições, por causa do comprimento do *matching*.
Saída S = $aba\langle 3, 3\rangle$

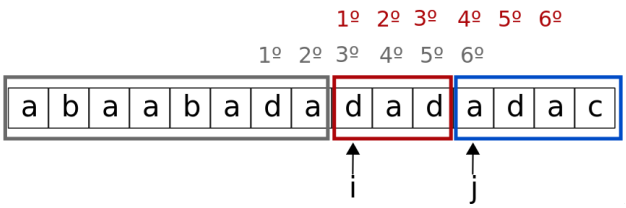


Passos 5 e 6 Com $i = 6$ e $j = 9$, o algoritmo não encontra instâncias anteriores do CS atual: dad. Escreve literal d. O mesmo ocorre para o próximo CS: ada. Escreve a.
Saída S = $aba\langle 3, 3\rangle da$

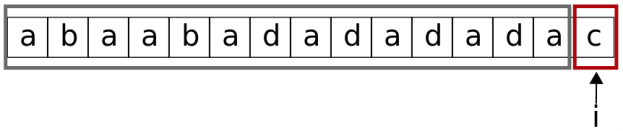


Passo 7 Com $i = 8$ e $j = 11$, a próxima substring a ser considerada é dad novamente. Esta substring aparece anteriormente na entrada, da posição $i = 6$ até $i = 8$. Note que neste *matching* as substrings em consideração compartilham o d da posição 8. Nestes casos, dizemos que o *matching* é auto-referenciado e tais *matching* são caracterizados pelo `comprimento` ser maior que o `r_offset`. Se o processo de *matching* parasse por aqui, o algoritmo

imprimiria o ponteiro $\langle 3, 2\rangle$. Porém, como foi dito anteriormente, o LZ77 sempre tenta achar o maior *matching* possível. Portanto, uma vez encontrado um *matching* inicial de tamanho 3, o algoritmo tentará expandi-lo, analisando os bytes que sucedem as substrings em consideração, continuando a expansão enquanto os bytes forem iguais. Nesse caso, o processo de expansão começa pelos bytes das posições $j_1 = 9$ e $j_2 = 11$. Como ambos são iguais, a, o algoritmo compara em seguida $j_1 = 10$ a $j_2 = 12$ (também iguais) e continua até encontrar uma divergência nas posições $j_1 = 12$ e $j_2 = 14$, com bytes d e c respectivamente. As substrings para as quais realizamos o *matching* foram *dadada*, com a primeiro ocorrência entre as posições $i_1 = 6$ e $j_1 = 12$ e a segunda ocorrência entre $i_2 = 8$ e $j_2 = 14$ (j_1 *e* j_2 *não inclusos*). Como a distância relativa das ocorrências é 2 e o comprimento é 6, o algoritmo imprime o ponteiro $\langle 6, 2\rangle$. Novamente, note que o `comprimento` é maior que o `r_offset` relativo, ou seja, o algoritmo conseguiu utilizar o prefixo *dad* para gerar a substring *dadada*, o que caracteriza um *matching* auto-referenciado.
Saída S = $aba\langle 3, 3\rangle da\langle 6, 2\rangle$



Passo 8 Com somente um byte restante, c, não é possível obter um *matching* com 3 ou mais bytes. Logo, o algoritmo imprime o literal c.
Saída S = $aba\langle 3, 3\rangle da\langle 6, 2\rangle c$



Fique atento aos intervalos dos valores de `comprimento` e `r_offset`. Como o maior `r_offset` permitido é 32768, este também é o tamanho do SB. De forma similar, o LAB tem tamanho 258, pois este é o tamanho do maior *matching* permitido.

2.1 Critérios para *matching*

O algoritmo deverá achar δ_0 no SB que fornece o maior *matching* com a substring no início do LAB, dando preferência aos *matchings* com menor `r_offset`. Por exemplo, se no Exemplo 1 a entrada tivesse um sufixo *aba*, haveria dois δ_0 possíveis: um começando em $i = 3$ e outro em $i = 0$. Neste caso, como ambos *matchings* possuem o mesmo comprimento, o algoritmo deve escolher o δ_0 que começa em $i = 3$ (mais recente) e, portanto, imprimindo o ponteiro $\langle 3, 12 \rangle$. Por outro lado, se o sufixo fosse *abaa*, novamente teríamos dois candidatos a δ_0 com *aba*, $i = 3$ e $i = 0$. Porém, neste cenário o *matching* com δ_0 mais antigo ($i = 0$) seria maior, com comprimento 4, e, portanto, deveria ser escolhido pelo algoritmo, resultando no ponteiro: $\langle 4, 15 \rangle$.

2.2 Codificação da saída

Os leitores mais atentos notaram que no Exemplo 1 expandimos a entrada ao invés de compactá-la. Inicialmente, ela tinha tamanho de 15 bytes e após a "compactação" ela passou a consumir 16 bytes. Isso ocorreu porque representamos a saída do algoritmo de forma "didática", com uma codificação textual. Entretanto, na prática **a saída do LZ77 é binária**.

Para realizar a codificação dos símbolos, usaremos códigos de comprimento fixo, no qual o primeiro bit irá identificar o símbolo codificado: 0 para literais e 1 para ponteiros.

2.2.1 Codificação de Símbolos

Literais Literais utilizarão 9 bits e serão a concatenação do bit 0[1] com a representação binária do valor do byte em questão. Por exemplo, para codificar o byte 97 (*01100001*), que em ASCII corresponde ao caracter *a*, utilizaremos o código *0 01100001*.

Ponteiros Para representar ponteiros, utilizaremos um código de comprimento fixo de 24 bits, ou seja, independente de qual ponteiro seja codificado, sua representação binária sempre terá 24 bits. Como foi dito anteriormente, o primeiro bit (identificação) será sempre 1. Os próximos 8 bits serão utilizados para codificar `comprimento` e os últimos 15 bits serão utilizados para representar `r_offset`. Por exemplo, o ponteiro $\langle 3, 2 \rangle$ possui a seguinte representação binária: *1 00000000 000000000000001*. Note que representamos o `comprimento` 3 usando o valor binário 0, isso porque 3 é o menor

[1] Bit que identifica um literal

`comprimento` possível. O mesmo vale para `r_offset`. Como o menor valor possível é 1, representamos `r_offset` de valor 2 com uma representação binária de 15 bits do valor 1.

É importante notar que, pelo fato do comprimento (em bits) da representação de literais não ser múltiplo de 8, é comum que a representação de literais e ponteiros não esteja alinhada com o inicio e términos de bytes.

2.2.2 *Endianess*

Endianness se refere à ordem que os bytes de uma palavra (*word*) são representadas no processador. Dado que o nosso compactador lida com códigos binários de mais de um byte de **big-endian**, ou seja, os códigos são escritos com o byte mais significativo primeiro. Por exemplo, no caso do ponteiro $\langle 3, 2 \rangle$, os bytes devem ser escritos conforme eles foram mostrados: *10000000 00000000 00000001*. Caso tivéssemos utilizado little-endian, este mesmo código deveria ser escrito como: *00000001 00000000 10000000*, ou seja, com o byte menos significativo primeiro. Esse é um detalhe importante, pois o compactador **deverá produzir um código big-endian**, independente do *endianess* do processador no qual ele é executado.

2.2.3 Bytes incompletos no fim da compactação

Como foi dito anteriormente, pelo fato de literais terem códigos de 9 bits, raramente a codificação de símbolos irá respeitar a fronteira entre bytes. Dessa forma, é provável que o último byte do arquivo compactado seja incompleto, i.e. o último símbolo codificado não utiliza todos os 8 bits do byte. **Nesse caso, os bits não utilizados deverão ser setados para** 0. Isso é de extrema importância para garantir que a saída do seu TP seja idêntica às saídas de referência utilizadas nos testes.

No caso do Exemplo 1, a saída em binário ficaria da seguinte forma: *00110000 10011000 10001100 00110000 00000000 00000000 01000110 01000011 00001100 00001100 00000000 00001001 10001100*.

2.3 Descompactação

A partir da explicação da compactação, o entendimento da descompactação é simples. Deve-se decodificar os bytes do arquivo compactado enquanto o número de bits seja suficiente para traduzir um literal ou um ponteiro. Note que o máximo que um literal ou um ponteiro pode ocupar são 3 bytes.

No início, o primeiro bit lido corresponde ao identificador do tipo de objeto codificado, literal ou ponteiro. Caso ele seja 0, os próximos 8 bits

correspondem diretamente ao literal codificado, sem alterações. Caso o bit seja 1, os próximos 8 bits são o `comprimento` (começando do `comprimento` 3 igual a 0), e em seguida, os 15 bits consecutivos são o `r_offset` (começando do `r_offset` 1 igual a 0). Note que como o `r_offset` máximo é igual a 32KB, deve-se armazenar a saída dos últimos 32KB para decodificar os ponteiros.

A dificuldade encontra-se na manipulação dos bits e das estruturas que devem armazenar a entrada e a saída. É importante notar que os bits lidos na entrada estarão separados em bytes diferentes, ao contrário dos bits que devem ser escritos. Lembre-se de que os números de bytes usados pelos tipos de variáveis podem ser diferentes, dependendo da arquitetura.

3 O que deve ser implementado

Você deverá implementar o compactador e o descompactador para o algoritmo LZ77 descrito acima. O `Makefile` que será submetido com o código deve permitir que ambos programas sejam compilados ao digitar o comando *'make all'* no terminal e os executáveis deverão ser nomeados: `compress.exe` para o compactador e `decompress.exe` para o descompactador. É de extrema importância que esses critérios sejam seguidos à risca, para que seu código seja compatível com os scripts de teste. Caso contrário, o seu TP não poderá ser testado e, consequentemente, você irá tirar 0.

4 Entrada e Saída

Tanto o compactador quanto o descompactador deverão ler da linha de comando dois parâmetros: **nessa ordem:** (i) nome do arquivo de entrada; (ii) nome do arquivo de saída. Para o compactador, a entrada será um arquivo qualquer (e.g. texto, video, imagem) e a saída será um arquivo binário que segue o formato especificado na Seção 2.2. Já no caso do descompactador, é o contrário. O arquivo de entrada será um binário comprido utilizando o LZ77 e a saída será um arquivo qualquer. Abaixo segue um exemplo de como os seus programas serão executados pelo sistema de teste:

```
# Compactando uma imagem
./compress.exe imagem.png imagem.lz77

# Obtendo a imagem original
./decompress.exe imagem.lz77 imagem_descompactada.png
```

Note que o formato do arquivo não-compactado, `imagem.png` no exemplo anterior, é irrelevante. Esse é o caso porque seu programa irá ler o arquivo (compactação) e escrevê-lo (descompactação) como uma sequência de bytes. Consequentemente, o processo de compactação de um texto ou de uma imagem, por exemplo, é idêntico.

5 O que deve ser entregue

Deverá ser submetido um arquivo *.zip* contendo somente uma pasta chamada `tp0` e dentro desta deverá ter: (i) Documentação e (ii) Implementação.

Documentação Poderá ter no máximo 10 páginas e deverá seguir tanto os critérios de avaliação discutidos na Seção 6.1, bem como as diretrizes sobre a elaboração de documentações disponibilizadas no *moodle*. Além desses requisitos básicos, a documentação do tp0 deverá ter:

1. Experimentos mostrando como a taxa de compressão varia para diferentes tipos de arquivo, e.g. texto, imagem, video etc. Levante hipóteses do porquê.

2. Experimentos mostrando se o tipo de arquivo sendo compactado afeta o *throughput* de compressão (MB/s). Levante hipóteses do porquê.

3. [**Extra**] Proponha uma mudança que poderia levar a melhorias na taxa de compressão. A mudança pode ser no algoritmo LZ77 e/ou no esquema de codificação descrito na Seção 2.2

Implementação Código fonte do seu TP (*.c* e *.h*) e um arquivo *Makefile* para realizar a compilação do seu código, atendendo aos critérios descritos na Seção 3.

6 Avaliação

O compactador valerá 70% da nota total, e o descompactador, 30%. Seu trabalho será avaliado de acordo com a qualidade do código e documentação submetidos. Eis uma lista **não exaustiva** dos critérios de avaliação que serão utilizados.

6.1 Documentação

Introdução Inclua uma breve explicação do problema que está sendo resolvido no seu trabalho e um resumo da sua solução.

Solução do Problema Você deve descrever a solução do problema de maneira clara e precisa, detalhando e justificando os algoritmos e estruturas de dados utilizados. Para tal, artificios como pseudo-códigos, exemplos ou diagramas podem ser úteis. Note que documentar uma solução não é o mesmo que documentar seu código. **Não** é necessário incluir trechos de código em sua documentação nem mostrar detalhes de sua implementação, exceto quando estes influenciem o seu algoritmo principal, o que se torna interessante.

Análise de Complexidade Inclua uma análise de complexidade de tempo e espaço dos principais algoritmos e estrutura de dados utilizados. Cada complexidade apresentada deverá ser devidamente **justificada** para que seja aceita.

Avaliação Experimental Sua documentação deve incluir os resultados de experimentos que avaliem o tempo de execução de seu código em função de características da entrada. Cabe a você gerar entradas para esses experimentos. Por exemplo: se esse trabalho fosse sobre ordenação, seria interessante mostrar como o tempo de execução de cada algoritmo varia quando o número de items a serem ordenados aumenta. Para tal, um gráfico mostrando o tempo de execução em função do tamanho da entrada pode ser interessante. Você também deve interpretar os resultados obtidos. Comente sobre cada gráfico ou tabela que você apresentar mostrando o que é possível concluir a partir dele. **DICA:** Use o guia e exemplos de documentação disponíveis no moodle.

6.2 Implementação

Linguagem & Ambiente O seu programa deverá ser implementado na linguagem **C** e poderá fazer uso de funções da biblioteca padrão da linguagem. Trabalhos que utilizem qualquer outra linguagem de programação e/ou que façam uso de outras bibliotecas que não a padrão serão zerados. Além disso, certifique-se que seu código compile e funcione corretamente nas máquinas **LINUX** dos laboratórios do DCC.

Casos de teste A sua implementação passará por um processo de correção automatizado e, portanto, o formato da saída do seu programa deve ser idêntico aquele descrito nessa especificação. Saídas com qualquer divergência serão consideradas erradas, mesmo que as divergências sejam *whitespaces*. e.g. espaços, *tabs*, quebras de linha, etc. Para auxilia-lo na depuração do seu código, será fornecido um pequeno, **não-exaustivo**, conjunto de entradas e suas respectivas saídas. É seu dever certificar-se que seu código funciona corretamente para qualquer entrada válida.

Alocação Dinâmica Algoritmos e estruturas de dados deverão fazer uso de memória alocada dinâmicamente (`malloc()` ou `calloc()`). Certifique-se que seu programa utiliza essas regiões de memória corretamente, pois os monitores penalizarão implementações que realizam *out-of-bounds access* e que tenham vazamento de memória (não desalocar memoria dinâmica).
DICA: Utilize `valgrind` antes de submeter o seu TP.

Qualidade do código Seu código também será avaliado no quesito de legibilidade, dando atenção, porém não limitando-se, aos seguintes items: (i) **INDENTAÇÃO**; (ii) nomes de variável e função descritivos e claros; (iii) Modularização adequada; (iv) Comentários dentro de funções, explicando o que certos trechos mais complicados fazem; (v) Comentários fora de funções, explicando, em alto-nível, o que as funções mais importantes fazem; (vi) funções concisas que desempenham somente uma tarefa; (vii) **Proibido uso de variáveis globais**; (viii) Quem usar `goto` zera a matéria =].
DICA: Consulte o exemplo de código no moodle

7 Considerações Finais

Dicas

1. Para abrir um arquivo para leitura binária com o `fopen`, use a flag "rb". Para escrita, use "wb".

2. Para manipulação de bits, use variáveis *unsigned*. O tipo *char* é implementado como um *int* em C.

3. A biblioteca *stdint.h* é útil para escrever código portável.

4. `xxd -b` ⟨`arquivo`⟩ exibe o contéudo do arquivo em binário (0's e 1's)

5. Calculadora disponível no Ubuntu e no Gnome tem um modo **programação** que mostra a representação binária **big-endian** de um número decimal

Atrasos Trabalhos poderão ser entregues após o prazo estabelecido, porém sujeitos a uma penalização regida pela seguinte fórmula:

$$\Delta_p = \frac{2^{d-1}}{0.32}\,\%$$

Por exemplo, se a nota dada pelo corretor for 70 e você entregou o TP com 4 dias corridos de atraso, sua penalização será de $\Delta_p = 25\%$ e, portanto, a sua nota final será: $N_f = 70 \cdot (1 - \Delta_p) = 52.2$. Note que a penalização é exponencial e 6 dias de atraso resultam em uma penalização de 100%.

8 Sobre referências

Esta documentação pode não estar isenta de ambiguidades. Tenha certeza de entender todos os pontos do trabalho. Este trabalho prático foi baseado no algoritmo DEFLATE do compactador Gzip, de forma que ele foi bastante simplificado, e assim, tanto a versão do LZ77 a ser implementada difere em alguns pontos das explicações em outras referências, como a codificação usada aqui é única, pois o Gzip usa uma codificação ainda mais compacta, de tamanho variável. Em caso de dúvidas, submeta-as no fórum do *minha.ufmg*.