Trabalho Prático 1 de Algoritmos 2

Nome: Artur Gaspar da Silva

Explicando o que foi feito:

O intuito deste trabalho foi implementar um algoritmo de compressão de arquivo que utiliza Tries, baseado no método LZ78. Para mais detalhes da especificação do trabalho, leia o arquivo "tp1.txt". Nesse relatório será explicado em maior detalhe as escolhas de implementação do algoritmo, como compilar e como executar o programa, além de uma brevíssima descrição dos arquivos presentes aqui e uma descrição do formato do arquivo comprimido.

Descrevendo rapidamente o algoritmo de compressão e de descompressão:

Tanto a compressão quanto a descompressão utilizam tries, que codificam o texto. A mesma trie deve ser montada para as etapas de compressão e descompressão. Veremos primeiro como montar a trie com o arquivo original, depois como ela é salva no arquivo comprimido (com uma intuição bem rápida de porque esse processo realmente deve comprimir o arquivo), e por fim como recuperar a trie original e o texto original a partir do arquivo comprimido.

Compressão (montar trie com arquivo original e criar arquivo comprimido):

Para montar a trie com base no arquivo original, fazemos o seguinte: itereamos por cada caractere do arquivo a ser comprimido, mantendo em uma string S (inicialmente vazia) quais caracteres não foram salvos ainda na trie. A cada passo então adicionamos um caractere no final de S, e em seguida verificamos se S está na trie atual (que o inicialmente tem apenas o nó 0). Se ela estiver, não fazemos nada e continuamos. Se não estiver, adicionamos S à trie e resetamos S como a string vazia. Ao final, se o arquivo terminou e S não é vazia, fazemos uma chamada extra para adicionar S à trie.

No meio desse processo todo, fazemos uma coisa extra na trie: quando uma string é adicionada, adicionamos caractere a caractere, e a cada caractere novo (ou seja, os caracteres depois do maior prefixo de S que estava na trie) adicionado criaremos um novo nó na trie. Esse nó possuirá um código que o identifica unicamente na trie, s*endo que o i-ésimo caractere adicionado nesse processo tem código i* (o nó de código 0 é o inicial que já começou na trie), e esse nó também representa uma string (a string que os caracteres das arestas no caminho da trie que vai de 0 até ele formam). Sempre que o nó de código n for criado, adicionaremos um par (m,c) em que m é o código do nó da trie que aponta para n e cuja aresta é identificada pelo caractere c (pois numa trie as arestas possuem um caractere cada).

Na etapa extra final, em que o arquivo acabou e S não era vazia, adicionamos na lista também o valor (m1,c1), em que c1 é o último caractere de S e m1 é o nó que identifica o nó da trie que representa S[1..n-1], em que n é o tamanho de S. Ou seja, adicionamos um par que indica que adicionamos c1 à string representada por m1 teremos S.

Em resumo: teremos uma trie, e uma lista que permite que recuperemos tanto a trie quanto o texto. Afinal, a lista é do tipo (m_i,c_i), com 1 <= i <= N indicando a posição do código na lista, que pela forma como o algoritmo foi executado até aqui indica que ao adicionar o caractere c_i ao nó m_i teremos o nó i. Se o último elemento levar a um nó que já estava presente, isso significa que o texto original tinha a string que esse nó representa no final dele.

O texto comprimido será um arquivo com essa lista de pares. Veja os detalhes de implementação para saber como exatamente salvamos isso de forma eficiente.

A ideia de porque isso comprimiria o texto é que é esperado que várias sequências de caracteres se repitam, e quando isso acontecer não vamos precisar mostrar a sequência toda, pois um código da trie sozinho é capaz de representar uma sequência de caracteres completa. Mas note que se os caracteres forem completamente aleatórios isso não ajuda muito. Na verdade pode

atrapalhar, pois teremos uma quantidade grande de códigos e cada código representará uma sequência bem pequena de caracteres, sendo necessários mais bits para representar cada código e ganhando muito pouco com cada código. Foi feito um caso de teste bônus assim, depois observe o resultado na seção de resultados.

Desompressão (montar trie com arquivo comprimido e recuperar o arquivo original):

O arquivo comprimido, como visto na seção anterior, já é praticamente uma receita direta de como montar a trie original. Simplesmente andamos pelos N pares (m_i,c_i), com $1 <= i <= N$ indicando a posição do código na lista, e criamos a trie de acordo, adicionando uma aresta com caractere c_i que vai do nó m_i para o nó i. Como os nós i vem em ordem, teremos a trie toda no final.

Pela forma como a lista foi montada, para recuperar o texto original basta escrever a string que o nó i representa a cada novo nó i criado. E caso no final o último nó já estivesse presente (se seguir o caractere c_N a partir do nó m_N já leva a um nó existente), basta adicionar a string que esse nó identifica. Ao final desse processo, teremos o arquivo original.

Escolhas de implementação:

Escolha da Trie Simples (ao invés da compacta):

Escolhemos implementar uma trie simples (ao invés de uma trie compacta), pois a trie compacta não apresentaria benefícios para esse trabalho: se fóssemos pegar todos os prefixos ou todos os sufixos do texto, a trie compacta teria complexidade assintótica de espaço linear, enquanto a trie padrão teria complexidade quadrática. Mas esse não é o caso, aqui a trie simples já é suficiente.

Escolha do limite da quantidade de códigos

Aqui foi decidido que o texto não teria mais que 2^{30}-1 caracteres, e portanto o arquivo comprimido não teria mais que essa quantidade de códigos (afinal no máximo adicionamos um código por caractere lido, nunca mais que isso). Note que 2^{30}-1 caracteres seriam mais de 1Gb de arquivo, mesmo com um algoritmo linear no tamanho da entrada é esperado que isso demore mais de um minuto num computador comum.

Escolha da representação de códigos e caracteres.

A fim de economizar espaço, foi tomada a decisão de usar o mínimo possível de bits para representar um caractere, e o mínimo possível de bits para representar um código. Ou seja, se temos 4 caracteres diferentes por exemplo, cada um será representado por 2 bits apenas. Se tivermos 5 códigos, cada um será representado por 3 bits (pois 3 é o menor inteiro tal que $2^3 >= 5$). Foi preciso implementar uma classe “buffer” de bits, porque o C++ só salva bytes inteiros nos arquivos.

Brevíssima descrição dos arquivos presentes:

Pra todo “$i” inteiro entre 01 e 10 (incluindo 01 e 10), “teste$i.txt” é um dos casos de teste padrão pedidos. Veja a seção “Resultados” pra uma melhor descrição do que é cada um.

Para todo “$i” inteiro igual a 1 ou 2, “bonus$i.txt” é um caso de teste bônus, que eu achei interessante. Veja a seção “Resultados” pra uma melhor descrição do que é cada um.

O arquivo “trie_utils.cpp” implementa a classe da trie com todas suas funcionalidades, e algumas utilidades como “imprimir erro de leitura de arquivo”.

O arquivo “d_compressor.cpp” implementa o algoritmo principal, utilizando-se das funcionalidades implementadas em “trie_utils.cpp”.

Descrição do formato do arquivo comprimido:

O arquivo comprimido é um arquivo binário, em que os primeiros 9 bits indicam quantos caracteres diferentes foram usados, seja essa quantidade m. Em seguida temos m blocos de 8 bits que indicam quais caracteres foram usados, em ordem crescente de valor binário. Logo depois,

temos 30 bits que indicam quantos pares (código, caractere) estão presentes no arquivo, seja esse número n. Por fim, temos n pares (código, caractere), em que cada código é representado em $\lceil \log_2(n) \rceil$ bits e cada caractere é representado em $\lceil \log_2(m) \rceil$ bits. Afinal, temos no máximo n códigos distintos e m caracteres distintos, então essa quantidade de bits é o mínimo possível.

Como compilar o programa:

Com os arquivos “d_compressor.cpp” e “trie_utils.cpp” no mesmo diretório do terminal, utilize o comando “g++ -O3 d_compressor.cpp -o d_compressor.out”.

Como executar o programa:

Compressão

./<programa> -c <arquivo_entrada> [-o <arquivo_saida>]

Descompressão

./<programa> -x <arquivo_entrada> [-o <arquivo_saida>]

A especificação do arquivo de saída é opcional. Caso ela não seja dada, o nome do arquivo de saída para compressão deve ser o nome original do arquivo acrescido da extensão z78. No caso da descompressão, será o nome do arquivo original, suprimida a extensão atual e acrescentada a extensão .txt.

Resultados:

Casos de teste:

teste01.txt: historinhas escritas por mim sobre um mundo de RPG on-line que eu estava inventando quando era mais novo. Taxa de compressão: arquivo comprimido com (34 026)/(61 621) = 55.22% do tamanho original, o original é (61 621)/(34 026) = 1.81 vezes maior.

teste02.txt: mais algumas historinhas escritas por mim quando eu era mais novo, de personagens que eu inventei pro mundo do teste01. Taxa de compressão: arquivo comprimido com (3 790)/(5 532) = 68,51% do tamanho original, o original é (5 532)/(3 790) = 1.46 vezes maior.

teste03.txt: resumo que eu escrevi quando era mais novo das histórias do teste01 e teste02. Taxa de compressão: arquivo comprimido com (6 800)/(10 617) = 64.04% do tamanho original, o original é (10 617)/(6 800) = 1.56 vezes maior.

teste04.txt: segunda versão do resumo do teste03, com algumas partes a mais na história. Taxa de compressão: arquivo comprimido com (9 546)/(15 630) = 61.07% do tamanho original, o original é (15 630)/(9 546) = 1.64 vezes maior.

teste05.txt: argumentos escritos por eu mais novo para tentar convencer as outras pessoas que ciravam coisas pro servidor de Minecraft de RPG a deixar essas histórias que eu inventei serem “oficiais”. Taxa de compressão: arquivo comprimido com (3 599)/(5 094) = 70.65% do tamanho original, o original é (5 094)/(3 599) = 1.41 vezes maior.

teste06.txt: Romeu e Julieta, de Willian Sheakspare (https://www.gutenberg.org/cache/epub/1513/pg1513.txt). Taxa de compressão: arquivo comprimido com (95 210)/(163 624) = 58,19% do tamanho original, o original é (163 624)/(95 210) = 1,72 vezes maior.

teste07.txt: The Critique Of Pure Reason, de Immanuel Kant (https://www.gutenberg.org/files/4280/4280-0.txt). Taxa de compressão: arquivo comprimido com (542 615)/(1 294 029) = 41.93% do tamanho original, o original é (1 294 029)/(542 615) = 2.38 vezes maior.

teste08.txt: The call of Cthulhu, de Howard Philips Lovecraft (https://www.gutenberg.org/cache/epub/68283/pg68283.txt). Taxa de compressão: arquivo comprimido com (54 668)/(90 450) = 60.44% do tamanho original, o original é (90 450)/(54 668) = 1.65 vezes maior.

teste09.txt: 山水情, de autor anônimo (https://www.gutenberg.org/files/25146/25146-0.txt). Taxa de compressão: arquivo comprimido com (217 862)/(339 002) = 64.27% do tamanho original, o original é (339 002)/(217 862) = 1.56 vezes maior.

teste10.txt: Also sprach Zarathustra: Ein Buch für Alle und Keinen, de Friedrich Wilhelm Nietzsche (https://www.gutenberg.org/cache/epub/7205/pg7205.txt). Taxa de compressão: arquivo comprimido com (286 784)/(551 325) = 52.02% do tamanho original, o original é (551 325)/(286 784) = 1.92 vezes maior.

Bônus (caso que comprime muito bem e caso que comprime muito mal):

bonus1.txt: “42” repetido um milhão menos uma vezes. Taxa de compressão: arquivo comprimido com (4 603) /(1 999 998) = 0.23% do tamanho original, o original é (1 999 998)/(4 603) = 434,50 vezes maior.

bonus2.txt: 1 999 998 bytes gerados aleatoriamente. Taxa de compressão: o tamanho do arquivo comprimido é (2 397 289)/(1 999 998) = 1.20 vezes maior, o original tem (1 999 998)/(2 397 289) = 83.43% do tamanho comprimido. O algoritmo na verdade piora o resultado para textos completamente aleatórios.