Exercício-Programa 4 de MAC0323 - Expressões Regulares

Matheus Oliveira da Silva, Nº USP: 13696262

12 de Julho, 2023

1 Introdução

Este relatório apresenta uma solução para o quinto exercício-programa proposto na disciplina de Algoritmos e Estrutura de Dados II (MAC0323), optativa livre do curso de Matemática Aplicada e Computacional do Instituto de Matemática e Estatística da Universidade de São Paulo.

Foi solicitado a construção de um projeto de compressão e descompressão de arquivos, baseados no algoritmo de Huffman.

Para tal, foi necessário interpretar a quantidade de repetições em um arquivo de texto e codificar os caracteres mais repetidos de forma a representá-los com menos bits e, assim economizar espaço.

2 Desenvolvimento

2.1 Compressão

2.1.1 Trie

A árvore ótima, trie, representa a codificação dos caracteres presentes no texto.

O primeiro passo para construir a trie é processar a frequência de cada caractere no texto. Tal tarefa foi feita de forma trivial utilizando um map do tipo char e int.

Após computar as frequências, para cada caractere foi criado uma folha - que carrega dois ponteiros nulos e o valor do caractere chamado de ch e o valor da frequência do caractere chamado de $freq$ - que foram adicionados à uma fila de prioridade, onde a prioridade era dada para as folhas de menor frequência.

Para a fila de prioridade comparar corretamente os nós, foi necessário criar uma estrutura de apoio que compara dois nós e retorna verdadeiro caso a frequência do primeiro fosse maior que a do segundo. Com isso, a fila de prioridade armazena corretamente as folhas.

E então, o algoritmo de Huffman para a construção da trie foi implementado. Os passos são:

1. Retirar as duas primeiras folhas da fila, ou seja, as duas de menor frequência, e ligá-las à um nó pai cuja frequência será dada pela soma das frequências das filhas. O nó pai possui o valor $ch = 0$;
2. Adicionar o nó pai para a fila, mantendo a regra de prioridade;
3. Repetir o primeiro e o segundo passo até que reste somente um nó na fila.

O último nó na fila é a raiz da árvore trie.

2.1.2 Mapa de Códigos

Com a árvore criada, obtemos o mapa que armazena cada código para cada caractere transcorrendo a árvore.

Começando da raiz, para cada nó à esquerda o número 0 é adicionado, e para cada nó à direita o número 1 é adicionado. Ao chegar em uma folha obtêm-se a codificação binária para o caractere da mesma. Essa tarefa é feita pela função $voidgeneratesCode(Node*root, std :: stringcurrentCode, std :: map < char, std :: string > \&codeMap$, onde root é o nó da raiz em primeiro momento e depois se torna o nó atual, currentCode é a string que representa o código do caminho percorrido, e codeMap é o mapa de códigos que será alterado para inserir o código em cada caractere.

2.1.3 Compressão do texto

Obtendo o mapa de códigos é fácil traduzir o texto original para a versão binária. Para cada caractere no texto, procura-se o código no mapa e então adiciona no final de uma string representando o texto codificado.

2.1.4 Criando o arquivo trie.bin

Decidi separar o arquivo binário trie.bin, que representa a árvore codificada, do arquivo do texto codificado. O tamanho dos arquivos separados não aumenta e facilita a implementação da compressão e descompressão. Com o único risco de perder-se o arquivo da árvore, tornando o arquivo do texto codificado impossível de descomprimir.

A função que escreve a árvore no arquivo binário é a $voidwriteTrie(Node*root, std :: ofstream\&outFile)\&$. A lógica de inserção é:

1. Se não é uma folha, adiciona 0 e chama recursivamente $writeTrie$ começando pelo nó da esquerda e depois da direita;
2. Se é uma folha, adiciona 1 e em seguida adiciona o char.

Obtemos, a árvore trie em um arquivo binário. A reconstrução da árvore será comentada mais adiante.

2.1.5 Criando o arquivo texto.bin

O arquivo binário texto.bin representa todo o texto original na sua forma comprimida. Para que tal compressão seja eficiente é necessário alterar bits por bits em um arquivo binário. No livro Algorithms[3], é mencionado duas APIS:

1. BinaryStdIn: Extensão da StdIn, porém para leitura de arquivos binários;
2. BinaryStdOut: Extensão da StdOut, porém para a inserção em arquivos binários.

Como não há dicas ou informações sobre suas implementações, tentei buscar alternativas para modificar e ler arquivos binários.

A solução mais próxima do apresentado em Algorithms[3] foi utilizar a biblioteca $fstream$.

Para a inserção de bits no arquivo de texto comprimido, a função $writeBitToFile(std :: ofstream\&outFile, boolbit)$ foi criada. A leitura padrão de arquivos binários feita pela biblioteca $fstream$ é feita por Byte, e não bit. Portanto, foi necessário manter a variável bitCount para saber qual bit estava sendo lido no byte atual.

Foi um sucesso utilizar tal adaptação, porém, o código apresenta uma falha. Como a leitura é feita a partir de 8 bits, ou 1 Byte, o texto comprimido deve conter um número de bits múltiplo de 8. Dessa forma, tive que inserir bits de complemento no final do arquivo. Optei por inserir bits 0. Dependendo da codificação uma ou mais letras podem ser inseridas no fim do texto de forma errônea.

É um erro de até no máximo 4 caracters, visto que nossos códigos possuem dois ou mais bits.

2.2 Descompressão

2.2.1 Trie

A descompressão precisa ler a árvore do seu arquivo binário e assim conseguir os códigos necessários para a descompressão.

Para a leitura da árvore, a função $Node * readTrie(std :: ifstream\&inFile)$ foi criada. A lógica de montagem da árvore é:

1. Lê o primeiro bit;
2. Caso seja 1, lê o próximo caractere armazenado, e cria uma nova folha - portanto, sem filhos - com esse caractere;
3. Caso seja 0, cria um novo nó com caractere nulo e chama recursivamente $readTrie$ para seu filho da esquerda e em seguida para seu filho da direita.

Dessa forma, a trie é reconstruída. É importante lembrar que a frequência não é uma informação valiosa para a descompressão, e por isso tal informação não é armazenada no arquivo binário da trie.

2.2.2 Descompressão do texto

Obtendo a trie fica fácil descomprimir o texto, uma vez que temos o código para cada caractere.

A função que realiza esta descompressão é $voidgenerateText(Node*root, std :: string\&currentText, std :: ifstream\&inFile)$. Utilizando a lógica de ler bits por bits, a partir de uma contagem, a função se movimenta na árvore - se encontra um bit 1 para a direita, e se encontra um bit 0 para a esquerda - e ao chegar em uma folha encontra o caractere desejado e o adiciona à uma string.

Por fim essa string, com todos os caracteres do texto já descomprimidos, é inserida em um arquivo de texto, sendo o arquivo de texto descompactado.

3 Testes

3.1 Teste 1

Para o primeiro teste, um texto do Projeto Gutenberg[2] foi comprimido e descomprimido. Seu arquivo original tem um tamanho de 53KB e seu arquivo compactado 31 KB.

3.2 Teste 2

Para o segundo teste, um texto também do Projeto Gutenberg[1] foi comprimido e descomprimido. Seu arquivo original tem um tamanho de 476KB e seu arquivo compactado 261KB.

4 Conclusão

A partir dos testes é possível observar que a compactação e descompactação funcionam bem e o tamanho economizado é significante. Apesar da dificuldade de inserção e leitura em bits na linguagem C++.

References

[1] D. L. Child. *The history of the condition of women, in various ages and nations (vol. 2 of 2) Comprising the women of Europe, America, and South Sea Islands.*

[2] Robert E. Howard. *The haunter of the ring.*

[3] Kevin SEDGEWICK, Robert; WAYNE. *Algorithms.* Pearson Education, 4th edition, 2011.