Trabalho Prático 1

DCC216 - Algoritmos

Igor Joaquim da Silva Costa

1. Introdução

O problema proposto foi implementar o algoritmo LZ78 para compressão de texto com o auxílio de uma árvore trie.

O algoritmo de compressão LZ78 é um método que busca substituir as sequências de caracteres repetidos em um texto por códigos que representam essas sequências, com o objetivo de reduzir o tamanho do arquivo original. Ele funciona em tempo linear em relação ao tamanho do arquivo de entrada, sendo eficiente para comprimir grandes córpus onde as sequências possuem estrutura parecida.

Para implementá-lo, é necessário utilizar um dicionário que possibilite a realização de muitas buscas e inserções de dados. Nesse sentido, a árvore Trie surge como uma estrutura de dados eficiente para a implementação do algoritmo. Este trabalho tem como objetivo a implementação do algoritmo LZ78 utilizando a árvore Trie como dicionário. Além disso, busca-se entender como a implementação do LZ78 pode ser utilizada como base para outros algoritmos de compressão atualmente disponíveis.

Diante do exposto, a documentação presente possui como objetivo detalhar como o sistema foi modelado (Seção 2), como ele foi implementado (Seção 3) e quais seus pontos fortes e limitações (Seção 4). Por fim, o projeto é sumarizado junto com os aprendizados gerados durante a produção do trabalho(Seção 5). A (Seção 6) sumariza as informações de compilação e execução.

2. Modelagem

Esta seção tem como objetivo discutir as decisões que levaram à atual modelagem do programa.

2.1 Dicionário como árvore trie

Conforme mencionado na seção 1, o problema de compressão de arquivos de texto por meio do método LZ78 pode ser resolvido de maneira eficiente por meio do uso de uma árvore trie. A eficiência dessa estrutura de dados se dá pelo fato de que ela é capaz de aproveitar ao

máximo a estrutura da informação que está sendo armazenada. Cada nó na trie representa um caractere, e o caminho da raiz até o nó representa a sequência de caracteres correspondente. Dessa forma, ao realizar uma pesquisa em uma trie, é possível percorrer a árvore de maneira rápida e eficiente, sem a necessidade de comparar cada caractere individualmente.

A busca pode ser realizada em tempo linear em relação ao tamanho da sequência de caracteres, o que é especialmente útil em casos em que há muitas sequências de caracteres comuns em um conjunto de dados, como é o caso da compressão de dados.

No contexto do LZ78, a busca por uma sequência de caracteres pode ser realizada em tempo constante. Isso se deve ao fato de que, ao adicionar uma nova sequência **Sx** ao dicionário, é garantido que ela será representada por um nó filho de **S**. Dessa forma, durante a execução do algoritmo, a cada novo caractere lido, armazenamos qual nó representa **S**, a fim de fazer a busca para o **Sx**. Ao final do processo, são executadas exatamente **|S|** operações O(1), garantindo a eficiência do algoritmo.

Além disso, a inserção de novas sequências de caracteres é feita de forma eficiente na trie, basta criar um novo nó na árvore para representar a nova sequência. Em resumo, a árvore trie é uma estrutura de dados eficiente para a implementação de busca de strings.

2.2 Estrutura de Dados

A definição abstrata apresentada em C++ para a classe Node tem como objetivo representar os nós em uma árvore. Todo nó possui o código identificador, e o caractere por si armazenado. Além disso, a classe possui um ponteiro para o pai do nó atual, que é inicializado como nulo, e um conjunto de ponteiros para outros nós, denominado children, que é organizado de acordo com a ordem lexicográfica dos caracteres e utilizado para representar os filhos do nó atual na árvore trie. A classe também conta com métodos para busca de sequências nos seus filhos e a obtenção da string correspondente ao seu caminhamento.

3. Implementação

A seção de implementação é onde serão apresentados os códigos e algoritmos que foram utilizados para resolver o problema proposto. Serão discutidos os principais aspectos da implementação, incluindo as estruturas de dados utilizadas e a lógica por trás do código. Além disso, serão apresentados exemplos de trechos de código relevantes, a fim de ilustrar a implementação em detalhes e ajudar na compreensão do processo de solução do problema.

3.1. Entrada e saída

No que diz respeito a entrada e saída, para garantir a robustez do método, cada caractere é entendido como 1 Byte. No caso de caracteres especiais que gastam 2 Bytes, como por exemplo “Ç”, cada Byte é entendido separadamente, o que não causa problemas (ver seção 4).

Para a compressão, é lido Byte a Byte do arquivo de entrada até EOF. Sempre que é necessário armazenar um par <Index,Char> da compressão, isso é feito em um arquivo auxiliar chamado “aux.z78”. Após toda a entrada lida, o arquivo “aux.z78” é reprocessado. Isso se dá para diminuir o número de Bytes necessários para indexar os pares. Por padrão, o arquivo “aux.z78” gasta 4 Bytes para armazenar o Index e 1 Byte para armazenar o Char do par comprimido, assim, ao conhecer o número de pares que realmente serão processados, é possível diminuir o número de Bytes usados pela indexação - tomando o log2(Quantidade de pares) -. Ao final do processo, os pares com o Index reduzido são armazenados no arquivo de saída desejado.

Para a descompressão, primeiramente é lida a quantidade de Bytes usadas para armazenar o Index - presentes nos 4 primeiros Bytes do arquivo - assim, são lidos pares <Index,Char> do tamanho de bytes especificado até o final do arquivo, enquanto a descompactação é salva no arquivo de saída.

3.2 Trie

A classe Trie representa uma árvore e contém métodos que realizam operações nessa estrutura de dados. A classe possui um vetor de ponteiros para nós, chamado indexes, que é utilizado para armazenar referências a todos os nós da árvore, permitindo o acesso a qualquer nó a partir de seu index. Esse vetor é usado durante a descompressão.

A classe Trie possui dois métodos add que adicionam uma string ou um caractere na árvore trie. O método add recebe como parâmetro a string ou o caractere a ser adicionado e um ponteiro para o último nó visitado. Esse último parâmetro é opcional e é utilizado para acelerar a busca na árvore trie. A primeira inserção na Trie também inicializa o nó adicional como raiz. O método search busca uma string na árvore e retorna um par contendo o ponteiro para o último nó correspondente à string prefixo da entrada e uma flag booleana que indica se a string foi encontrada. Ambos métodos de adição de nós são usados na compressão e descompressão, enquanto o método de busca só é usado na compressão.

3.3 Compressão

O processo de compressão começa com um dicionário inicial vazio. Em seguida, o algoritmo começa a ler o arquivo de entrada caractere por caractere e, a cada novo caractere, ele busca se a sequência gerada já apareceu anteriormente no texto. Quando uma sequência inédita é encontrada, ela é adicionada ao dicionário com um novo código e a busca recomeça a partir do próximo caractere do arquivo de entrada. Se a sequência já estiver no dicionário, nada é feito. Ao final do processo, se a última sequência lida **Sx** já estiver presente no texto, é feita uma busca ao nó S, sendo escrito o par <Index,x> na saída, onde Index é o index de **S**.

3.4 Descompressão

O processo de descompressão em LZ78 envolve a reconstrução da sequência original de caracteres a partir do arquivo comprimido gerado pelo processo de compressão. Inicialmente, o arquivo comprimido é lido em blocos contendo pares (index, char), onde index é um número que representa um índice em um dicionário de padrões já encontrados e char é um caractere a ser adicionado à sequência. O processo começa com uma sequência vazia e, para cada bloco lido, é encontrado o nó equivalente ao index no dicionário. A sequência é atualizada com o padrão correspondente ao índice mais o caractere adicionado. Essa nova sequência é inserida na Trie. Esse processo é repetido para todos os blocos do arquivo comprimido, até que a sequência original seja totalmente reconstruída. O resultado final é a sequência original de caracteres armazenada em um arquivo.

4. Casos de Teste

A seção de casos de teste apresenta 10 exemplos de compressão de diferentes arquivos, incluindo textos e imagens. Os exemplos foram selecionados para demonstrar a eficácia do algoritmo em diferentes tipos de texto, bem como para apresentar uma ideia geral da taxa de compressão que pode ser alcançada em diferentes situações.

A taxa de compressão segue a fórmula:

$$1 - \frac{Tamanho\ Comprimido}{Tamanho\ Original}$$

4.1 Arquivos de texto

Nome do Arquivo	Tamanho Original (KB)	Tamanho comprimido (KB)	Taxa de compressão
constituicao1988.txt	637	348	45,37%

KoreanDic.html	113	64	43,36%
os_lusiadas.txt	337	191	43,32%
russian.txt	100	58	42,00%
53489-0.txt	167	97	41,92%
geshukunin.txt	439	259	41,00%
pg1513.txt	166	99	40,36%
pg345.txt	862	560	35,03%
pg14765.txt	89	59	33,71%
greek.txt	78	52	33,33%

Sendo assim, a taxa de compressão média foi de 40,68%.

5. Conclusões

Com o intuito de implementar o algoritmo LZ78 para compressão de texto, foi implementado um programa que utiliza algoritmos de armazenamento de sequências para resolver o problema.

Durante o projeto do sistema foram levadas em consideração não só aspectos práticos da implementação de uma modelagem computacional, mas também como a linguagem de programação escolhida poderia ser uma ferramenta útil para chegar no objetivo esperado. Toda a questão de mapear um mini-mundo de interesse em um modelo computacional robusto se mostrou bastante produtiva, levando o aluno a pensar em formas criativas de se resolver e entender o problema, tendo como resultado um extenso aprendizado sobre como pensar, questionar e implementar uma trie na prática. Por fim, o tempo extra usado para projetar o sistema trouxe várias recompensas no sentido da implementação, sendo um aspecto a ser levado para trabalhos futuros.

Nesse sentido, todo o fluxo de trabalho foi essencial para a consolidação de conteúdos aprendidos em sala, além de apresentar, de forma prática, como softwares maiores, mais consistentes, robustos e inteligentes são projetados e implementados.

6. Instruções para compilação e execução:

6.1 Compilação

Existem partes do programa que são compatíveis apenas às versões mais recentes da linguagem c++, dito isso, deve-se seguir as seguintes configurações para a compilação:

Linguagem: C++
Compilador: Gnu g++
Flags de compilação: -std=c++17 -g
Versão da linguagem: standard C++17
Sistema operacional (preferência): distribuições baseadas no kernel Linux 5.15.

O comando para compilar o programa automaticamente está presente no arquivo **"Makefile"** e sua execução é chamada pelo comando **"make all"**. Deste modo, o executável "tp1" estará compilado e pronto para ser utilizado.

6.2 Execução

Seguem as instruções para a execução manual:

1. Certifique-se que o compilável foi gerado de maneira correta, se algum problema ocorrer, execute o comando "make all" presente no "Makefile".

2. Uma vez que os passos anteriores foram cumpridos, execute o programa com o comando:
    a. Compressão:
       /bin/tp1.out -c <arquivo_entrada> [-o <arquivo_saida>]
    b. Descompressão
       ./bin/tp1.out -x <arquivo_entrada> [-o <arquivo_saida>]

3. A saída será impressa no arquivo_saida.