Relatório TP1 – Algoritmos 2

Gabriel Franco Jallais – Matrícula: 2021031890

Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)

Belo Horizonte – MG – Brasil

gfjallais@ufmg.br

1. Introdução

Este relatório tem como objetivo apresentar o desenvolvimento e resultados do trabalho prático de manipulação de sequências, no qual foi implementado um algoritmo para compressão de arquivos texto utilizando o método LZ78. O algoritmo LZ78 é baseado em dicionários e consiste na substituição de strings que se repetem no texto por códigos, com a finalidade de diminuir o número de bytes gravados na saída.

A implementação do algoritmo utilizou árvores de prefixo (Tries padrão) para otimizar as buscas e inserções no dicionário, tendo em vista que o LZ78 executa essas operações com frequência.

O relatório apresentará uma breve descrição da implementação do algoritmo LZ78 e da Trie utilizada, os conceitos teóricos envolvidos na sua implementação, bem como os resultados obtidos na compressão de arquivos de teste e a análise da eficiência do algoritmo.

2. Implementação

O código usado para realizar o trabalho foi desenvolvido em Python 3.10.6 dentro do WSL2 rodando Ubuntu 22.04 LTS, como bibliotecas externas foram utilizadas, a **sys**, para a leitura e escrita de arquivos, **struct**, para a leitura de bytes do arquivo .z78, assim como a **math**, para o cálculo do número de bytes necessário para representar as tuplas do algoritmo.

3. Modelagem

3.1 Abordagem do problema

Para abordar o problema atendo os requisitos do trabalho, foram criadas classes para representar a Trie, utilizada no lugar do dicionário no algoritmo de compressão, assim como os seus nós. Métodos foram criados na classe trie para realizar a inserção, compressão e descompressão dos arquivos passados como argumento na linha de comando.

3.2 LZ78

O algoritmo LZ78 é um método de compressão de dados que usa um dicionário para substituir cadeias de caracteres que se repetem por um código. A compressão ocorre em tempo linear, ou seja, proporcional ao tamanho do arquivo de entrada. O dicionário é construído à medida que o algoritmo processa o arquivo de entrada. Cada entrada no dicionário é uma sequência de caracteres que já ocorreu antes no arquivo de entrada, associada a um código que representa essa sequência. Durante a compressão, se

uma sequência de caracteres já ocorreu anteriormente no arquivo de entrada, ela é substituída por seu código correspondente. Caso contrário, a sequência é adicionada ao dicionário e um novo código é atribuído a ela. A descompressão é realizada usando o mesmo dicionário construído durante a compressão. O algoritmo LZ78 é a base para muitos algoritmos de compressão utilizados atualmente em ferramentas de compressão de dados.

No caso deste trabalho, o dicionário utilizado pelo algoritmo foi substituído por uma trie, visto a recorrência de consultas e inserções feitas pelo algoritmo e para a fixação do conteúdo lecionado em sala.

3.3 Trie e Node

Para, de fato, implementar o algoritmo LZ78 foram construídas as classes Trie e Node, que representa cada nó da árvore de prefixos.

A classe Node possui um construtor com dois parâmetros: **string** e **value**. **string** é uma string que representa o caractere que o nó representa e **value** é o valor associado à string, de acordo com o algoritmo LZ78, equivalente ao código que estaria associado a string utilizando-se um dicionário.

A classe tem as seguintes propriedades:

- **children**: uma lista que contém todos os nós filhos do nó atual
- **string**: uma string que representa a sequência de caracteres que o nó representa
- **value**: o valor associado à string, de acordo com o algoritmo LZ78
- **end_of_word**: um booleano que indica se o nó atual representa o fim de uma palavra

A classe tem um método chamado **is_char_in_children** que recebe um caractere como argumento. Esse método retorna **False** se o caractere não for igual ao valor **string** de nenhum filho na lista **children**. Caso contrário, o método retorna o nó filho correspondente àquele caractere.

A classe Trie implementa uma árvore de prefixos (Trie) e os métodos necessários para codificar e decodificar um texto usando o algoritmo LZ78.

- O método **__init__** inicializa a raiz da árvore com um nó vazio e um nó de fim de linha (com o caractere `\n`), além de uma variável para controlar os códigos a serem atribuídos aos nós.

- O método **find** percorre a árvore procurando por um nó correspondente a uma determinada palavra. Ele retorna True se a palavra for encontrada e False caso contrário.

- O método **insert** insere uma palavra na árvore, criando nós se necessário, e atribuindo os valores correspondentes, seguindo o algoritmo LZ78.

- O método **encode** recebe um conjunto de linhas de texto e um arquivo de saída e codifica o texto usando o algoritmo LZ78, escrevendo a saída no arquivo. Ele itera sobre cada caractere do texto e usa a árvore para determinar a sequência mais longa de caracteres já vista. Se essa sequência ainda não foi vista, ele a adiciona à árvore e

escreve o código correspondente à sequência anteriormente vista e o caractere atual na saída. Se a sequência já foi vista, ele simplesmente a adiciona ao prefixo atual. Para cada linha é escrito no arquivo de saída um código e um caractere de quebra de linha para indicar.

As tuplas, (código, caractere) são escritas em binário utilizando o menor número de bytes necessário para codificar as strings repetidas e para representar os caracteres, esses valores mínimos são obtidos em uma varredura prévia do arquivo utilizando o método **number_of_bytes_needed**.

- O método **get_code** recebe uma sequência de caracteres e retorna o código correspondente a essa sequência na árvore.

- O método **decode** recebe um arquivo de entrada e um arquivo de saída e decodifica o texto usando o algoritmo LZ78, escrevendo a saída no arquivo. Ele lê os primeiros dois bytes do arquivo de entrada para obter informações sobre o número de bytes necessários para representar os códigos e os caracteres. Em seguida, lê cada tupla e executa o algoritmo LZ78 para construir outro dicionário e assim reconstruir o texto. Ele começa com um dicionário contendo os códigos para o caractere vazio e o caractere de fim de linha e adiciona novas entradas conforme necessário. Ele escreve o texto reconstruído no arquivo de saída.

- O método **number_of_bytes_needed** recebe o conjunto de linhas do texto e determina o número de bytes necessários para armazenar os códigos da árvore e para representar os caracteres. Para isso é utilizado o algoritmo LZ78 padrão, mas sem escrever as tuplas na saída, assim é possível identificar o número de bytes para representar o maior código encontrado pelo algoritmo, e durante essa varredura, também é possível encontrar os diferentes caracteres e o maior valor necessário para representá-los.

4. Resultados

A seguir são apresentados os resultados obtidos para na compressão de 10 arquivos de texto, obtidos a partir do projeto Gutenberg. (Free eBooks | Project Gutenberg)

Arquivo de Entrada (.txt)	Tamanho em Bytes	Tamanho Arquivo de Saída (.z78)	Taxa de Compressão
a_cidade_e_as_serras.txt	440K	356K	19,05%
a_reliquia.txt	516K	416K	19,34%
alice_adventures_in_wonderland.txt	168K	144K	14,21%
crime_and_punishment.txt	860K	768K	10,68%

geshukunin.txt	440K	180K	58,97%
iracema.txt	200K	132K	33,85%
memorias_postumas.txt	388K	320K	17,48%
pride_and_prejudice.txt	740K	660K	10,80%
quincas_borba.txt	472K	376K	20,30%
the_great_gatsby.txt	296K	244K	17,51%
Taxa de compressão média			22,22%

O algoritmo LZ78 é um algoritmo de baixa compressão e sem perdas, isto é, não se perde informação ao se compactar o arquivo, por isso, os resultados não são tão impressionantes.

Em média, esta implementação do algoritmo possui uma taxa de compressão de 22,22%. É importante ressaltar que essa taxa pode variar bastante dependendo do tipo de dado que está sendo comprimido e do algoritmo de compressão utilizado.

Observando os resultados individuais de cada arquivo, é possível notar uma variação relativamente alta na taxa de compressão. Por exemplo, o arquivo "geshukunin.txt" obteve uma taxa de compressão de 58,97%, enquanto o arquivo "pride_and_prejudice.txt" teve uma taxa de compressão de apenas 10,80%. Isso se dá pelo número de bytes necessários para representar as strings recorrentes, em textos maiores como pride_and_prejudice e crime_and_punishment, o número de strings salvas na árvore/dicionário do algoritmo é grande, o que implica na necessidade de mais bytes para representar o código dessas strings, outro fator é o número de bytes necessários para representar os caracteres, dependendo da codificação usada e dos caracteres presentes no texto esse número pode variar.

5. Conclusão

Ao concluir o trabalho prático de manipulação de sequências, é possível perceber a utilidade do algoritmo LZ78 em diferentes contextos, especialmente naqueles que envolvem armazenamento ou transmissão de dados em formato de texto. Reduzir o espaço necessário para armazenar ou transmitir arquivos de texto pode trazer vantagens significativas em termos de otimização de recursos.

Durante a implementação do algoritmo, também foi notável a importância do uso de estruturas de dados eficientes na busca e inserção de sequências de caracteres repetidos. A Trie tradicional mostrou-se uma opção viável e eficiente para esse fim, embora haja opções melhores, permitindo que o algoritmo LZ78 identificasse e codificasse as sequências repetidas de forma rápida.

É importante destacar que a eficiência do algoritmo LZ78 pode variar de acordo com o tipo de texto que está sendo comprimido. Como visto na sessão anterior. Existem possíveis melhorias para o algoritmo, que não foram implementadas neste trabalho. Variantes desta solução já foram desenvolvidas e apresentam um desempenho muito melhor.

Apêndice

Para realizar os testes e gerar os resultados expostos na sessão 4, foi utilizado o seguinte script de bash, que se encontra neste repositório.

```bash
#!/bin/bash

# Define o diretório onde os arquivos .txt estão localizados
dir="/home/gfjallais/Code/TP-ALG2/entrada"

# Cria o arquivo results.txt ou sobrescreve o arquivo existente
echo "" > results.txt
ratios=()
# Itera sobre cada arquivo .txt no diretório
for file in $dir/*.txt; do
  # Define o nome dos arquivos de saída
  input="${file}
  output1="${file%.*}_saida.z78"
  output2="${file%.*}_saida.txt"
  log="${file%.*}_log.txt"

  # Executa os comandos e salva a saída em um arquivo de log
  python3 TP_ALG2.py -c "${input}" "${output1}" &> "${log}"
  python3 TP_ALG2.py -x "${output1}" "${output2}" &>> "${log}"
  diff "${input}" "${output2}" &>> "${log}"

  # Obtém o tamanho dos arquivos e os grava no arquivo results.txt
  input_size=$(du -h "${input}" | awk '{print $1}')
  output_size=$(du -h "${output1}" | awk '{print $1}')
  ratio=$(bc -l <<< "1 - $output_size/$input_size")
  ratios+=("$ratio")
  echo "${input} tem tamanho ${input_size}. ${output1} tem tamanho ${output_size}. A relação entre os tamanhos é de ${ratio}." >> results.txt

done

mean=$(printf "%s\n" "${ratios[@]}" | awk '{s+=$1} END {print s/NR}')
echo "A média das relações entre os tamanhos é de ${mean}." >> results.txt
```