Algoritmos II - Trabalho prático I

Gabriella de Lima Araujo

Índice

Introdução

Este trabalho tem como objetivo aplicar conceitos de manipulação de sequências vistos em sala de aula, além de um problema relacionado à compressão de arquivos. Para realizar a compressão/descompressão de tais arquivos, foi escolhido o método LZW (Lempel-Ziv-Welch), que é baseado em dicionários e, basicamente, substitui strings que se repetem no texto por códigos. Além disso, o dicionário utilizado no método não é nativo de nenhuma linguagem, ele foi implementado como uma árvore Trie compacta.

Máquina e Especificações

O trabalho foi implementado utilizando o sistema operacional **Pop_OS**, 16 GB de RAM e um processador **Intel Core i5 de 11ª geração**. Foi também utilizado o Python (versão 3.10.12).

A escolha pelo Python se deu pelo fato da oportunidade de implementar algo mais complexo nesta linguagem, além da facilidade proporcionada por algumas funções existentes. Um exemplo disso é o seguinte trecho de código extraído do projeto:

```
numero = 6

representacaoBinaria = format(numero, '012b')

print(representacaoBinaria)

000000000110
```

Neste trecho, estamos convertendo um número para sua representação binária de 12 bits.

O método LZW

O LZW (Lempel-Ziv-Welch) é um algoritmo de compressão de dados baseado nos conceitos do algoritmo LZ78 (desenvolvido por Abraham Lempel e Jacob Ziv em 1978). É um algoritmo utilizado para compressão de texto, sendo implementado em formatos como GIF e TIFF.

O algoritmo LZW funciona construindo um dicionário de padrões recorrentes à medida que lê os dados. Esses padrões são substituídos por códigos que representam as sequências repetitivas, o que resulta em uma compressão eficiente.

Escolhas de projeto

Representação do fim de uma string na trie

Diferente do que foi apresentado em sala de aula, optei por implementar a estrutura da trie de uma forma que permita representar finais de strings (códigos) não apenas nas folhas, mas, também, em nós internos, pois não encontrei uma maneira satisfatória de indicar o final de uma string usando símbolos arbitrários.

Uma implementação que "puxasse" nós para o final da trie, utilizando símbolos especiais, poderia causar inconsistências, já que nosso alfabeto inicial é composto por todos os caracteres da tabela ASCII estendida.

Portanto, a solução adotada foi permitir que qualquer nó dentro da trie possa representar o final de uma string, sem depender de ser uma folha.

Inserção no dicionário

Quando pensamos na implementação do dicionário como uma árvore **trie**, em que o número máximo de códigos presentes na trie foi limitado, temos duas possíveis implementações:

1. Quando chegamos ao limite, podemos simplesmente parar de inserir novos códigos e utilizar somente os que já temos no processo de compactação.
2. Quando atingimos o limite de códigos, podemos recomeçar a trie com as codificações já feitas.

Por questões de complexidade, preferi implementar a forma 1.

Além disso, ao iniciar uma instância dos problemas, uma trie contendo todos os símbolos do alfabeto ASCII estendido é criada. Neste projeto, cada símbolo foi representado em sua forma binária e, por esse motivo, cada nó da trie pode ter até dois filhos: 0 ou 1.

```
class NoTrie:
    def __init__(self):
        self.descendentes = [None] * 2
        self.prefixo = ""
        self.codigo = None
```

Processo de codificação

Em ambas as implementações, o dicionário é uma árvore trie que já foi iniciallizada com todos os 256 símbolos do alfabeto ASCII estendido em sua representação binária.

A entrada a ser codificada pode ser tanto um arquivo `.txt` quanto um arquivo de imagem `.bmp` (BitMap) que não estejam comprimidos.

Neste projeto, quando uma seqência é comprimida, ela é salva em sua representação binária em um arquivo `.bin`.

Processo de decodificação

Para realizar a decodificação em ambas as implementações, assumimos que os códigos estão em um arquivo `.bin`. A decodificação é escrita no arquivo de saída especificado na linha de comando e os códigos são escritos na sua representação binária.

Geração dos relatórios

A geração dos relatórios ao fim da execução do programa é opcional, caso se deseje gerar eles, a tag `--testes` precisa ser informada na linha de comando.

O relatório é gravado em um arquivo `.txt` e os gráficos gerados em um arquivo `.png` na pasta relatorio.

Implementações

Trie compacta

A implementação da Trie compacta foi feita de forma completa, com as funções inserir, remover uma string, buscar um código e buscar uma string, getTamanho, além da função imprimir que foi de grande auxílio durante a implementação. Ela foi muito utilizada para garantir que as remções, inserções e junção dos prefixos foi feita de forma correta.

Implementação fixa

Na implementação utilizando uma abordagem fixa de bits do método LZW, o número de bits foi fixado em 12 e é possível representar até 2^{12} códigos, essa quantidade máxima foi limitada como

```
quantidadeMaxCodigos = pow(2, 12)
```

Durante a execução do programa sempre é verificado se ultrapassamos ou não essa quantidade, para garantir que a árvore trie não cresça mais que o estabelecido.

Para realizar a decodificação, é lido um arquivo codificado e ele é separado em blocos de 12 bits, para que no processo de decodificação os símbolos corretos sejam gerados.

Implementação variável

Na implementação utilizando uma abordagem variável de bits do método LZW, o número de bits pode aumentar de acordo com a necessidade e, consequentemente, o número máximo de códigos também. Nessa abordagem, o dicionário ainda é iniciado contendo todos os símbolos do alfabeto ASCII, mas agora eles são representados com 9 bits. À medida que o dicionário é criado, é verificado se a quantidade máxima de códigos no momento atual na trie foi ultrapassado ou não e, caso essa quantidade atual tenha sido ultrapassada, aumentamos em um bit o formato de representação dos símbolos e a quantidade máxima de códigos na trie passa a ser $2^{quantidadeAnterior+1}$. É importante salientar que, mesmo aumentando na medida do necessário, há um limite superior geral para o crescimento da trie.

Esse controle e mudança podem ser observados abaixo:

```
tamanhoMaxCodigos = maxBits
tamanhoAtual = 9 # inicialmente 9 bits
quantidadeMaxCodigos = pow(2, self.tamanhoMaxCodigos)

if codigosInseridos >= pow(2, tamanhoAtual) and tamanhoAtual < tamanhoMaxCodigos:
                        tamanhoAtual += 1       # verifica se ultrapassamos a quantidade máxima do tamanho atual. Se sim e a quantidade mmáxima geral não ter sido

codificacoes.append(format(codigoPrefixo, f'0{tamanhoAtual}b')) # agora as codificações são representadas no formato binário com tamanhoAtual como quantidade de b
```

Principais diferenças na implementação das duas abordagens

A principal diferença entre as abordagens fixa e variável está na maneira como os bits e as strings são manipulados. Na implementação fixa, os símbolos iniciais do dicionário são representados como strings de 8 bits, enquanto, na abordagem variável, eles são representados com 9 bits. O mesmo padrão é aplicado aos arquivos de entrada.

Na função de codificação da implementação fixa, a entrada é uma lista onde cada posição representa um caractere codificado com 12 bits. Já na implementação variável, cada caractere é codificado inicialmente com 9 bits.

Nas funções de decodificação, a implementação fixa recebe uma lista em que cada posição contém um símbolo codificado com 12 bits. Em contraste, a implementação variável trabalha com uma lista de uma única posição, que contém toda a codificação do arquivo em sequência. Durante o processo de decodificação, o tamanho do símbolo é ajustado dinamicamente. Dependendo do valor da variável `self.tamanhoAtual`, os primeiros bits referentes ao símbolo atual são extraídos e removidos da lista. À medida que a decodificação avança, a quantidade de bits utilizados aumenta conforme necessário.

Resultados

1. Compactação de Arquivos Pequenos

1.1 Não há compactação ou a compactação é insignificante.

Neste exemplo, o arquivo `entrada.txt`, que continha a string `abbababac` e tinha um tamanho de 9 bytes, não apresentou redução no tamanho ao executar o código com o comando `python3 program/main.py codificar entrada.txt codificacao.bin fixo --testes`. Como podemos observar no relatório gerado, a compactação foi inexistente, sem diferença no tamanho dos arquivos.

Relatório de um exemplo em que não há compressão.

Além disso, ao observar o gráfico, notou-se um padrão muito irregular de leitura e gravação de bytes.

Gráfico da taxa de compressão ao longo do tempo em que não há compressão.

Por outro lado, ao executar o código com o comando `python3 program/main.py codificar entrada.txt codificacao.bin variavel --testes` utilizando a abordagem de tamanho variável, foi possível observar uma leve redução no tamanho do arquivo.

Relatório de um exemplo em há compressão insignificante.

1.2 Em alguns casos, a compressão pode até aumentar o tamanho do arquivo.

Neste exemplo, o arquivo `entrada.txt` continha o texto `az` e, ao aplicarmos o processo de compactação, observamos que, ao invés de reduzir o tamanho do arquivo, o processo acabou o expandindo: o tamanho do arquivo final foi maior do que o tamanho do arquivo original. Esse comportamento pode ser explicado pelo fato de que, para textos muito curtos ou simples, os algoritmos de compressão podem adicionar uma sobrecarga de dados ao tentar compactar informações que já estão em um formato bastante eficiente.

Relatório de um exemplo em que há expansão.

Esse efeito pode ser claramente visualizado no gráfico da taxa de compactação ao longo do tempo, em que a taxa de compressão diminui, o que indica que a quantidade de dados gravados foi superior à quantidade de dados lidos.

Gráfico de um exemplo em que há expansão.

2. Comparação entre Abordagens de Compressão

2.1 Espaço utilizado pela compressão

Ao executarmos ambas as implementações sob uma mesma entrada (para gerar esses relatórios, um arquivo de entrada de 92.6 Kb foi utilizado), é notório que a abordagem variável requer menos espaço. Isso se dá pelo fato de que iniciamos os nossos códigos com 9 bits e vamos acrescentando conforme necessário.

Relatório abordagem fixa.

Relatório abordagem variável.

2.3 Taxa de compressão final

Analisando os relatórios mencionados acima, podemos observar que, na implementação variável, há um leve aumento no desempenho em comparação à taxa de compressão final.

2.2 Tempo de compressão e descompressão

Além disso, a abordagem variável apresentou um tempo de execução ligeiramente inferior ao da abordagem fixa. Isso pode ser atribuído ao fato de que, no início do processo, a implementação variável lida com strings menores, o que pode resultar em um desempenho inicial mais ágil.

2.4 Taxa de compressão ao longo do processo

Nos gráficos a seguir, podemos observar a taxa de compressão durante o processo de compressão de uma mesma entrada, utilizando as duas implementações: o primeiro gráfico se refere à implementação fixa, enquanto o segundo à implementação variável.

Embora, à primeira vista, não seja perceptível uma grande diferença entre os dois gráficos, é interessante notar um padrão comum em ambos. Inicialmente, há uma taxa de compressão muito baixa, seguida por um "boom" na compressão para então estabilizar-se em um valor mais constante. Esse comportamento sugere que, após uma fase inicial de ajustes, ambos os algoritmos atingem uma eficiência de compressão mais estável.

Gráfico da taxa de compressão ao longo do tempo implementação fixa Gráfico da taxa de compressão ao longo do tempo implementação variável

3. Tipos de textos

Abaixo, estão as imagens dos relatórios gerados para duas entradas distintas, que inicialmente possuíam, aproximadamente, o mesmo tamanho. Um dos textos consistia em palavras aleatórias (geradas por um gerador de Lorem Ipsum), enquanto o outro era um texto com conteúdo com sentido.

Relatório palavras aleatórias.

Relatório texto com sentido.

Ao analisar os resultados, podemos observar que o texto com linguagem natural obteve uma taxa de compressão mais alta em comparação com o texto gerado aleatoriamente. É possível que isso ocorra pelo fato de que o texto com sentido possui padrões mais previsíveis, o que facilita a compressão. Já o texto com palavras aleatórias, por não seguir uma estrutura lógica, pode apresentar uma distribuição mais dispersa de dados, dificultando a compressão de forma eficiente.

4. Tipos de Dados

4.1 Imagens vs. textos

Ao utilizar o comando ´python3 program/main.py codificar myMelodyImage.bmp codificacao.bin fixo –testes´ e codificar uma imagem '.bmp' da personagem My Melody de tamanho aproximado de 353 Kb, foi gerado o seguinte relatório:

Relatório My Melody

Ao codificarmos uma entrada textual de aproximadamente 3 Mb com o comando ´python3 program/main.py codificar entrada.txt codificacao.bin variavel –testes´ o relatório gerado foi:

Relatório texto grande

A partir da análise desses relatórios, podemos observar que, apesar do arquivo de texto ser significativamente maior que o arquivo de imagem, o processo de codificação do texto foi realizado em muito menos tempo. Isso pode ser atribuído ao fato de que a leitura de arquivos de imagem envolve maior complexidade.

Além disso, foi possível observar que a taxa de compressão final do arquivo ´.bpm´ foi muito mais eficiente em comparação ao arquivo de texto.

Memória no geral

Para uma verificação do consumo de memória durante a execução do código, utilizei uma biblioteca python chamada my_profiler e ´@profile´ para controlar a memória. Abaixo, temos um exmeplo de um dos relatórios gerados:

Relatório profile

Podemos observar que o uso de memória é mais alto nas linhas em que os dados são adicionados a estruturas (como em codificacoes.append() e taxaCompressao.append()). Isso faz sentido, já que essas operações acumulam valores ao longo das iterações, o que aumenta a quantidade de memória necessária.

além disso, quando um valor é inserido no dicionário (linha 52-54) também há um aumento no uso da memória, o que sugere que o dicionário pode esta se expandindo em termos de tamanho à medida que mais prefixos são codificados.

6. Compressão vs Descompressão

Durante os testes, observei que o processo de descompressão é consideravelmente mais complexo e demanda muito mais tempo em comparação com o processo de compressão. Em casos de entradas grandes e mais complexas, o tempo de descompressão foi tão significativo que se tornou inviável gerar um relatório.

Entretanto, ao testar com uma entrada pequena, em que foi possível gerar o gráfico e o relatório é possível notar um comportamento em que lemos menos bits do que escrevemos. Na verdade, esse comportamento é o comportamento esperado para a descompressão, ou seja, o inverso do esperado na ocmpressão.

Gráfico descompressão

Dificuldades

Durante a implementação deste projeto, uma das maiores dificuldades foi representar a codificação de uma maneira que realmente mostrasse uma compressão efetiva. Inicialmente, a saída da codificação — a representação binária dos códigos — estava sendo gravada em um arquivo `.txt`. No entanto, essa representação binária estava no formato de *string*. Como resultado, cada `0` e `1` era tratado não como bits, mas como caracteres. Isso significa que, na prática, cada `0` e `1` estava sendo representado por 8 bits.

Por exemplo, considerando a string original `abbababac`, que tem 9 caracteres, o tamanho total seria 9 bytes, já que cada caractere ocupa 8 bits (ou seja, 72 bits no total).

Após aplicar a codificação utilizando uma implementação com códigos de tamanho variável, a sequência codificada ficou assim (sem os espaços): `001100001 001100010 001100010 100000000 100000011 001100011`

Consegui reduzir a sequência original de 9 para 6 "dígitos" codificados. No entanto, devido ao fato de que cada `0` ou `1` ainda estava sendo representado por 8 bits, o tamanho real da codificação não melhorou muito. Cada dígito (`0` ou `1`) é armazenado como um caractere de 8 bits, o que significa que cada código gerado ocupa 72 bits. Com 6 códigos, o total é de 432 bits, o que equivale a 54 bytes.

Portanto, mesmo que a quantidade de códigos tenha diminuído, o tamanho final em bits não estava refletindo uma compressão eficiente.

Para contornar essa situação, utilizei uma biblioteca Python chamada bitString e foi possível gravar a codificação de uma forma que deixasse explícito a compressão.

Além disso, seria muito útil ter registrado a saída decodificada durante o processo de descompressão em sua representação como caracteres (especialmente para textos), pois isso facilitaria a visualização do resultado e tornaria o processo mais intuitivo. No entanto, encontrei dificuldades ao tentar criar um método que funcionasse de maneira eficaz tanto para a abordagem fixa quanto para a variável.

Como rodar o código

1. Clone o repositório

Primeiro, clone este repositório para a sua máquina:

```
git clone git@github.com:hellolima/LZWCompressionAlgorithmProject.git
```

2. Pré-requisitos

Para codificar, tenha um arquivo `entrada.txt` ou um arquivo `.bmp` no seu repositório. Para decodificar, tenha um arquivo `codificacao.bin` no seu repositório.

3. Codificar arquivos

Usando a implementação fixa (onde o número de bits é fixado):

```
python3 program/main.py codificar entrada.txt codificacao.bin fixo
```

Gerar relatório e gráfico de estatísticas:

```
python3 program/main.py codificar entrada.txt codificacao.bin fixo --testes
```

Usando a implementação variável (onde o número de bits pode variar, aqui utilizamos limite máximo de 12 bits):

```
python3 program/main.py codificar entrada.txt codificacao.bin variavel --bits 12
```

Gerar relatório e gráfico de estatísticas:

```
python3 program/main.py codificar entrada.txt codificacao.bin variavel --bits 12 --testes
```

Nota: Não é obrigatorio informar a quantidade máxima de bits. Caso não seja informada, o valor padrão será de 12 bits.

4. Decodificar arquivos

Usando a implementação fixa:

```
python3 program/main.py decodificar codificacao.bin saidaDecodificada.txt fixo
```

Usando a implementação variável (com limite máximo de 12 bits):

```
python3 program/main.py decodificar codificacao.bin saidaDecodificada.txt variavel --bits 12
```

A flag testes também pode ser utilizada na decodificação.

5. Observações importantes

Decodificação: Um arquivo só pode ser decodificado utilizando a abordagem fixa se ele foi codificado utilizando a abordagem fixa. O mesmo vale para a abordagem variável. Limite de bits: Caso esteja utilizando a abordagem variável, atente-se ao limite de bits informado, que deve ser o mesmo utilizado durante a codificação.

Referências

https://www.youtube.com/watch?v=as3fuSWa6xs

https://pypi.org/project/bitstring/

https://www.adobe.com/br/creativecloud/file-types/image/raster/bmp-file.html

https://pypi.org/project/memory-profiler/