Trabalho Prático 1 - Algoritmos II

Luiz Philippe Pereira Amaral

[1]Departamento de Ciência da Computação - Universidade Federal de Minas Gerais (UFMG)

Abstract. *In modern computer systems, due to the fact that we often need to transport or store large volumes of data on limited-capacity storage media, is common to make use of compression algorithms. The purpose of these algorithms is to encode data in a reduced notation that can later be reverted to it's original format. In this document, we will discuss the implementation of one of these algorithms.*

Resumo. *Em sistemas de compuatção modernos, uma necessidade comum é a de compressão de arquivos, isso porque frequentemente precisamos transportar ou armazenar grandes volumes de dados em uma mídia de armazenamento de capacidade limitada. O objetivo dos algoritmos de compressão é codificar um dado em uma notação reduzida que depois pode ser revertida para o formato original. Neste documento, discorreremos sobre a implementação de um destes algoritmos.*

1. Introdução

Nossa abordagem para compressão se dará por meio do algoritmo LZ78 criado por Abraham Lempel e Jacob Ziv em 1978. Nele, atribuiremos códigos numéricos a sequências que se repetem e salvaremos sua correspondência em um dicionário que poderá ser remontado a partir da nossa saída. Saída essa que se dará por uma lista de tuplas contendo um código correspondete a uma sequência e um caractere a ser concatenado.

2. Algoritmo

Para a execução do algoritmo, usaremos uma árvore de prefixos na etapa de compressão e na descompressão, um dicionário (no nosso caso juntamente com a árvore de prefixos). Cada nó da nossa árvore será composto de um código numérico e uma sequência de caracteres. Para exemplificar, suponhamos que queremos compactar o texto "A_ASA_DA_CASA".

2.1. Compressão

Na etapa de compressão, iniciamos nossa árvore apenas com um nó raiz, que terá código 0 e sequência vazia (aqui denotada por ε), como na figura 2.1. Ao ler um caractere da entrada, pesquisaremos pelo prefixo formado por esse caractere na nossa árvore, caso ele seja encontrado, concatenamos a esse prefixo o próximo caractere lido da entrada e refazmos a busca, caso contrário, adicionaremos ao nó do último prefixo casado um filho com um novo código e com o último caractere lido da entrada, imprimimos na saída a tupla contendo o código do último prefixo encontrado e o último caractere lido da entrada e reiniciamos nossa sequência.

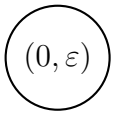

Figura 2.1.1: Estado inicial da árvore de prefixos.

No nosso exemplo, lemos o caractere ”A”e fazemos a busca por esse prefixo na nossa árvore. Como no momento a árvore contém apenas o prefixo ε, nenhum casamento será feito e adicionaremos um nó com um novo código e o caractere ”A”aos filhos da raiz, o mesmo acontece como o caractere seguinte ”_”que ainda não contém um prefixo na árvore. As tuplas (0,A) e (0,_) são adicionadas na saída, a árvore resultante é apresentada na figura 2.2. A saída nesse ponto é (0,A)(0,_).

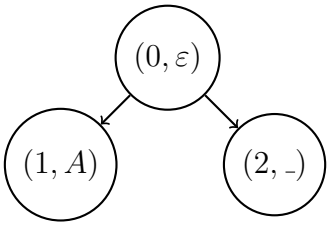

Figura 2.1.2: Árvore de prefixos após a leitura de ”A_”.

É importante que sejam atribuídos códigos aos novos nós de forma previsível, portanto, por motivos de simplicidade, faremos essa atribuição sempre de forma sequêncial.

A seguir, lemos o caractere ”A”que já foi adicionado como um prefixo da árvore, dessa forma concatenamos o próximo arquivo da entrada, no caso ”S”e refazemos a busca. Como o prefixo ”AS”ainda não existe na árvore, adicionamos um nó com um novo código e o caractere ”S”ao nó do último padrão casado, o nó (0,A), assim obtemos a árvore da figura 2.3. Em seguida imprimimos na saída a tupla que contém o código do ultimo nó casado e o último caractere lido da entrada obtendo a saída: (0,A)(0,_)(1,S).

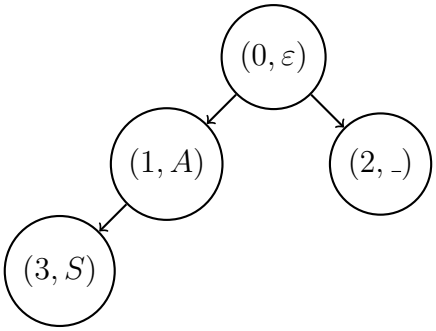

Figura 2.1.3: Árvore de prefixos após a leitura de ”A_AS”.

Então zeramos a sequência, lemos o próximo caractere e repetimos o processo. Após ler toda a entrada, teremos a árvore da figura 2.4. Nossa saída obitada será (0,A)(0,_)(1,S)(1,_)(0,D)(4,C)(3,A).

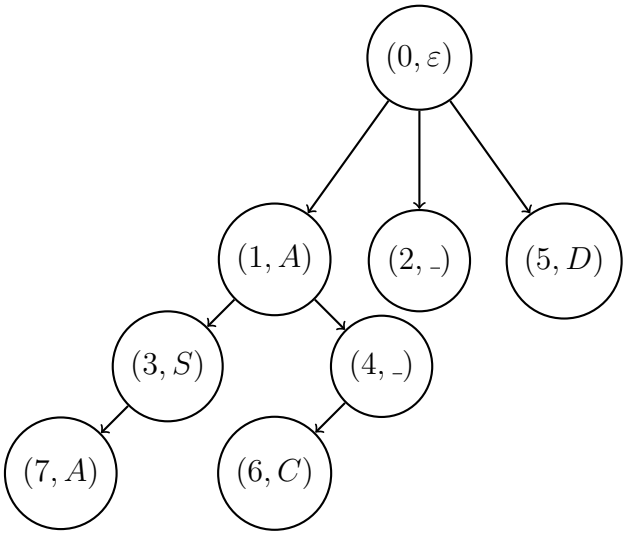

Figura 2.1.4: Árvore de prefixos de "A_ASA_DA_CASA".

2.2. Descompressão

Ao descomprimir, precisamos remontar a árvore a partir da sequência comprimida, para isso, utilizaremos a previsibilidade da distribuição do código ao ler tupla por tupla. Iniciamos lendo a primeira tupla "(0,A)", como o código é 0, sabemos que essa tupla foi adicionada a partir da raíz, portanto, apenas adicionamos uma entrada no nosso dicionário que mapeia um novo código ao caractere "A"e adicionaremos a tupla (novo código, "A") na árvore da mesma forma que fizemos durante a compressão e imprimimos o caractere "A"na saída, como estamos distribuindo códigos sequencialmente, esse valor será 1. Fazemos o mesmo para a próxima tupla "(0,_)"e obtemos a saída "A_". Em seguida lemos "(1,S)", como agora temos um código diferente de 0, precisamos fazer uma busca pelo código 1 no nosso dicionário e assim encontraremos o nó (1,A), realizando o *backtracking* desse nó até a raíz e termos que sua sequência atrelada é "A", portando, adicionamos "AS"a saída e obtemos "A_AS". Repetimos o processo até que toda a entrada tenha sido processada e por fim teremos o texto novamente em seu formato original.

3. Implementação

O programa foi implementado em C++ versão 11. Todo o código fonte desta implementação pode ser encontrado em `https://github.com/LuizPPA/smal`.

3.1. Árvore de prefixos

A representção da árvore de prefixos foi feita utilizando uma estrutura *prefix_tree* que contém um nó raíz e um tamanho armazenado. A árvore possui além de seu construtor e destrutor e dos *getters* de seus atributos, um método *prefix_tree.find(int code)* para realizar uma busca em largura por um de seus nós com o valor de código recebido pela função, um método *prefix_tree.add(node parent, string prefix)* que adiciona ao nó *parent* um novo filho com prefixo *prefix* e um novo código atribuído baseado no tamanho da árvore e um método *prefix_tree.clear()* que limpá toda a memória ocupada pela árvore.

A estrutura de um nó consiste de um número inteiro representando seu código, um caractere representando sua sequência, um ponteiro para seu nó pai e um vetor de nós filhos. Além de construtor, destrutor e *getters*, o nó possui o método *node.get_full_prefix()* que realiza o *backtracking* de seu prefixo até a raiz e retorna toda a sequência representada por aquele nó, um método *node.get_child(int i)* que busca um de seus filhos pelo código e um método *node_child(string prefix)* que busca um de seus filhos pelo seu prefixo.

3.2. Leitura e escrita

Para realizar a compressão, o programa lê a *stream* do arquivo de entrada caractere por caractere até fim executando os passos do algortimo. Todas as escritas no arquivo são feitas em formato de bytes, isso porque a string 529846 ocupa 6 bytes enquanto o número binário 529846 ocupa apenas 4 (em uma arquitetura de 64 bits, em outras arquiteturas, um número inteiro pode ocupar 1 ou 2 bytes). Além disso, com essa estratégia descartamos a necessidade de utilizar os separadores "(", ","e ")"na codificação do arquivo, já que sabemos exatamente os bytes referentes a cada dado de cada tupla, economizando 3 bytes por tupla escrita na saída. Se o arquivo de entrada chega ao fim no meio de um casamento, adicionamos a tupla (C, EOF) a saída, onde C é o código do último padrão casado e EOF é o caractere de fim de stream utilizado para indicar fim de uma sequência de leitura (geralmente representado por ^D). Esse é o caso na seqência "AAAAA"por exemplo, onde teríamos como saída "(0,A)(1,A)(2,ε)".

Ao descomprimir um arquivo, seguiremos o algoritmo descrito sobre a leitura de um arquivo binário. A leitura segue da forma como definimos a escrita, as tuplas são lidas sempre como 5 bytes (assumindo um inteiro de 4 bytes) onde os primeiros 4 representam o inteiro do código da tupla e o último byte o caractere desse código. Durante a leitura, sempre que um nó é adicionado a árvore, também salvamos sua referência em um dicionário que associa o código ao nó para que mais tarde esse nó possa ser acessado em aproximadamente $O(1)$ quando bsucarmos pelo seu código, contudo, para realiza o *backtracking* e montar a sequência desejada, utilizamos a árvore que para esse caso é ligeiramente mais eficiente que o dicionário, já que o acecsso ao ponteiro do pai de um nó é sempre $O(1)$, enquanto que a estratégia de hash do map pode ser menos eficiente em casos onde ocorrem colisões de chave.

4. Análise de Complexidade

Durante a compressão, todo o algoritmo é executado dentro de um loop que itera uma vez sobre cada caractere do arquivo de entrada, esse loop realiza uma consulta entre os filhos de um nó e, possivelmente, escreve no arquivo de saída e na árvore de prefixos. A busca nos filhos de um nó tem complexidade linear no número de filhos desse nó, no entando, o número de filhos de um nó é limitado pela quantidade de símbolos do alfabeto de entrada que é sempre finito e, na maioria das vezes, é no máximo da ordem de algumas centenas, assim dizemos que essa etapa tem complexidade contante. A escrita na árvore e na saída podem ser executadas em complexidade constante. Sendo assim, a etapa de compressão tem complexidade $O(n)$ no tamanho do arquivo de entrada.

A descompressão funciona de forma similar, uma iteração do loop principal acontece a cada tupla lida do arquivo de entrada. Cada iteração consiste em uma busca no dicionário de nós, o *backtracking* do nó até a raíz, e a escrita no arquivo de saída. A

busca no dicionário é feita em complexidade $O(1)$. O *backtracking* do nó é linear com o nível do nó árvore, esse valor necessariamente varia entre 1 e a altura total da árvore, logo flutua em um intervalo finito, assim diremos que essa etapa tem complexidade constante. Por fim, a escrita na saída ocorre em complexidade constante. Dessa forma, semelhante a compressão,a descompressão pode ser feita em complexidade $O(n)$.

5. Instruções de execução

O projeto inclui um makefile que pode ser usado para compilar o programa simplesmente com o monado *make*. Ao excutar a compilação, um arquivo executável "smal"é gerado na pasta *./build* do projeto.

Para compirmir um arquivo, basta utilizar o comando *smal -c input_file [output_file]* na mesma pasta onde se encontra o executável substituindo *input_file* pelo caminho do arquivo de entrada que se deseja comprimir, e *[output_file]* pelo caminho desejado para o arquivo de saída, caso um nome para o arquivo de saída não seja fornecido, ele será igual ao do arquivo de entrada com a extensão ".z78".

De forma semelhante, para descompirmir um arquivo, basta utilizar o comando *smal -x input_file [output_file]* na mesma pasta onde se encontra o executável substituindo *input_file* pelo caminho do arquivo de entrada que se deseja descomprimir, e *[output_file]* pelo caminho desejado para o arquivo de saída, caso um nome para o arquivo de saída não seja fornecido, ele será igual ao do arquivo de entrada com a extensão ".txt".

6. Conclusão

Podemos mesurar a eficacia de um algoritmo de compressão com a taxa de compressão que obtemos ao executá-lo, ou seja, em quanto em média o tamanho de um arquivo é reduzido ao ser comprimido. Para ter uma leve noção desse valor, podemos analizar algumas saídas.

N teste	Tamanho original	Tamanho após compressão	Taxa de compressão
1	13 bytes	35 bytes	2,7
2	524.288 bytes	809.295 bytes	1,54
3	1.048.575 bytes	966.470 bytes	0,92
4	1.124.682 bytes	761.230 bytes	0,67
5	1.048.576 bytes	1.533.630 bytes	1,46
6	2.167.737 bytes	831.105 bytes	0,38
7	1.043.005 bytes	633.195 bytes	0,60

Embora nossos casos de teste não sejam muitos, é possível ter uma visão geral do comportamento do algoritmo. Observamos que para arquivos pequenos, onde não há grande repetição de sequências, o algoritmo na verdade tornou os arquivos maiores, como visto nos teste 1 e 2. No entanto, para arquivos maiores (mais de 1 megabyte), onde é mais provavel que haja reocorrencia de sequências, a taxa de compressão média foi de 0.805, isto é, os arquivos comprimidos se tornaram 20% menores. Eventualmente, como é no caso do teste 5, observamos que mesmo arquivos grandes podem se tornar maiores após a execussão do algoritmo, isso ocorre em casos onde não há grande repetição de sequências, o que é mais comum em arquivos gerados aleatoriamente onde não há semântica no conteúdo (como foram os casos do teste).