Trabalho Prático 1 - Algoritmos II

Luiz A. D. Berto
2018047099

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

luizberto@dcc.ufmg.br

Abstract. *This work aims to implement the LZ78 compression algorithm, using a backing compact trie. During the implementation, the necessity to optimize the compact binary's protocol was realized, as well as the decayment of the compact trie to a common trie due to how the LZ78 algorithm uses it. It was possible to verify a good compression performance in the sample of natural language texts.*

Resumo. *Esse trabalho visa implementar o algoritmo de compressão LZ78, utilizando uma trie compacta como estrutura de dados de apoio. Durante a implementação, foi percebida a necessidade de otimizar o protocolo do binário comprimido, tal como o decaimento da trie compacta para uma trie comum. Pôde-se verificar uma boa performance de compressão nas amostras de textos em linguagem natural.*

1. Introdução

Algoritmos de compressão são de suma importância em vários campos da ciência da computação. Desde otimizar uso de memória de grandes conjuntos de dados, até mesmo otimizar o uso de banda de rede, eles estão presentes no dia-a-dia de praticamente toda a população mundial, seja em smartphones, computadores ou dispositivos embarcados. Nesse trabalho foi feita a implementação de um desses algoritmos, o LZ78, e também a implementação da estrutura de dados de apoio (trie compacta ou *radix tree*), o que é um ótimo exercício para se entender a importância, a complexidade e os limites da compressão de dados.

2. Implementação

O programa foi implementado em *Python3*, e é composto por seis arquivos: *common.py* e *utf8_utils.py*, onde foram definidas constantes, funções utilitárias e funções de codificações de bytes para UTF-8, além dos arquivos *node.py*, *compressed_trie.py*, *lz78.py* e *main.py*, onde foram definidas as estruturas de dados, o algoritmo e o programa principal.

2.1. Node

A classe *Node*, definida no arquivo *node.py*, representa um nó de uma *trie* compacta, tendo como membros a chave do nó (*key)*), o índice do nó no contexto do algoritmo LZ78 (*index*), e os filhos daquele nó, que são representados por um mapa de *string* para *Node*, onde a chave são os caracteres do caminho entre o nó pai e o nó filho.

Além de um construtor, essa classe possui dois outros métodos:

1. *get_child_by_first_char*: retorna o filho do nó que tem como caminho a primeira letra da chave fornecida, caso exista;
2. *split*: é utilizado pela *trie* compacta caso ela tenha que realizar uma realocação de nós, em cenários onde uma nova chave está sendo cadastrada e ela possui um prefixo em comum com o nó.

   Esses métodos são utilizados pela *trie* compacta em seu método de inserção.

2.2. Trie Compacta

A classe *CompactTrie* implementa uma *trie* compacta, que permite no máximo um índice por nó, garantindo tempos de consulta e inserção na ordem de $O(k)$, onde k é o tamanho da chave pesquisada/inserida. A classe mantém uma referência de seu nó raiz, que é um nó sentinela, e além de prover métodos de inserção e consulta, a árvore também possui o método *contains*, que retorna uma chave existe ou não.

2.2.1. Inserção

A inserção na *trie* compacta é feita de forma recursiva. Para explicar sua execução, alguns identificadores serão declarados para facilitar o compreendimento, e a operação "-" denota a remoção de um prefixo de uma *string*. Por exemplo: se a *string* for $S = aabbab$ e o prefixo for $P = aa$, então $S - P = bbab$. Em cada nível de recursão, o seguinte procedimento é realizado:

1. Verifica-se se o nó atual N possui um filho C cuja chave de transição começa com o primeiro caractere da chave K a ser inserida.
2. Caso não possua, significa que K não possui prefixo em comum com nenhuma das subárvores de N. Com isso, um novo nó é criado, e é associado à N pela chave K, e o algoritmo finaliza.
3. Caso possua, é computado o maior prefixo em comum P entre a chave de C (CK) e K
4. Se P for igual a CK, significa que todos os caracteres de CK podem ser "consumidos" na chave K, e o algoritmo prossegue recursivamente, tomando o nó C como nó atual, e a chave $K - P$.
5. Se P não for igual a CK, isso significa que o nó C terá que ser dividido, pois nem todos os caracteres de CK podem ser consumidos. Para isso, um novo nó D será criado e terá como chave o prefixo em comum P, e esse nó será filho de N. O nó D será pai tanto do nó C, que passará a ter apenas $CK - P$ como chave, quanto de um novo nó R, que terá como chave $K - P$. Nesse passo, o algoritmo cessa a recursão e finaliza.

2.2.2. Busca

A busca na *trie* compacta é feita por meio de um algoritmo iterativo, que cria um prefixo P (inicialmente vazio), e a cada caractere da chave K provida, verifica se o nó atual N possui um filho C cuja transição é dada pelo prefixo P. Caso possua, o nó atual passa a ser C, e os caracteres do prefixo são "consumidos", de forma que na próxima iteração ele possua apenas um caractere. Com isso, ao final do processamento da chave, o nó atual N

será o nó que tem como caminho a chave K, ou nulo. O método de busca retorna o índice do nó, que é o dado que o algoritmo LZ78 precisa para fazer o processo de compressão.

2.3. Contains

O método *contains* é muito similar ao método de busca, tendo como único diferença o tipo de retorno: o método *contains* retorna um booleano (True caso a chave pesquisada exista na *trie*, e False caso contrário).

2.4. LZ78

2.4.1. Compressão

O método de compressão funciona com uma janela de caracteres, que inicialmente começa com um único caractere, e é incrementada de um em um caractere. Essa janela de caracteres é pesquisada na *trie*, e caso exista, o algoritmo armazena qual o índice do nó que possui a janela. O algoritmo tenta sempre encontrar a maior janela de caracteres possível na *trie*, e só faz a compressão da janela caso ele não a encontre na *trie*. Exemplo: suponha que a janela $J = abcde$ já exista na *trie*, com índice N. O algoritmo, então, tentará incrementar essa janela com o próximo caractere do texto, digamos z, de forma que J passe a ser $abcdez$. Caso J não seja encontrada na *trie*, o algoritmo comprimirá essa janela para a tupla (N, z), que basicamente é o índice do prefixo de J que já existe na *trie*, e o último caractere (que fez com que J como um todo não existisse mais na *trie*). Sempre que uma janela de caracteres é comprimida, ela também é inserida na *trie* com um índice crescente (incrementado sempre que ocorre uma inserção na *trie*), para que possa auxiliar nas próximas compressões, e o algoritmo reinicia o tamanho da janela para 1, considerando apenas o próximo caractere não processado do texto.

Existe uma situação onde o algoritmo precisa inserir apenas o índice de uma janela: quando o final do arquivo é alcançado e a janela atual existe na *trie*. Nesse caso, não existe um caractere extra que faria com que a janela não existisse mais na *trie*. Para resolver isso, o algoritmo mantém uma flag *inserted_last*, para saber se a última janela foi ou não inserida. Caso não tenha sido, o algoritmo insere apenas o índice da janela.

O algoritmo também propõe um protocolo de codificação do arquivo, para que a descompressão possa interpretar o arquivo comprimido de forma correta. A tupla (N, z), que representa a compressão de uma janela de caracteres, é escrita num buffer de bytes, de forma que o índice N utiliza 3 bytes e portanto pode assumir valores de até $2^{24} - 1$, pois pode ser interpretado como um inteiro sem sinal, visto que os índices só assumem valores positivos. Esse valor foi escolhido pois é suficiente para mapear qualquer arquivo, mesmo em piores casos onde um arquivo tenha todos os caracteres possíveis do UTF-8). Já o caractere z utiliza de um a quatro bytes, dependendo da largura da sua codificação em UTF-8. Após processar todo o texto da entrada, o algoritmo escreve em um arquivo o buffer de bytes resultante de sua execução, em modo binário.

É importante notar que o protocolo de codificação do arquivo interfere diretamente na taxa de compressão obtida, pois o uso de mais bytes para o índice ou de bytes de *padding* para os caracteres implicaria em menor taxa de compressão, pois o arquivo comprimido teria dados desnecessários. *Nota de implementação: seria possível utilizar índices com menos do que 3 bytes, ou até mesmo fazer uso de quantidades arbitrárias de*

bits, mas isso implicaria em perda de funcionalidade para alguns caracteres especiais do UTF-8.

2.4.2. Descompressão

O método de descompressão mapeia cada tupla (N, z) para uma sequência de caracteres, onde N é a chave de uma *string* num dicionário, e z é o último caractere da sequência. Por exemplo, se a *string* $aabbcc$ estiver associada à chave N no dicionário, então a tupla (N, z) é mapeada para a sequência $aabbccz$. O algoritmo de descompressão, diferentemente do algoritmo de compressão, não utiliza uma *trie* (e sim um dicionário simples), pelo simples fato de que as chaves são números e não *strings*, e o uso de *tries* nesse caso seria inadequado, pois todas as chaves estariam no primeiro nível da *trie*, o que derrotaria o propósito de uso da *trie*.

Para cada tupla (N, z), o algoritmo pesquisa N no dicionário, e caso o valor associado V exista, então a *string* $V + z$ é concatenada numa *string* de resultado, e adicionada ao dicionário com um índice que é incrementado à cada iteração do loop principal do algoritmo.

Para construir a tupla (N, z), o algoritmo de decompressão lê 3 bytes do array de bytes lido do arquivo comprimido, e converte-os em um número inteiro com o formato *big endian*. Depois disso, são lidos mais quatro bytes (ou a quantidade de bytes remanescente no array), que formam um potencial caractere UTF-8. Na função *parse_block*, será feito o *parse* do bloco, que pode ter um caractere de 0 a 4 bytes. A verificação tamanho do caractere é feita usando máscaras de bits, de acordo com a especificação do UTF-8:

- Caso o bloco seja nulo, não existe nenhum caractere
- Caso o primeiro byte comece com 0, trata-se de um caractere de 1 byte, que é convertido por meio função *chr*, *built-in* do *Python3*
- Caso o primeiro byte comece com 110, trata-se de um caractere de 2 bytes
- Caso o primeiro byte comece com 1110, trata-se de um caractere de 3 bytes
- Caso o primeiro byte comece com 11110, trata-se de um caractere de 4 bytes.

Nos três últimos casos, o *array* de bytes é convertido com a função *decode*, *built-in* do *Python3*. Com isso, a função retorna o caractere (caso exista) e o seu tamanho l, que é usado para incrementar o ponteiro de iteração do *array* de bytes: a cada iteração, o ponteiro é incrementado de 3 (tamanho do índice) + l (tamanho do caractere lido) unidades.

Note que o caso especial é tratado de forma automática pelo algoritmo de descompressão, pois caso ele chegue ao final do arquivo e a última tupla possua apenas o índice, o próximo bloco será nulo (de tamanho 0 bytes), e o algoritmo irá tratá-lo como um caractere vazio de tamanho 0.

Ao final da execução, o algoritmo de descompressão escreve a *string* resultante no arquivo de saída, em modo texto.

2.4.3. Decaimento da trie para uma trie ordinária

Durante o desenvolvimento do algoritmo LZ78, percebeu-se que na prática a *trie* compacta decaía para uma trie ordinária devido à forma que o LZ78 a usa. Como o LZ78

trabalha com uma janela de caracteres, e tenta sempre encontrar a maior janela possível na trie, podemos classificar todas as inserções em dois casos, e com isso verificar que a *trie* nunca terá nós com múltiplos caracteres:

- O algoritmo encontra a janela de n caracteres na *trie*, portanto não a insere, o que implica que existe um caminho na *trie* que chega em um nó cuja chave é a janela de caracteres
- O algoritmo não encontra a janela de n caracteres na *trie*, o que significa que existe um caminho na *trie* para os $n - 1$ caracteres da janela (caso não existisse, a inserção já teria acontecido anteriormente), o que implica que a janela só tem 1 caractere que ainda não existe na trie.

Com isso, conseguimos verificar que todas as inserções irão criar nós de 1 caractere, o que garante que a *trie* compacta sempre decairá para uma *trie* comum. No entanto, a *trie* manterá a propriedade de que todos os seus nós são nós que contém valor, e não existirão nós "sentinela", que só servem como caminho entre um nó e outro. Isso implica que o custo de memória não é alterado pelo "decaimento".

3. Análise experimental

Para analisar a eficiência do algoritmo LZ78, foram utilizados 13 arquivos de texto de tamanho entre 100KBs e 2MBs. Esses arquivos foram obtidos no site Project Gutenberg, e são o conteúdo de texto de obras da literatura nacional e internacional. São eles:

1. Adventures of Huckleberry Finn, Complete, de Mark Twain
2. Alice's Adventures in Wonderland, de Lewis Carroll
3. Crime and Punishment, de Fyodor Dostoevsky
4. The Extraordinary Adventures of Arsène Lupin, Gentleman-Burglar, de Maurice Leblanc
5. Hamlet, de William Shakespeare
6. Metamorphosis, de Franz Kafka
7. Moby Dick; or The Whale, de Herman
8. Quincas Borba, de Machado de Assis
9. The Adventures of Sherlock Holmes, de Arthur Conan Doyle
10. The Brothers Karamazov de Fyodor Dostoyevsky
11. The Scarlet Letter, de Nathaniel Hawthorne
12. The Strange Case Of Dr. Jekyll And Mr. Hyde, de Robert Louis Stevenson
13. Viagens na Minha Terra, de João Baptista da Silva Leitão de Almeida Garrett

Para automatizar os testes, foi criado um *shell script* para executar o programa de forma automática, inicialmente comprimindo o arquivo original, e após descomprimindo o arquivo comprimido. O script, além de verificar se a descompressão resultou no arquivo original, calcula a taxa de compressão, que é definida por: $\frac{\text{Tamanho do arquivo original}}{\text{Tamanho do arquivo comprimido}}$.

As seguintes taxas de compressão foram observadas:

Título	Taxa de compressão
Adventures of Huckleberry Finn, Complete	$1,471$
Alice's Adventures in Wonderland	$1,247$
Crime and Punishment	$1,565$
The Extraordinary Adventures of Arsène Lupin	$1,341$
Hamlet	$1,254$
Metamorphosis	$1,273$
Moby Dick; or The Whale	$1,512$
Quincas Borba	$1,411$
The Adventures of Sherlock Holmes	$1,431$
The Brothers Karamazov	$1,646$
The Scarlet Letter	$1,423$
The Strange Case Of Dr. Jekyll And Mr. Hyde	$1,235$
Viagens na Minha Terra	$1,371$

Podemos perceber que o algoritmo teve uma boa performance para essa amostra de dados. Contudo, deve-se salientar que a performance do algoritmo não pode ser generalizada para qualquer tipo de dado: na verdade, a performance depende da entropia dos dados fornecidos como entrada. Testes feitos com arquivos gerados por um gerador de *lorem ipsum* produziram taxas de compressão próximas de 4, pois esses geradores de arquivos possuem baixa aleatoriedade. Por outro lado, testes com arquivos de fontes pseudoaleatórias (como */dev/random*) produziram taxas de compressão menores que 1.

4. Conclusão

Conclui-se que algoritmos de compressão são bastante interessantes, tanto do ponto de vista teórico quanto do ponto de vista prático, e o LZ78 é um grande exemplo de como estruturas de dados podem potencializar a eficiência de algoritmos.

References

(2021). Project gutenberg. `https://www.gutenberg.org/`. Acessado em: 2021-02-02.