Trabalho Prático 3 - Estruturas de Dados

Luiz Philippe Pereira Amaral [1]

[1] Departamento de Ciência da Computação – Universidade Federal de Minas Gerais

luizphilippe@dcc.ufmg.br

Abstract. *Rick Sanchez's data compression problem revolves around the need to, given a data set, get an encoding that causes more frequent data to be assigned to a smaller encoding, while less frequent data is assigned to a larger encoding. This way we can store and transport this data set represented in a smaller space. The goal of this project is to create a stable and efficient algorithm to solve this problem so that for the same input, the same output is always produced in a timely manner.*
Keywords: *Data Structures, Binary Tree, Huffman Tree, Hash, C ++.*

Resumo. *O problema da compressão de dados de Rick Sanchez gira em torno da necessidade de, dado um conjunto de dados, obter uma codificação que faça com que dados mais frequentes sejam atribuídos a uma codificação menor, enquanto os menos frequentes recebam uma codificação maior. Dessa forma podemos armazenar e transportar esse conjunto de dados representados em um espaço menor.*
O objetivo desse projeto é criar um algoritmo estável e eficiente para resolver esse problema de forma que, para uma mesma entrada, a mesma saída seja sempre produzida em tempo hábil.
Palavras-chave: *Estruturas de Dados, Árvore Binária, Árvore de Huffman, Hash, C++.*

1. Implementação

A implementação do código foi feita em inglês devido ao benefício de se tratar de um idioma que não utiliza acentos.

Para a solução do problema foram implementados um *Hash* - que internamente utiliza uma lista *List::LinkedList* para o tratamento das colisões - e uma árvore binária (*BinaryTree*). O *hash* armazena os índices de palavras e torna o acesso a elas eficiente enquanto a árvore facilita a distribuição dos códigos. Também foram implementadas classes que auxiliam na representação das entidades do sistema, mais especificamente dos índices de palavras (*Index*) e *DataCompressor*.

1.1. Lista Encadeada

A classe lista contém um inteiro que armazena a quantidade de objetos inseridos na lista e dois ponteiros "first" e "*last*" que apontam para duas células, a primeira e a última da lista respectivamente. A lista possui um método "*get*" que retorna seu tamanho. A adição de objetos é feita por padrão no fim da lista com o método "*add*" (complexidade *O(1)*) e a remoção é feita por meio de um índice i recebido na função "*remove*" (complexidade *O(n)*

para o pior caso e *O(1)* para o melhor, sendo os melhores casos aqueles onde i é igual a primeira ou a última posição da lista). Também é possível inserir ou remover em qualquer posição da lista com complexidade *O(1)* caso seja fornecida uma referência para uma célula utilizando as funções "*insert_after*", "*insert_before*" e "*remove*(*Cell*)". É possível pesquisar células e objetos na lista com complexidade *O(n)*. A lista fornece um método "*each*" que recebe uma função *callback* e a executa para cada item presente. A lista pode ser liberada de forma recursiva e iterativa.

1.2. Hash

A classe *Hash* possui um inteiro *size* para seu tamanho inicializado com 0, um inteiro *max_size* inicializado com 255 que contém o tamanho do vetor de listas do hash e o vetor de listas propriamente dito onde cada posição é inicializada como uma lista vazia. As listas do vetor armazenam o tipo *HashPair*, que contem uma cadeia de caracteres (*char **) e um objeto de tipo genérico associado. As operações do hash são construídas ao redor dos métodos *get* e *add*. *Get* recebe uma cadeia de caracteres e retorna o objeto associado com complexidade *O(1)* no melhor caso e *O(n)* no pior, caso o objeto com a chave recebida não exista, a função retorna *nullptr*. *Add* recebe uma cadeia de caracteres e um objeto e adiciona esse objeto associando-o a string recebida com complexidade *O(1)*. O hash sobrescreve o operador *[ ]*, de forma que acessar hash["chave"] é o mesmo que fazer um get com a chave especificada, no entanto, por meio deste operador, um novo objeto é criado no para a chave recebida caso ele ainda não exista. O hash também fornece um método "*each*" que similar ao método da lista recebe uma função *callback* e a executa para cada item presente.

1.3. Árvore

A árvore utilizada é uma árvore binária de busca montada a partir de um inteiro *size*, um inteiro *height*, um inteiro *leaves* e um ponteiro para o tipo *Node* que representa a raiz da árvore. O tipo nó (*Node*) aponta para três outros nós: *left*, *right* e *parent*, que são respectivamente o galho a esquerda, a direita e o nó pai. Cada nó também armazena um objeto de tipo genérico. Os principais métodos da árvore são *add* (complexidade *O(logn)* no melhor caso e *O(n)* no pior), e gets que retornam o maior ou menor elemento da árvore. O método *add* recebe um objeto de tipo genérico e uma função callback que retorna um booleano, essa função é utilizada para comprara qual é o maior entre dois objetos da árvore. A árvore tem um método "*each*" que recebe um *callback* e percorre os elementos em pré-ordem aplicando o *callback* para cada elemento.

1.4. Resolução do problema

Inicialmente, um *Hash* é criado para guardar elementos do tipo *Index*. A leitura das palavras é feita uma por uma acessando a sua posição no hash e incrementando o contador do índice, em seguida partimos para a montagem da árvore de Huffman. Na função *DataCompressor::build_tree* começamos criando uma lista de árvores de índices, cada uma contendo apenas um nó, então iteramos por essa fila e encontramos as duas árvores cujas as raízes têm o índice com o menor número de ocorrências, caso o número de ocorrências seja o mesmo, verificamos o número de folhas da árvore (também armazenado no índice) e caso esse número também seja igual, selecionamos o menor em ordem alfabética da palavra do índice. Então, tendo as duas árvores*a* e *b* as menores da lista,

removemos tais árvores, criamos um novo nó, cujo número de folhas do índice é igual a soma do número de folhas de a e b (com $a < b$), o número de ocorrências é a soma do número de ocorrências de a e b, e a palavra é a palavra da árvore a. Adicionamos a esquerda desse nó a árvore a e a direita a árvore b e então montamos uma nova árvore tendo o nó recém criado como raiz. Repetimos esse processo até que a lista e árvores contenham apenas um elemento. Por fim, percorremos toda a árvore com o método each atribuindo os códigos as nós. Tais códigos são designados de acordo com o caminho percorrido até o nó fazendo esquerda = 0 e direita = 1. Agora temos os índices finais armazenados no hash criado ao início da execução e podemos simplesmente ler as entradas e acessar as saídas.

Finalmente liberamos todos os índices contidos na árvore (todos os índices criados), liberamos a árvore em si e o hash criado.

2. Instruções de compilação e execução

O makefile preparado para a execução do código tem as seguintes opções:

- make: Compila os arquivos de código fonte e armazena a saída da compilação em uma pasta "./build/".
- make run: Executa o arquivo gerado pela compilação para entrada manual.
- make mem: Executa o arquivo gerado pela compilação em modo de depuração de memória, assim podemos nos certificar que não existe nenhum vazamento acontecendo durante a execução.
- make test: Executa os script de testes do trabalho.
- make clean: Limpa a pasta contendo as saídas da compilação.

3. Análise de complexidade

A leitura das entradas e atribuição no hash tem complexidade *O(n)* já que fazemos o acesso ao hash com complexidade *O(1)* $\sim$ *O(n)*. A montagem da árvore de Huffman depende de uma passada completa pela lista de árvores e sucessivas passadas completas pela fila (agora com tamanho reduzido em i elementos na i-ésima passada) então temos uma complexidade de (quase) $O(n^2)$. E por fim, fazemos a atribuição dos códigos passando uma vez por cada elemento da árvore, que contém agora cerca de $nlog(n)$ elementos, portanto, complexidade de $O(nlog(n))$. Assim, ficamos com função de complexidade: $T(n) = n + nlog(n) + n^2$.

Como $f(x) + g(x) = O(max(f,g))$, nossa ordem de complexidade é de $O(n^2)$.

4. Conclusão

O acesso ao hash é eficiente mesmo para grandes volumes de dados, desde que esses dados tenham uma organização que favoreça a criação de uma chave numérica de forma que minimize o número de colisões ocorridas. Já a árvore, apesar de menos eficiente, pode ser aplicada de forma mais ampla e oferece vantagens enormes quando se trata de visitar elementos de forma ordenada.

5. Bibliografia

https://www.ime.usp.br/~song/mac5710/slides/05tree.pdf