Relatório – Trabalho 2

Aluno: Matheus Salgueiro Castanho
Engenharia de Computação
Estrutura de Dados I
Profª. Patrícia Dockhorn Costa
Universidade Federal do Espírito Santo

I – Introdução

Como forma de avaliação para a disciplina de Estrutura de Dados I, foi solicitados aos alunos que implementassem, em linguagem de programação C, dois programas: um compactador e um descompactador de arquivos. Esses deveriam ser baseados no Algoritmo de Huffman.

Um compactador basicamente é um programa que, a partir de um arquivo de entrada, produz um arquivo de saída que contém as informações do arquivo inicial, mas que ocupa um menor espaço na memória do computador. O descompactador é o caminho inverso. A partir do arquivo compactado ele é capaz de gerar o arquivo de origem, com sua informação e tamanho originais.

O Algoritmo de Huffman é um algoritmo desenvolvido por David A. Huffman em 1952, que tem o intuito de criar um tabela de compactação para um arquivo baseado na frequência de repetição de seus caracteres.

Basicamente, o algoritmo calcula o número de repetições de cada caracter no arquivo e cria uma lista com os caracteres e suas respectivas frequências em ordem crescente das frequências (pesos). A partir dessa lista, cria-se uma árvore binária cujas duas sub-árvores sejam os primeiros caracteres da lista de frequências (os dois com menor peso), e cujo peso é a soma dos pesos de suas sub-árvores. Essa árvore é inserida na lista em ordem crescente do peso. Esse processo é repetido até que a lista tenha tamanho unitário.

Esse último elemento restante da lista é uma árvore com todos os caracteres do arquivo original, posicionados de acordo com o seu peso. Por exemplo, caracteres com maior peso estarão mais próximos da base, enquanto caracteres com menor peso estarão em níveis mais profundos da árvore, mas sempre como folhas.

Baseado nessa árvore, denominada Árvore de Huffman, é possível determinar uma nova codificação binária para os caracteres em função do caminho da raíz até as suas respectivas folhas. Dessa forma, caracteres com maior peso terão uma codifição menor do que a sua codificação natural na Tabela ASCII. Reescrevendo o arquivo de origem utilizando essa nova codificação leva a um arquivo de tamanho menor que o original, que é o arquivo compactado.

O que o compactador faz é montar a Árvore de Huffman para o arquivo de entrada e então reescrever toda a informação em função da codificação obtida através da Árvore de Huffman. O descompactador, por sua vez, tem o caminho inverso, tendo a Árvore de Huffman ele traduz a informação compactada novamente para a codificação da Tabela ASCII, gerando o arquivo de origem.

II – Implementação

Para a implementação do programa, foram utilizadas 3 estruturas de dados:

- Árvore Binária;
- Lista de Árvores Duplamente Encadeada com Sentinela;
- Nó da Lista de Árvores;

A estrutura Árvore Binária (*Arv*) tem os seguintes campos:

Tipo	Nome
unsigned char	*item*
unsigned long long int	*peso*
*unsigned char**	*codigo*
int	*tamCod*
*Arv**	*filhaEsq*
*Arv**	*filhaDir*

No programa, o campo *item* armazena um caracter do arquivo de origem, o campo *peso* a sua frequência, em *codigo* é armazenada a nova codificação do caracter (baseada na Árvore de Huffman), *tamCod* guarda o tamanho dessa nova codificação, e *filhaEsq* e *filhaDir* armazenam ponteiros para as sub-árvore esquerda e direita, respectivamente.

Já a Lista de Árvores (*ListaArv*) consiste da seguinte estrutura:

Tipo	Nome
*NoArv**	*pri*
*NoArv**	*ult*
int	*tam*

Onde *pri* e *ult* correspondem ao primeiro e último nós da lista, respectivamente, e *tam* armazena o tamanho atual da lista.

A estrutura Nó de lista (*NoArv*) foi implementada da seguinte maneira:

Tipo	Nome
*Arv**	*arv*
*NoArv**	*prox*
*NoArv**	*ant*

Onde *prox* é um ponteiro para o próximo nó da lista e *ant* é um ponteiro para o anterior. O campo *arv* armazena um ponteiro para uma estrutura do tipo *Arv*.

Foram implementadas as funções básicas de cada um desses TADs que seriam necessárias para o desenvolvimento do projeto, como funções de criação, destruição, acesso, alteração e inserção e remoção, no caso da lista. Todos os TADs foram implementados utilizando-se o conceito de tipo opaco.

Além desses TADs, utilizou-se também o TAD bitmap, de autoria do Prof. João Paulo Almeida, que é uma representação de um mapa de bits. À estrutura *bitmap* foi adicionado um campo extra, denominado *lengthAgregado.* A função *bitmapInit* foi alterada para alocar um bitmap dinamicamente, ao invés de estaticamente, como havia sido implementado. Outras funções também foram adicionadas a esse TAD. Para mais informações a respeito dessas funções, favor conferir os arquivos *bitmap.c* e *bitmap.h*. A estrutura bitmap então tem o seguine formato:

Tipo	Nome
unsigned int	*max_size*
unsigned int	*length*
*unsigned char**	*contents*
unsigned int	*lengthAgregado*

Todo o trabalho está dividivo em oito arquivos:

- TadArvore.c
- TadArvore.h
- TadLista.c
- TadLista.h
- bitmap.c
- bitmap.h
- Compactador.c
- Descompactador.c

A descrição de todas as funções, com inputs e outputs, pré e pós-condições estão detalhadas acima de cada função em seus respectivos arquivos '.c'.

Funcionamento do Compactador

Pode-se dividir o compactador em 4 etapas essenciais:

1. Leitura da informação do arquivo e determinação da frequência dos caracteres;
2. Montagem da Árvore de Huffman;
3. Gravação do cabeçalho do arquivo compactado.
4. Tradução da informação para o arquivo compactado.

Para a primeira etapa, utilizou-se os argumentos da função main para se obter o nome do arquivo de entrada e determinar o nome do arquivo de saída (que substitui a extensão do nome do arquivo de entrada por .comp, ex: o arquivo compactado de arquivo.txt se chamará arquivo.comp). Logo após, utilizou-se um vetor de frequências, como orientado pela professora, para determinar a frequência de cada caracter no arquivo de origem.

Com o vetor de frequências em mãos, partiu-se para a segunda etapa. Para cada caracter do vetor de frequências que aparecesse pelo menos uma vez no arquivo de origem foi criado um nó de lista de árvores, com seu peso, e inserido em uma lista crescentemente ordenada. A partir dessa lista, a função responsável por montar a Árvore de Huffman é a função *arv_huffman*.

Essa função executa exatamente o algoritmo de Huffman, montando a árvore a partir da lista. Porém, optou-se por não inicializar o campo *codigo* na árvore de Huffman no começo do programa. Essa escolha foi feita pois antes da montagem da árvore, não há como saber qual será o tamanho máximo do código de Huffman para um caracter. Dessa forma, determina-se a Árvore de Huffman e só depois os campos *código* de todas as árvores são inicializados pela função *arv_inicia_codigos*, que as aloca dinamicamente com o tamanho correspondente à altura da Árvore de Huffman, que é o tamanho máximo que o código de um caracter terá, e preenche o vetor *codigo* com '2'.

Uma vez que os campos *codigo* na Árvore de Huffman estão inicializados, a função *arv_codifica* é chamada. Essa função é responsável por gravar o código de cada nó folha em seu respectivo campo *codigo*, para poder ser usado em seguida no programa. Para a codificação dos caracteres foi utilizada a convenção de “0 para a esquerda e 1 para a direita” ao se andar na árvore.

Em seguinda inicia-se a gravação de informação no arquivo compactado, que tem a seguinte estrutura:

Tamanho Cabeçalho	Tamanho Informação Compactada	Tamanho Nome Arquivo	Cabeçalho	Nome Arquivo	Informação Compactada
A bits	B bits	C bits	Tamanho Nome Arquivo bits	D bits	Tamanho Informação Compactada bits

Estrutura do Arquivo Compactado

Onde,

- A = sizeof(unsigned int);
- B = sizeof(unsigned long long int);
- C = sizeof(int);
- D = Tamanho Nome Arquivo.

As primeiras informações a serem gravadas no arquivo são os tamanhos referentes ao cabeçalho, à informação compactada e ao nome do arquivo original, nessa ordem. O nome do arquivo original é compactado junto com o seu conteúdo, logo após o cabeçalho.

A função *arv_serializa* grava o cabeçalho no bitmap, que depois será escrito no arquivo compactado. Para a serialização da árvore foi utilizado '0' para nó não folha e '1' para folha. O retorno dessa função é a quantidade de bits escritos por ela, ou seja, a quantidade de bits do cabeçalho do arquivo. Portanto, nesse ponto já se tem o tamanho do cabeçalho, que então é gravado no arquivo compactado.

Para a tradução da informação, poderia ser feita uma busca na Árvore de Huffman pelo caracter que se deseja traduzir, e então gravar o código de compactação desse caracter no bitmap. Porém essa operação seria muito custosa e acarretaria em perda de eficiência do programa. Para resolver esse problema, foi criada uma lista de árvores apenas com as folhas da árvore de huffman, que nesse ponto já têm os seus devidos códigos gravados nas suas estruturas, e não dependem mais da Árvore para a tradução. Essa lista foi ordenada crescentemente pelo peso e a busca por caracteres foi feita de trás pra frente. Dessa forma, para caracteres de maior frequência, a função *tradutor_huffman* (responsável pela tradução) precisaria de poucas interações até achar o caracter

procurado e imprimir o seu código no bitmap, enquanto para caracteres de menor frequência ela varreria quase a lista toda.

É utilizando essa mesma lista de nós folha que foi determinado o tamanho da informação compactada. A função *lista_tamanho_ArqComp* passa por todos os elementos da lista e soma o produto entre o tamanho do código (*tamCod*) e o peso do caracter (*peso*) a uma variável acumuladora, que no final da execução da função terá armazenada o tamanho total da informação compactada em bits.

Em todas as funções que precisavam gravar informações no bitmap, foi utilizada a função *bitmap_verifica*. Essa função sempre era chamada antes de se escrever qualquer bit no bitmap. Ela verifica se o bitmap está cheio, e caso afirmativo, escreve toda a informação armazenada no bitmap no arquivo de saída e reinicializa o bitmap. Utilizar essa técnica poderia causar problemas se o tamanho do bitmap não fosse múltiplo de 8. Nesse caso, sempre que a informação do bitmap cheio fosse transferida para o arquivo compactado alguns bits de lixo também seriam escritos, alterando a integridade da informação do arquivo compactado. Por esse motivo optou-se por utilizar um tamanho de 262144 bits para o bitmap, que é a potência sexta de 8. Esse tamanho foi escolhido por ser múltiplo de 8 e por não ser nem muito pequeno nem muito grande. Se fosse muito pequeno, durante a compactação de arquivos grandes o bitmap teria que ser reinicializado várias vezes, acarretando perda de eficiência do programa. Se fosse muito grande, o espaço necessário para a alocação do bitmap na memória do computador poderia acarretar efeitos colaterais ou até não poder ser alocado por insuficiência de memória disponível.

Funcionamento do Descompactador

O coração do descompactador está na função *DesTraduz*, que é a função responsável por ler a informação compactada e transformá-la na informação original, de acordo com a Tabela ASCII. Mas antes de chamar a funçao *DesTraduz*, era necessário se remontar a Árvore de Huffman a partir do cabeçalho do arquivo compactado.

No início da execução, o descompactador cria um bitmap para conter o número mínimo de bytes necessários para conter toda a informação do cabeçalho. Caso o tamanho do cabeçalho não seja tenha um número inteiro de bytes, uma parte do nome do arquivo (próxima informação no arquivo compactado após o cabeçalho) é lida também. Para corrigir esse erro, posteriormente esses bits excedentes são copiados para o bitmap que será utilizado na tradução do arquivo compactado. Após inicializar o bitmap para o cabeçalho, toda a informação do cabeçalho é copiada do arquivo compactado. Nesse ponto a função *arv_deserializa* é chamada, que é a responsável pela remontagem da Árvore de Huffman.

Essa função tem um aspecto recursivo, e vai construindo a árvore de baixo pra cima. Isso é necessário pois no início da leitura do cabeçalho não se tem a informação necessária sobre as sub-árvores da árvore da base, de forma que é necessário fazer chamadas recursivas até que as sub-árvores de uma árvore já tenham sido montadas para então inicializá-la.

Por causa do caráter recursivo da função *arv_deserializa*, era necessário gravar a posição atual de leitura do bitmap em uma variável externa à função (utilizando ponteiros), para que ela pudesse ser acessada independentemente do nível da recursão atual. Para isso foi criado o campo *lengthAgregado* no bitmap. Durante a execução da função, esse campo é incrementado com o número de bits do bitmap já lidos. Assim, é possível saber qual é a próxima posição de leitura em qualquer chamada da função.

Com a Árvore de Huffman montada, a função *DesTraduz* é chamada para decodificar o arquivo compactado. Uma peculiaridade dessa função é o fato de que o arquivo de saída, ou seja, o arquivo produto da descompactação, é inicializado dentro dela. Essa decisão foi tomada por que como o nome do arquivo original foi compactado junto com o seu conteúdo, apenas no início da etapa de decodificação é que se teria acesso ao nome do arquivo original. Assim, a função começa decodificando o nome do arquivo original. Quando esse nome é obtido, o arquivo de saída é então inicializado com esse nome, e nas interações seguintes, o programa escreve os caracteres descompactados nesse arquivo.

III - Conclusão

Ambos os programas, tanto o Compactador quanto o Descompactador foram implementados com sucesso e estão anexados juntamente com esse texto. Uma das principais dificuldades enfrentadas durante a implementação dos programas foi a manipulação de bits. Como a linguagem C não oferece ferramentas para manipulação direta de bits, como tem para bytes, todo esse processo teve que ser feito indiretamente através do TAD bitmap. Muitos cuidados tiveram que ser tomados com relação à escrita e leitura em arquivos, pois poderiam conter informação inútil (lixo).

Nos testes feitos com o compactador, o tamanho do arquivo compactado foi de cerca de 55.98% do tamanho original para arquivos de texto, 99.98% para imagens (jpg e png) e para arquivos pdf houve expansão de 0.2% na maioria dos casos, e compactação de 4% em alguns outros. Na descompactação, todos os arquivos descompactados tinham exatamente o mesmo conteúdo do arquivo original. No caso de imagens, não houve perda de cor, qualidade nem nitidez. A integridade dos arquivos descompactados foi de 100% em relação aos originais.

A exemplo dos arquivos do formato pdf, o compactor expande alguns arquivos pequenos, devido à necessidade de se gravar a Árvore de Huffman serializada no arquivo compactado. Porém, esse fato é irrelevante, pois só acontece com arquivos até 110 bytes, aproximadamente (teste feito com arquivos de texto), os quais são improváveis de serem alvo de compactação no dia a dia.