***Multimídia***

**Trabalho 02**

**Grupo 04**

Caio Augusto da Silva Gomes 7239072

Fernando Cury Gorodscy 7152354

Vanessa Apolinário de Lima 7239256

**Bibliotecas**

A única biblioteca fora do padrão C-ANSI utilizada pelo grupo foi a *stdbool.h*, com o intuito de facilitar a formação dos bytes durante a compressão dos dados, pois com a variável booleana é possível utilizar comandos binários para inserção dos bits dentro do byte.

**Estrutura de Dados**

1. ***Struct Colors:***

Esta *struct* foi criada para facilitar a passagem de parâmetros entre funções, pois ao invés de enviar 3 matrizes de cores, é possível enviar apenas uma estrutura que as representa. Além disso, esta pode ser adaptada tanto para a imagem inteira, como para os blocos preparados de 8x8 para cada cor.

1. ***Struct huffman\_tree\_t:***

A estrutura utilizada por Huffman tem como utilidade criar a árvore, dessa forma, esta *struct* representa as folhas e nós da arvore, gerando uma conexão entre os mesmos para facilitar o acesso na árvore, e consequentemente, a codificação e descodificação do código de Huffman. Além disso, esta estrutura de dados preza pela generalização do algoritmo, baseando a codificação em um valor inteiro, o qual pode representar qualquer contexto desejado.

1. ***Vetor elements:***

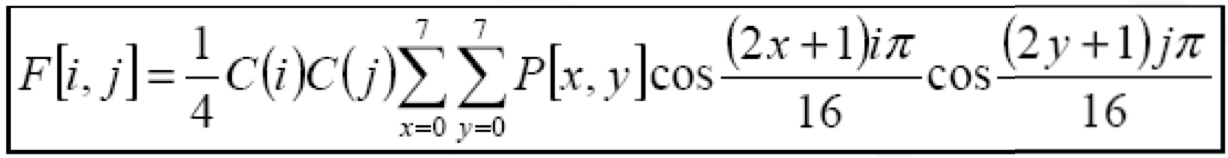
Este não pode ser considerado uma estrutura de dados, mas por sua complexidade, decidiu-se explicá-la na descrição deste projeto.

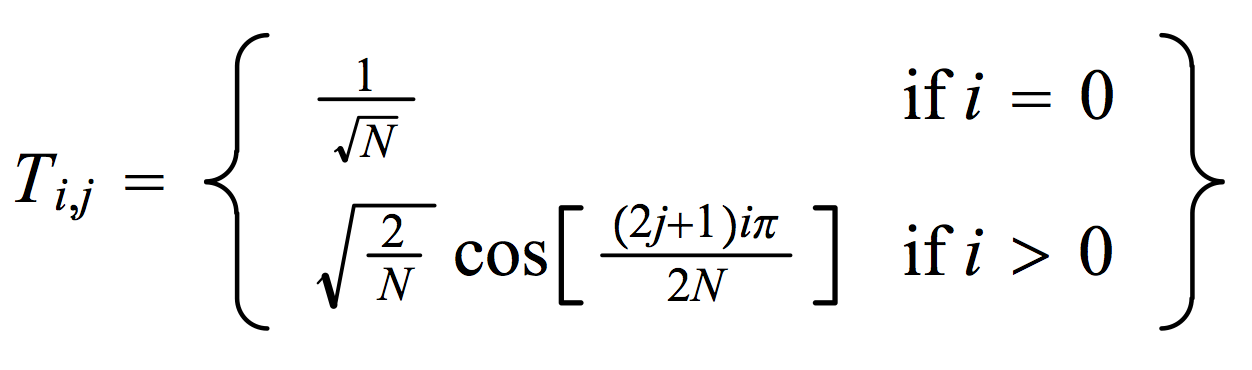
Primeiramente, é necessário explicar que no algoritmo desenvolvido, a árvore de Huffman codifica a combinação entre o número de bits da cor do pixel e o número de repetição deste valor; contudo, para que este possa ser codificado em um inteiro, foi preciso uma conversão destes valores para bits que possam gerar um inteiro.

Assim, os três primeiros bits do valor são compostos pelo número de bits da cor do pixel; o maior valor para o número de bits é 8, porém, para evitar a perda de 1 bit que seria utilizado apenas para representar o valor 8, substituímos sua representação por 0 (em binário 000) e por isso, consideramos que quando o valor da cor do pixel é zero, este possui número de bits igual à 1 (em binário 001). Os próximos 6 bits são compostos pelo número de repetições do valor da cor do pixel, o qual é limitado pelo valor 64, e da mesma forma, para que não seja perdido 1 bit, subtraí-se 1 do valor, para que este varie entre 0 e 63.

Esta combinação de bits gera um valor de 0 à 511, por isto, o vetor elements possui tamanho 512. Desta forma, o vetor elements é capaz de representar a frequência com a qual um dado par de número de bits e quantidade de repetições acontecem em uma imagem. Por fim, esta frequência armazenada pode ser passada para o algoritmo de Huffman e auxiliar na construção dos nós e da árvore em si.

1. ***Transformada Discreta de Cosenos***

Esta operação, considerando tanto a transformada (DCT) quanto a inversa (IDCT), deve ser aplicada com os valores da imagem sendo convertidos de 0 à 255 para -128 à +127. Após a conversão dos valores, a fórmula abaixo deve ser aplicada:

Contudo, conforme descrito em [1], uma alternativa para essa fórmula é aplicar uma multiplicação de matrizes. A primeira matriz é composta pela fórmula:

Dessa forma, calculando essa matriz T com N igual à 8, e sua inversa T’, basta aplicar as seguintes fórmulas para calcular a DCT e a IDCT:

DCT = T M T’ e IDCT = T’ M T

1. ***Quantização***

Para a quantização foram pesquisadas diversas tabelas, visto que as tabelas de compressão do JPEG não apresentam um bom desempenho, pois consideram a imagem em YCbCr. Assim, testando diferentes tabelas, foi encontrada as tabelas apresentadas no em [1], que consideram a imagem organizada em RGB.

Todas as tabelas no artigo apresentam um bom desempenho, contudo, umas causam maior perda de informação do que outras, devido ao não armazenamento das casas decimais; dessa forma, para manter a qualidade, foi utilizada a tabela com redução de 50%; pois essa tabela de quantização combinada com a Transformada dos Cosenos através da multiplicação de matrizes, apresenta uma precisão de 70%, conforme o artigo [1].

1. ***Armazenamento de Número Negativos***

Como a DCT gera números com altos valores tanto positivos quanto negativos, foi necessário armazenar números negativos. Todavia, em C++, é utilizado 1 byte para armazenar o sinal, desperdiçando espaço e afetando os resultados da compressão. Assim, para melhorar a compressão, ao invés de utilizar 1 byte, é utilizado 1 bit para indicar o sinal do valor. Caso o valor seja negativo, é armazenado um, caso contrário, é armazenado zero.

1. ***O(n^2) compressão e O(n) descompressão***

É interessante ressaltar que a exigência de menor tempo de execução do algoritmo é maior na descompressão do que na compressão, por isso, o algoritmo desenvolvida visa uma mais rápida descompressão, possuindo parte dos algoritmos com complexidade O(n), enquanto a compressão apresenta nesta mesma fase uma complexidade O(nˆ2).

Durante a compressão é necessário gerar a árvore de Huffman, para isso, primeiro deve-se calcular as frequências na imagem, e apenas depois percorrer a imagem novamente para que possa codificá-la. Enquanto na descompressão as frequências se encontram salvas no arquivo, sendo necessário percorrer o arquivo uma única vez para reconstruir a imagem original.

**Referências**

1. Ken Cabeen e Peter Gent, Image Compression and the Discrete Cosine Transform, College pf the Reedwoods.