UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO
http://www.icmc.usp.br/

Professores: Fernando V. Paulovich (paulovic at icmc.usp.br)
Moacir Ponti Jr (moacir at icmc.usp.br)
Aluno PAE: Felipe S. L. G. Duarte (fgduarte at icmc.usp.br) - Turma A
Monitor: Cassiano K. Casagrande (cassianokc at usp.com) - Turma B

Trabalho 02: Júpiter Not Web - Parte 2

1 Prazos e Especificações:

O trabalho descrito a seguir é individual e não será tolerado qualquer tipo de plágio ou cópia em partes ou totalidade do código. Caso seja detectado alguma irregularidade, os envolvidos serão chamados para conversar com o professor responsável pela disciplina.

A entrega deverá ser feita única e exclusivamente por meio do Sistema de Submissão de Programas (SSP) no endereço eletrônico `http://ssp.icmc.usp.br` até o dia **5 de maio de 2013 as 23 horas e 59 minutos**. Sejam responsáveis com o prazo final para entrega, o SSP está programado para não aceitar submissões após este horário e não será aceito entrega fora do sistema.

Leia a descrição do trabalho com atenção e várias vezes, anotando os pontos principais e as possíveis formas de resolver o problema. Comece a trabalhar o quanto antes para não ficar dúvidas e você consiga entregar o trabalho a tempo.

O trabalho deverá ser submetido em formato zip/Makefile contendo o arquivo principal, o Makefile e as TAD's implementadas (pares `.h .c`) para solucionar o problema. Serão avaliados não só o resultado final do sistema, mas também a identação e documentação interna.

2 Descrição do Problema:

A reunião com os responsáveis pelo "Júpiter Not Web - JNW" foi um sucesso, eles adoraram a proposta e elogiaram bastante todo o seu trabalho, parabéns! Algumas novas ideias e provas de conceito foram pedidas para a próxima reunião e como você já sabe, você é um dos programadores responsáveis pela implementação dessas novas ideias, o futuro do "Júpiter Not Web" continua dependendo de você.

A sua missão é desenvolver o modulo de compressão e descompressão de informação para a transmissão pela rede, você deverá implementar o algoritmo de *run-length* para imagens e o algoritmo de *Huffman* para texto.

Assim, elabore um sistema que servirá como prova de conceito para a utilização dos algoritmos de compressão no nosso JNW. O sistema será dividido em 2 partes, a primeira responsável pela compressão e descompressão de uma imagem e a segunda tratará de informações textuais.

3 Comandos do Sistema:

Como estamos desenvolvendo apenas um protótipo como prova de conceito, o sistema deverá funcionar como um terminal onde se aceita comandos texto e o mesmo deverá reagir de acordo com a especificação de cada comando:

- `'sair'` - Este comando deverá sair do sistema liberando toda memória possivelmente alocada e fechando corretamente possíveis arquivos abertos;
- `'compactar run-length <filename>.pgm'` - Este comando deverá **COMPACTAR uma imagem** PGM;
- `'descompactar run-length <filepath>.rl'` - Este comando deverá **DESCOMPACTAR uma imagem** que foi compactada previamente utilizando o algoritmo de run-length;
- `'compactar huffman <filename>.txt'` - Este comando deverá **COMPACTAR um texto** utilizando o algoritmo de *Huffman*;
- `'descompactar huffman <filename>.huff'` - Este comando deverá **DESCOMPACTAR um texto** tomando como base uma tabela de símbolos e bits bem como o texto compactado previamente;
- `'dump tree <filename>.txt'` - Este comando deverá exibir a arvore de huffman, percorrendo-a in-order;

4 Parte 1: Run-Length

Como descrito anteriormente, você deverá implementar o algoritmo de run-length para a compactação de uma imagem no formato PGM.

4.1 Imagem PGM[1]

A designação de formato de imagem PBM (Portable Bitmap) engloba três formatos de imagem para imagens a preto e branco, em escala de tons cinzentos e a cores, todos eles sem compressão e que apresentam uma estrutura comum. Estes três tipos de formato de imagens são:

- PBM (Portable BitMap) - imagens a preto e branco (sem tons de cinzento)
- PGM (Portable GrayMap) - imagens em tons de cinzento
- PPM (Portable PixMap) - imagens a cores

A definição original destes formatos teve em vista permitir a transmissão de imagens por meio de correio eletrônico que até então não permitia a transmissão de ficheiros anexados, binários ou não. A definição do formato foi mais tarde modificada para permitir a representação binária dos conteúdos das imagens.

Os formatos de imagem PBM são constituídos pelos seguintes campos:

```
<magic number>
<Altura da imagem ‘‘m’’> <Largura da imagem ‘‘n’’>
<Valor máximo dos tons de cinza>
<valor do pixel (0,0)> ... <valor do pixel (0,n)>
          .          .              .
          .             .           .
          .                .        .
<valor do pixel (m,0)> ... <valor do pixel (m,n)>
```

O Identificador do tipo de formato (designado por “magic number”), é determinado de acordo com Tipo, ASCII ou Binário

Tipo	ASCII	Binário
PBM	P1	P4
PGM	P2	P5
PPM	P3	P6

Em particular, no nosso caso, iremos trabalhar com imagens PGM do tipo ASCII, ou seja, “P2”. Assim, um exemplo de imagem PGM que o programa deve ser capaz de processar é:

```
P2
24 7
15
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  3  3  3  3  0  0  7  7  7  7  0  0 11 11 11 11  0  0 15 15 15 15  0
0  3  0  0  0  0  0  7  0  0  0  0  0 11  0  0  0  0  0 15  0  0 15  0
0  3  3  3  0  0  0  7  7  7  0  0  0 11 11 11  0  0  0 15 15 15 15  0
0  3  0  0  0  0  0  7  0  0  0  0  0 11  0  0  0  0  0 15  0  0  0  0
0  3  0  0  0  0  0  7  7  7  7  0  0 11 11 11 11  0  0 15  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
```

4.2 Run-Length[3]

Run-length (ou RLE) é uma técnica para comprimir cadeias de caracteres onde existem sequências longas de caracteres repetidos. O princípio do funcionamento dessa codificação é simples: Quando temos a ocorrência de uma repetição contínua de determinado caractere, por exemplo, BAAAAAAAAAAAAC, é possível substituir sua representação pelo par (A, 12) precedido por um identificador resultado em uma string do tipo B@A12C.

Na compressão de imagens esta técnica é mais promissora pois imagens apresentam maiores áreas contínuas de uma mesma cor. Desenhos e outras imagens com número limitados de cores tendem a gerar melhores resultados usando esta técnica.

4.3 Proposta

O sistema deve ser capaz de compactar um arquivo `*.pgm` e descompactar um arquivo `*.rl`. A compactação consiste em ler um arquivo de imagem no formato PGM-P2 (como exemplificado acima) e gerar um output no `stdout` com o novo arquivo já compactado. A descompactação deve ser capaz de ler um aquivo RL e gerar um output no `stdout` com o conteúdo do PGM-P2. O output bem como o arquivo `*.rl` seguirá o padrão de cabeçalho do arquivo de imagens PBM, ou seja, contem o *magic number* "`P8`", altura e largura e o valor máximo dos tons de cinza. Seguindo, tem-se a matriz de pixels compactada com o algoritmo de run-length. O exemplo abaixo é o arquivo `feed.rl` resultado da compactação do exemplo dado na sub-seção 4.1:

```
P8
24 7
15
ff 0 24
0 ff 3 4 0 0 ff 7 4 0 0 ff 11 4 0 0 ff 15 4 0
0 3 ff 0 5 7 ff 0 5 11 ff 0 5 15 0 0 15 0
0 3 3 3 0 0 0 7 7 7 0 0 0 11 11 11 0 0 0 ff 15 4 0
0 3 ff 0 5 7 ff 0 5 11 ff 0 5 15 ff 0 4
0 3 ff 0 5 ff 7 4 0 0 ff 11 4 0 0 15 ff 0 4
ff 0 24
```

Observe que a substituição do número pela tríade (demarcador, cor do pixel e frequência) só é vantajoso quando o número de vezes que um mesmo pixel aparece em sequência é maior que 3, ou seja, em um cenário que o número de um mesmo pixel apareça 3 vezes ou menos, a saída deverá ser o próprio número e não a representação reduzida. Por exemplo, em uma sequência

`'1 2 2 3 3 3 4 4 4 4 5 5 5 6 6 6 6'`

o resultado final deverá ser

`'1 2 2 3 3 3 ff 4 4 5 5 5 ff 6 4'`

Atente para o fato que no fim de cada linha não existe um espaço em branco, o caractere de quebra de linha deve preceder imediatamente o ultimo elemento da linha.

5 Parte 2: Huffman[2]

Além da compactação de imagem é necessário a compactação do texto, para isso iremos usar o algoritmo de compactação de Huffman.

5.1 Algoritmo de Huffman

A codificação de Huffman é um método de compressão que usa as probabilidades de ocorrência dos símbolos no conjunto de dados a ser comprimido para determinar códigos de tamanho variável para cada símbolo.

Para atribuir aos caracteres mais frequentes os códigos binários de menor comprimento, constrói-se uma árvore binária baseada nas probabilidades de ocorrência de cada símbolo. Nesta árvore as folhas representam os símbolos presentes nos dados, associados com suas respectivas probabilidades de ocorrência. Os nós intermediários representam a soma das probabilidades de ocorrência de todos os símbolos presentes em suas ramificações e a raiz representa a soma da probabilidade de todos os símbolos no conjunto de dados. O processo se inicia pela junção dos dois símbolos de menor probabilidade, que são então unidos em um nó ao qual é atribuída a soma de suas probabilidades. Este novo nó é então tratado como se fosse uma folha da árvore, isto é, um dos símbolos do alfabeto, e comparado com os demais de acordo com sua probabilidade. O processo se repete até que todos os símbolos estejam unidos sob o nó raiz.

5.2 Proposta

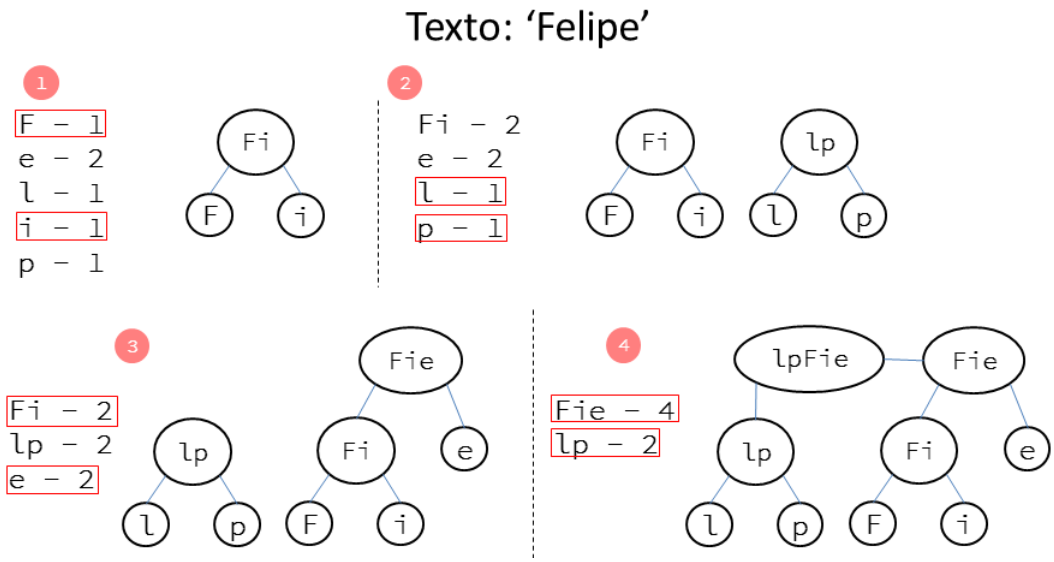
5.2.1 Árvore de frequência

Para determinar o mapa de bit por letra é importante calcular a frequência com que os caracteres aparecem no texto. Em particular, em nosso trabalho, iremos trabalhar com os caracteres de a-z, A-Z, 0-9 e o caractere espaço ‘ ’.

Ao determinar a frequência dos termos, monta-se a árvore de Huffman em que nós a esquerda recebem bit zero e nós a direita bit 1. Para determinar a posição dos nós e a hierarquia entre eles, monta-se a árvore baseado na frequência dos termos em que se escolhe sempre os dois termos com menor frequencia, sendo o menor dentre eles vira o filho da esquerda e o maior o filho da direita.

Um problema factível está quando frequência de 3 ou mais termos coincidem e é necessario uma metrica para determinar qual serão os dois termos que iremos tomar para criar o novo nó.Em particular para o trabalho, deve-se escolher aqueles que apresentam o primeiro caractere o menor código ASCII para criar o novo nó da arvore, ou seja, a esquerda irá o termo com o primeiro caractere contendo o menor código ASCII e a direita o termo com o primeiro caractere contendo o segundo menor código ASCII.

Uma vez que determinou-se quais os dois caracteres vão ser unidos em uma nova árvore, o novo nó tera como frequência a somas dos nós que lhe deram origem e o seu termo é a concatenação dos termos dos nós filhos. Abaixo, é exemplificado o processo para criação da árvore para a palavra “Felipe”:



O comando de ‘`dump tree <filename>.txt`’ deverá ser capaz de imprimir a árvore de frequência in-order. Para que não exista qualquer problema quanto a ordem ou a formatação dos que os termos são apresentados na tela e comparados no SSP, o algoritmo para percorrer a árvore in-order está disponível abaixo:

```
void inorder(Node *tree){
    if(tree != NULL){
        inorder(tree->left);
        printf("%6d - %s\n",tree->frequence, tree->term);
        inorder(tree->right);
    }
}
```

O algoritmo in-order é recursivo e percorre todos os elementos da árvore. No exemplo acima, o resultado do programa para o texto de entrada “Felipe” seria:

```
 1 - l
 2 - lp
 1 - p
 6 - lpFie
 1 - F
 2 - Fi
 1 - i
 4 - Fie
 2 - e
```

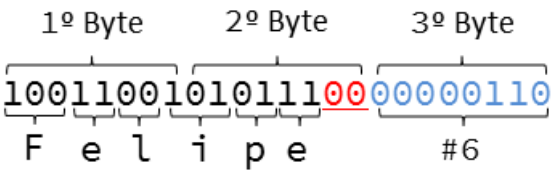
5.2.2 Compactação de Huffman

O sistema aqui implementado deve ser capaz de compactar um arquivo de texto plano (`*.txt`), de no máximo 1000 caracteres válidos, utilizando o algoritmo de Huffman. Assim, uma vez que a árvore de frequência foi gerada, é necessário criar uma tabela que converta cada caractere do texto em um código binário único. Uma importante característica do processo de construção da tabela é que os caracteres com maior frequência recebem o menor número de bits o que garante no final um texto representado com o menor número de bits possível no algoritmo de Huffman.

Essa tabela de símbolo/bits tem a função de ajudar na descompactação da informação recebida o que torna necessário, ao compactar o texto original, que antes tenhamos a tabela de bits de cada termo. Esta tabela deverá ser exibida ordenada pelo ASCII dos elementos e ao fim deve ser colocado um caractere ‘-’ para indicar o fim da tabela e o inicio do texto compactado. Abaixo esta a tabela de símbolo/bits para o texto “Felipe”:

```
F - 100
e - 11
i - 101
l - 00
p - 01
-
```

A compactação é feita substituindo-se os caracteres pelos bits correspondentes de acordo com a tabela já calculada. A cada 8 bits concatenados, salva-se o byte no arquivo de texto compactando assim todo o texto original.



Um problema comum neste tipo de compactação é quando o número total de bits não é múltiplo de 8, ou seja, como somente é possível salva informações no disco com a unidade de bytes, o ultimo byte do texto pode não ser totalmente utilizado. Com o intuito de resolver tal problema, existem varias formas de informar o fim do arquivo ou o número total de bits

utilizado. Em nosso trabalho, utilizaremos o ultimo byte para indicar quantos bits foram usados no ultimo byte do texto compactado.

Na imagem anterior, o texto 'Felipe' utilizou somente 6 bits do último byte, desta forma os últimos dois bits foram completados com zero o ultimo byte traz o número 6 que indica quantos bits foram utilizados no byte final do texto.

Após a compactação, o resultado final foi:

```
F - 100
e - 11
i - 101
l - 00
p - 01
-
???
```

Segue alguns links que podem ajudar na elaboração do algoritmo de Huffman:

- `http://www.programminglogic.com/implementing-huffman-coding-in-c/`
- `http://michael.dipperstein.com/huffman/`

5.2.3 Descompactação de Huffman

O comando '`descompactar huffman <filename>.huff`' deverá fazer o serviço contrario da compactação, ou seja, este comando deverá ser capaz de abrir um arquivo `*.huff` e retornar a mensagem original contida neste documento. Como padrão, todo arquivo `*.huff` tem a mesma estrutura que a saída gerada pelo algoritmo de compactação, abaixo uma explicação desta estrutura:

```
<Primeiro elemento da tabela de símbolo/bit>
                    .
                    .
                    .
<Último elemento da tabela de símbolo/bit>
<delimitador de fim da tabela>
<texto compactado>
```

O resultado da descompactação do arquivo `Felipe.huff` que previamente continha o texto 'Felipe' é, por razões obvias, o texto original 'Felipe'.

6 Observações importantes:

- A implementação é livre, crie quantas TAD's julgarem necessárias. Importante notar que a modularização do código bem como a forma como as TAD's foram criadas serão levados em consideração na atribuição final da nota.
- Coloque dentro do zip todos os arquivos de código (`*.h *.c`), o Makefile e um arquivo texto com o nome e número USP.
- Compile o programa em um sistema operacional linux antes de submeter ao SSP, **trabalho em que o comando `make` e `make run` não funcione, não será corrigido e consequentemente receberá nota zero**. Caso não tenha o sistema operacional linux instalado em sua maquina, aconselha-se a utilização de uma máquina virtual. Como instalar o sistema operacional linux em uma maquina virtual está descrito no seguinte tutorial `http://maistutoriais.com/2012/01/instalar-maquina-virtual-usando-o-virtual-box/`
- Referencie, com um comentário no próprio código, qualquer algoritmo ou trecho de código retirado da internet. Código copiado sem a devida referencia é considerado plágio que por sua vez é crime.
- Nenhuma saída do trabalho será em arquivo, ou seja, o resultado dos algoritmos de compactação e descompactação deverá ser exibidos utilizando o stdout.

Referências

[1] J. M. Brisson Lopes. *Computação Gráfica*.

[2] a enciclopédia livre. Wikipédia. Codificação de huffman, março 2013.

[3] a enciclopédia livre. Wikipédia. Codificação run-length, Março 2013.