Relatório
Compressor de dados

Introdução

Para o desenvolvimento da minha ferramenta de compressão o código de terceiros que implementei foi a função de ordenação bubble sort (my_bubbleSort) e a função que gera os códigos binários (shannon_fano), é claro que foram adaptados às minhas necessidades.
No relatório irei expor a minha estratégia de desenvolvimento, bem como apresentarei uma pequena tabela com resultados de testes que fiz com a minha ferramenta.

À parte deste relatório existe ainda um pequeno manual (manual.pdf) que explica como usar a ferramenta de compressão e respetiva descompressão.

Infelizmente a ferramenta é limitada no tamanho dos ficheiros que esta pode comprimir, o maior ficheiro que comprimiu ronda os 25 MB.
Também a ferramenta de descompressão não acompanha em termos de execução a ferramenta de compressão pelo que é mais lenta e também funciona apenas com um leque pequeno de ficheiros.

Explicar o código…

A estrutura de dados:

```
typedef struct simbolo{
  char byte;
  float  ocorrencias;
  int* codigo;
  int bit;
  float probabilidade;
}Bloco;
```

Esta é a estrutura de dados que uso para armazenar um byte e tudo o que lhe diz respeito, vamos ver com mais detalhe:

Char byte – contém o byte (8 bits);

Float ocorrencias - é o número de vezes esse byte ocorre no ficheiro;
Int* código – servirá para armazenar a codificação que é atribuída a este byte após a execução do algoritmo Shannon-Fano;
Int bit – é o numero de inteiros que são necessários para a respetiva codificação Sannon-Fano;
Float probabilidade – está relacionado diretamente com o campo ocorrências;

Começar do princípio …

A primeira tarefa desta função é muito simples, abrir o ficheiro introduzido pelo utilizador, copiar o seu conteúdo para um buffer, usar a função fseek para apontar para o final do ficheiro para desse modo podermos retirar o valor da variável "unsigned long int tamanho" que contém o valor do tamanho do ficheiro.

```
  printf(" >Nome do ficheiro que deseja comprimir: ");
  n_bits=scanf("%s", nome_ficheiro);
  n_bits=system("clear");
  ficheiro_origem=fopen (nome_ficheiro , "rb");
  if (ficheiro_origem==NULL) {
     fputs (" >Erro Ficheiro\n",stderr);
     return 0;
  }
  fseek(ficheiro_origem, 0, SEEK_END);
  tamanho=ftell(ficheiro_origem);
  rewind(ficheiro_origem);
  buffer_ficheiro=(char*) malloc(tamanho*sizeof(char));
  resultado=fread (buffer_ficheiro, sizeof(char), tamanho,
ficheiro_origem);
  if(resultado!=tamanho){
     printf(" >Erro de Leitura\n");
     return 0;
  }
  fclose (ficheiro_origem);
```

De seguida efetuam-se as seguintes etapas:

I - Estatística do ficheiro;

II – Ordenação dos bytes por ordem decrescente de ocorrências;

III – Calcular probabilidades de cada byte;
IV – Compressão da informação pelo algoritmo Shannon-Fano;
V – Codificar a informação num buffer em que substituimos cada byte pela respetiva codificação obtendo assim (supostamente) um buffer com tamanho inferior ao tamanho do ficheiro original (objetivo da compressão);
VI – Por último escrever no ficheiro de destino a informação que codificamos;

Vamos explorar cada etapa com mais detalhe …

I - Estatística do ficheiro

Simplesmente percorremos o buffer que contém o ficheiro (buffer_ficheiro) e executamos uma das duas seguintes ações:
1 – Caso o byte lido já exista na estrutura de dados , apenas incrementamos ao numero de ocorrências;
2 – Caso o byte não exita alocamos espaço na memória para um novo byte colocando o campo ocorrências a 1;

Esta função no final vai retornar um valor crucial para cálculos posteriores, o número de bytes diferentes lidos do ficheiro, por outras palavras o tamanho do array de estruturas de dados (blocos).

Não há nada mais de especial a salientar relativamente a esta etapa.

II , III
(Não há nada de especial a salientar relativamente a estas etapas)

IV – Compressão da informação pelo algoritmo Shannon-Fano

Esta função vai requerer as probabilidades de cada byte diferente lido do ficheiro, daí a variável n_bytes_diferentes que nos diz quantas linhas terá a tabela Shannon-Fano.
Usamos duas varias pack1 e pack2 para dividir em grupos consoante as probabilidades e atribuímos 0's no grupo de cima e 1's no grupo de baixo, depois fazem-se duas chamadas recursivas da função em que numa vamos codificar os bytes que ficaram em pack1 e noutra os bytes que ficaram em pack2.
Apenas tive que alterar alguns campos para que o algoritmo escreve-se diretamente na minha estrutura de dados.

V – Codificar a informação num buffer

Este passo foi o mais difícil de implementar e o que logicamente sofreu mais alterações.

No código podemos ver comentados os três ciclos “for” (1), (2) e (3).

(1) – Ciclo que acompanha o buffer_ficheiro, só terminamos a função quando todos os bytes forem sujeitos aos ciclos interiores;
(2) Lê para um array byte codificações de tamanaho variado que representam os bytes do ficheiro original;
Quando este byte atinge 8 bits o que fazemos é chamar uma função que transforma esses 8 bits num valor inteiro que posteriormente passará para uma carater e é este carater que vamos despejar no ficheiro comprimido.
Esta é a resposta à pergunta: ***como colocar codificações de tamanho variado em que cada dígito binário é um bit de informação num array de bytes?***
Esta foi a minha estratégia para resolver esse problema.
(3) Basicamente aqui é feito o que foi dito em (2) mas saliento:

```
if(i_byte<8&&(i+1)==n_fich&&k==blocos[j].bit){
  for(;i_byte<8;i_byte++){
     byte[i_byte]=0;
  }
```

Pois representa a situação limite, i.e: Imaginemos que o ficheiro está no final e apenas sobraram 3 bits para codificarmos, ora é impossível colocarmos 3 bits num byte a solução passará então por acrescentar 0’s ao resto do byte depois dos 3 bits isto para na descompressão através do numero de bits que contém este buffer conseguirmos marcar aqueles 5 bits a 0 como “lixo”;

VI – Por último escrever no ficheiro de destino a informação que codificamos

Nesta etapa começamos por calcular o tamanho do ficheiro de destino que será o ficheiro comprimido, como?

```
for(j=0;j<bytes_difs;j++){
   i+=2;//byte da struct mais o '\0' carater
```

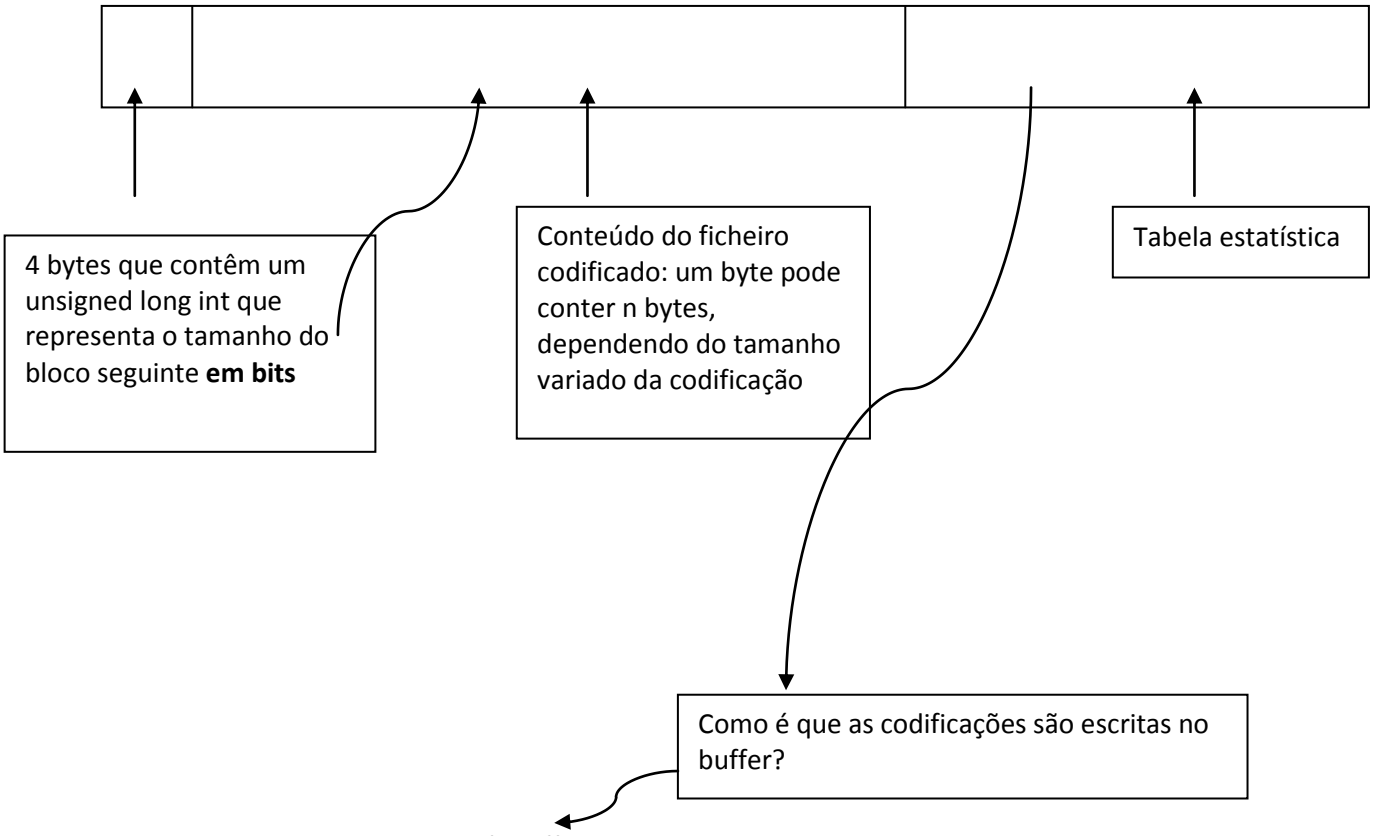
```
    i+=blocos[j].bit+1;//n bits da codificação shannon_fano +1 <=...
  }
  i+=4+(n_bits/8);

  buffer = (char*) malloc(i*sizeof(char));
```

Não nos podemos esquecer a nenhuma altura que a tabela tem de estar no ficheiro comprimido por daí i+=blocos[j].bit+1 estamos a guardar espaço para cada bit da codificação que corresponde a cada byte.
Por fim temos o (+4) bytes que servirão para armazenar o numero de bits que o bloco com o conteúdo codificado do ficheiro vai ter, isto porquê?

Irá revelar-se muito útil a quando da descompressão para sabermos quando começa o ficheiro, quando termina e por sua vez quando começa a tabela estatística.

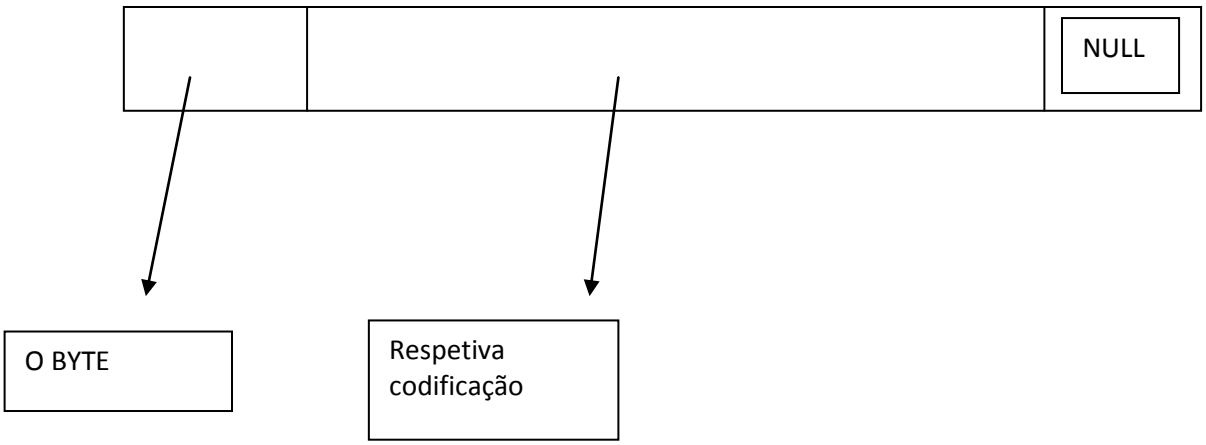
O ficheiro comprimido terá o seguinte aspeto se o disposermos num array:



Vamos ver mais em detalhe:

```
  printf(" >A copiar tabela para buffer...\n");
  //colocar byte seguido de respetiva codificação no buffer (A tabela estatistica para
descompressão)
  for(j=0; j<bytes_difs; j++){
     buffer[ind_buffer]=blocos[j].byte;
     ind_buffer++;
     for(k=0;k<=blocos[j].bit; k++, ind_buffer++){ //<= crucial para apanhar todo o codigo
shannon-fano
        if(blocos[j].codigo[k]==1)
          buffer[ind_buffer]='1';
         else buffer[ind_buffer]='0';
     }
     buffer[ind_buffer]='\0'; ind_buffer++;
  }
```

Pelo código podemos observar que colocamos o byte e seguidamente colocamos a respetiva codificação em que cada dígito binário é um byte. Servimo-nos do valor de blocos.bit para imprimir no buffer a quantidade certa de bits (visto que o malloc que salvaguarda memória para os códigos é por excesso 20, um defeito da minha aplicação).
Como sabemos se terminou ou não uma determinada codificação?
Simples coloca-se um carater NULL dizendo que o seguinte byte é um BYTE e não uma parte da codificação, não existe hipótese de colisão de dados uma vez que os códigos são binários.
A tabela terá esta disposição num array:



Conclusão

É essecialmente isto a minha ferramenta de compressão de dados, a ferramenta de descompressão segue o mesmo raciocínio apesar de não ter sido implementada tão bem quanto esta.

Um grande passo em falta na minha aplicação é a codificação RLE (Run-length encoding), por tal motivo eu tentei uma implementação do meu código com a codificação RLE.
Verifiquei que defacto RLE pode ser uma mais valia uma vez que para bytes que se repetem num array em posições contíguas ele vai substituir todos esses bytes por um único a dificuldade será arranjar um método para ao descomprimir saber que naquele índice aquele byte se repete n vezes. Ainda assim tentei implementar essa versão.

Resultados de alguns testes efetuados…

(versão sem RLE)

Nome do ficheiro	Tamanho do ficheiro original	Tamanho do ficheiro após compressão
myfile.pdf	118 375 bytes	121 393 bytes
isd.pdf	1 727 175 bytes	1 733 953 bytes
surface.pdf	2 608 029 bytes (2,6 MB)	2 558 998 bytes (2,6 MB)
android.pdf	15 555 590 bytes (15,6 MB)	15 429 450 bytes (15,4 MB)
bro.pdf	22 568 101 bytes (22,6 MB)	22 511 285 bytes (22,5 MB)
azeitonas.avi	17 519 698 bytes (17,5 MB)	17 506 730 bytes (17,5 MB)
nancy.mp3	6 480 033 bytes (6,5 MB)	5 855 458 bytes (5,9 MB)
fool.mp3	10 013 314 bytes (10 MB)	9 997 848bytes (10 MB)
queen.mp3	15 949 860 bytes (16 MB)	15 893 922 bytes (15,8 MB)

(versão com RLE)

Nome do ficheiro	Tamanho do ficheiro original	Tamanho do ficheiro após compressão
myfile.pdf	118 375 bytes	121 347 bytes
isd.pdf	1 727 175 bytes	1 727 960 bytes
surface.pdf	2 608 029 bytes (2,6 MB)	2 495 118 bytes (2,5 MB)
android.pdf	15 555 590 bytes (15,6 MB)	15 330 720 bytes (15,3 MB)
bro.pdf	22 568 101 bytes (22,6 MB)	22 467 115 bytes (22,5 MB)
azeitonas.avi	17 519 698 bytes (17,5 MB)	17 493 549 bytes (17,5 MB)
nancy.mp3	6 480 033 bytes (6,5 MB)	5 202 013 bytes (5,2 MB)
fool.mp3	10 013 314 bytes (10 MB)	9 811 551 bytes (9,8 MB)
queen.mp3	15 949 860 bytes (16 MB)	15 538 302 bytes (15,6 MB)

(Ficheiro comprimido maior que original)
(Ficheiro comprimido menor do que o original)
(Ficheiro comprimido praticamente igual ao original)