2020/2021

Descodificação recursiva de um código binário

Algoritmos e Estruturas de Dados (40437)



Trabalho realizado por:

Tiago Alvim 95584 50%

Vasco Costa 97746 50%

# Introdução

O trabalho proposto consiste na descodificação de uma mensagem representada binariamente através de símbolos que a formam e nas várias possibilidades subjacentes, nomeadamente, o número de vezes que a função descodificadora tem de ser chamada, numa implementação recursiva, que facilita a sua solução, fazendo utilização de *depth-first* de forma eficaz e simples. E no estudo de ao final de quantos símbolos será possível verificar-se que o símbolo que se colocou está de facto errado. Isto acontece visto que, existem símbolos que apresentam partes em comum, no entanto, apenas um permite que a mensagem seja corretamente descodificada.

A codificação e a descodificação de mensagens são algo que sempre se apresentaram muito relevantes em diversas matérias, na guerra por exemplo. Sendo que uma das mais famosas máquinas de codificação e descodificação é a Enigma que era utilizada para mensagens de guerra durante a 2ª Guerra Mundial, acredita-se que também graças a ter-se conseguido decifrar as diversas cifras o conflito tenha acabado mais cedo.

Atualmente o ramo da criptografia é da mais alta importância e utilizada diariamente uma vez que permite que a informação pessoal seja guardada de quem a tenta utilizar, desde dados bancários a mensagens pessoais. No entanto, é necessário ter em atenção a sua aplicação. Por vezes faz-se o compromisso entre segurança e facilidade de uso, para o uso quotidiano não seria prático a utilização de uma palavra passe bastante grande com caracteres especiais, inclusive, para desbloquear o telemóvel todas as vezes que se pretendesse utilizar. Tornando-se numa cifra mais complexa e difícil ou até impossível de decifrar sem a chave, obrigando a uma abordagem *brute-force* que obrigaria a vários anos para a descodificar. A obtenção de uma palavra passe de *Wi-Fi* do tipo *WPA2*, dos mais utilizados, como nas casas de cada um, poderá levar desde segundos a milhares de anos consoante o método utilizado. A utilização de dicionários grandes e diversos poderá facilitar tal tarefa como a paralelização do processo em placas gráficas.

# Abordagem computacional

## *recursive\_decoder\_1*

Para resolver este problema a nossa primeira abordagem começou por percorrer todos os símbolos, passando a analisar-se o código binário desse mesmo símbolo. Na ocorrência de algum caracter diferente ao da mensagem codificada resulta nesse símbolo não ser possível pelo que se passa a analisar o símbolo seguinte.

Caso o símbolo seja possível, então comparamos com a mensagem original, se corresponde então é porque se descodificou um símbolo correto incrementando *good\_decoded\_size*. Aumentando o índice da mensagem descodificada, (*decoded\_idx*), levando a chamar a mesma função para analisar o resto da mensagem. E como existe a possibilidade de haver mais do que um símbolo possível, nesse local, tem que se diminuir o *decoded\_idx* evoltar a repor *encoded\_idx* para se poder analisar com as condições iniciais, *backtracking*. No entanto, consideramos que isto não é o método mais desejado dado que, estava-se a comparar com a mensagem original e a adicionar apenas os símbolos corretos. Continuando-se a fazer a recursão tendo apenas atenção aos índices para se poder concretizar o estudo do número de chamadas à função e quantos símbolos teriam que ser percorridos até se chegar à conclusão de que era um símbolo incorreto. Para se fazer isto, ao início da função verifica-se se já nos encontramos com símbolos extra, *good\_decoded\_size* será menor que *decoded\_idx,* incrementando o número de chamadas à função.

## *recursive\_decoder*

Para evitar o ato menos honesto referido anteriormente alterou-se a função para se ir acrescentando sempre o símbolo possível à mensagem descodificada. Alcançando-se o final da mensagem codificada caso seja um símbolo de facto correto deixando-se esse permanecer, o número de soluções terá que ser superior ao que se tinha antes. Na eventualidade de este não ser o correto então coloca-se o símbolo anterior.

Para uma estimativa de complexidade computacional, afirmamos que terá um polinómio quadrático para representar o seu crescimento. Isto por se ter um ciclo para símbolos todas as vezes, com uma passagem por *bits* de cada símbolo sendo que se avança *bits* também na mensagem codificada, acontecendo um certo número de vezes até se chegar ao final, tendo apenas uma constante associada.

## *bit\_by\_bit*

Com o intuito de se realizar a contagem do maior número de casos a ter em consideração num dado instante esta função nasceu. Teve-se, no entanto, uma abordagem diferente à sugerida pelos docentes uma vez que já nos encontrávamos focados na ideia da descodificação *bit* a *bit*. Daí se fazer uso de uma pilha em vez de se fazerem alterações à função anterior, *recursive\_decoder.*

Na idealização de um uso prático para esta função tivemos em mente não se saber se nos encontrávamos no início da mensagem, o que alterou bastante o raciocínio. Referimos prático no sentido de apenas se ter começado a fazer a leitura após o início ou caso tenha acontecido algum erro de transmissão da mensagem.

Deste modo a cada *bit* verifica-se se o início de cada símbolo é possível e caso possa ser então adiciona-se a uma pilha. Se houver algo na pilha, então, para o *bit* anterior um dado símbolo era possível pelo que tem que se verificar se o mesmo agora ainda o é. Em cada um destes casos verificamos que há mais uma possibilidade, e faz-se comparação com o valor obtido anteriormente. Para se poder fazer um estudo detalhado de como esta variação ocorre tiveram que se fazer algumas alterações no restante código. Por isso se inclui todo o código na secção código.

Não sabendo se os resultados aqui obtidos estão corretos para o ponto seguinte não se os decidiu utilizar. Daí a utilização de números grandes para garantirmos que não resultaria daí algum problema, à custa de maior uso de memória.

## *bit\_decoder\_2* e *bit\_decoder\_3*

O objetivo desta função era realizar a descodificação *bit* a *bit*, realizando a descoberta de forma *breadth-first.* Para o efeito realiza-se uma passagem inicial por todos os símbolos e se o *bit* inicial do símbolo corresponder ao primeiro da mensagem então coloca-se um objeto *auxiliary* numa pilha. O objeto *auxiliary* é caracterizado por ter a indicação do símbolo que representa, da posição em que nos encontramos a ler o *bit* do símbolo e um ponteiro para o símbolo possível anterior que originou este.

Seguidamente tiram-se todos os símbolos da pilha e verifica-se se o bit seguinte ainda corresponde ao da mensagem. E caso corresponda então guarda-se numa estrutura secundária para se poder esvaziar a primeira. E assim sucessivamente. Chegando ao ponto em que é o final de um símbolo concluimos que esse é possível na mensagem pelo que temos que procurar quais serão os símbolos que poderão ser o começo. Ao chegar ao fim da mensagem codificada, através do ponteiro na estrutura *auxiliary* consegue-se descodificar a mensagem do fim para o princípio.

No entanto, não conseguimos obter resultados corretos. Por vezes a mensagem descodificada apresenta sempre o mesmo símbolo. Isto leva-nos a crer que não conseguimos implementar corretamente o uso de ponteiros e endereços para se conseguir fazer a atribuição dos símbolos corretos à mensagem descodificada.

Também se tentou fazer a implementação deste método com uma fila, mas os resultados obtidos foram idênticos. Esta tentativa foi pensada por usualmente se utilizarem filas em procuras *breadth-first* mas nem sempre é necessário fazer recurso a tal, como verificámos. Também seria irrelevante a estrutura de armazenamento utilizada uma vez que se retiravam todos os elementos, mas teve-se mais interesse em fazer também a implementação de estruturas de armazenamento de forma a se aprender mais.

# Análise[[1]](#footnote-2)

Os dados seguintes foram obtidos utilizando o comando *-x* para diversos números de símbolos, com a utilização de 100 testes para cada, com 80 medições válidas e 20 descartadas.

## Número de chamadas à função

Um aspeto que pode ser relevante na análise destas situações será possivelmente o número de vezes que a função é chamada recursivamente e como é possível visualizar, o resultado obtido é de facto muito curioso, verificar que tem o aspeto de uma função logarítmica e de facto, fazendo a aproximação a um polinómio do tipo obteve-se um de 0,9999 com o polinómio e não assumindo que depende de um valor para obteve-se com um valor de de 0.9992. Assim, pode-se assumir que para valores superiores o mesmo se verificará. Isto poderá tornar-se significativo para a descodificação para um número de símbolos superior uma vez que um crescimento de ordem logarítmico é comparativamente lento com um crescimento de ordem , por exemplo. Como se verifica de forma clara, o nível de recursões necessárias para a descodificação cresce de forma logarítmica com o número de símbolos levando a que uma duplicação de não necessite do dobro de memória para o stack da recursão, algo que se torna bastante necessário a ter em consideração.

Gráfico 1

## Número máximo de símbolos

Para esta situação a obtenção de um polinómio que consiga representar o crescimento deste número de casos é de facto mais complexo e poderá nem fazer grande sentido. Uma vez que se verificam a existência de picos e para o número máximo de símbolos analisados até se chegar à conclusão de que a mensagem estava errada é bastante variável, de 16 a 1238, para pouca variação no número de símbolos. No entanto, os picos referidos ocorrem aproximadamente, tanto para o número máximo como para o número mínimo e para a mediana num mesmo número de . Sem consistência no crescimento de .

Gráfico 2

Através da visualização do gráfico é possível verificar-se que *look\_max* começa por ser bastante disperso passando a agrupar-se para um número superior. Assim como *look\_min* e o conjunto destes passa a ter uma dispersão menor, aproximando-se dos valores da mediana.

Na análise destes resultados para um número de símbolos entre 0 e 100, aproximadamente, é possível verificar-se que esta se poderá assemelhar a uma função logarítmica. No entanto, na tentativa da realização de uma aproximação todas as tentativas apresentaram um valor para abaixo de 0,90 pelo que se acabou por excluir a hipótese de este elemento ter um crescimento polinomial, assim como os outros.

Gráfico 3

## *bit\_by\_bit*

Fazendo a alteração na parte do código na parte de análise de forma a que este faça a mesma análise para esta função obtêm-se os resultados que se conseguem visualizar no gráfico seguinte.

Através de uma análise ao seguinte gráfico consegue-se verificar que existe um crescimento linear dos resultados obtidos com o crescimento do número de símbolos. De forma explicita verificamos que segue uma reta de declive 1 e ordenada na origem de menos 1. , com de 1. Assim sendo, podemos considerar um aumento necessário do tamanho do *stack* de forma linear.

Gráfico 4

Na tentativa do ponto seguinte optou-se por usar um número bastante grande de forma a não se encontrarem erros devido a um tamanho da estrutura de armazenamento pequeno.

Posteriormente, testou-se se de facto era necessário o uso de um valor tão grande e verificámos que a aproximação linear seria provavelmente a melhor opção. Uma vez que a utilização de um número inferior ao aqui observado indicava que era necessário adicionar mais um elemento à estrutura sendo tal impossível, valores muito superiores apenas fariam uso de mais memória consumindo recursos desnecessariamente. Posto isto, a aposta de um crescimento linear é o desejado sugerindo-se a alteração da ordenada na origem para um valor arbitrário positivo.

# Conclusão

Através da realização deste projeto conseguimos verificar que a codificação e descodificação de mensagens são bastante pertinentes nos dias de hoje. Apresentado pormenores que à partida poderiam não ser considerados, como a profundidade necessária a alcançar para a obtenção de uma solução correta, demonstrando também a desvantagem natural de uma abordagem *depth-first*.

Com uma solução *breadth-first* a funcionar corretamente seria interessante analisar e comparar a execução com o método recursivo implementado, tanto em termos de desempenho como de uso de recursos.

# Referências

Material de aula disponibilizado no eLearning

# Código

//

// AED, 2020/2021

//

// Decoding a non-instantaneous binary code

//

#include <stdio.h>

#include <stdlib.h>

//

// Compile time parameters

//

#ifndef MAX\_N\_SYMBOLS

# define MAX\_N\_SYMBOLS 1000 // maximum number of alphabet symbols in a code

#endif

#ifndef MAX\_CODEWORD\_SIZE

# define MAX\_CODEWORD\_SIZE 23 // maximum number of bits of a codeword

#endif

#ifndef MAX\_MESSAGE\_SIZE

# define MAX\_MESSAGE\_SIZE 100000 // maximum number of symbols in a message

#endif

#ifndef N\_OUTLIERS

# define N\_OUTLIERS 20 // discard this number of mesurements (outliers) on each side of the median

#endif

#ifndef N\_VALID

# define N\_VALID 80 // use this number of measurements on each side of the median

#endif

#define N\_MEASUREMENTS (2 \* N\_OUTLIERS + 2 \* N\_VALID + 1) // total number of measurements

//

// Random number generator interface

//

// In order to ensure reproducible results on Windows and GNU/Linux, we use a good random number generator, available at

// https://www-cs-faculty.stanford.edu/~knuth/programs/rng.c

// This file has to be used without any modifications, so we take care of the main function that is there by applying

// some C preprocessor tricks

//

// DO NOT CHANGE THIS CODE

//

#define main rng\_main // main gets replaced by rng\_main

#ifdef \_\_GNUC\_\_

int rng\_main() \_\_attribute\_\_((\_\_unused\_\_)); // gcc will not complain if rnd\_main() is not used

#endif

#include "rng.c"

#undef main // main becomes main again

#define srandom(seed) ran\_start((long)seed) // start the pseudo-random number generator

#define random() ran\_arr\_next() // get the next pseudo-random number (0 to 2^30-1)

//

// Generation of a random non-instantaneous uniquely decodable code with n symbols (inverted Hufffman code)

//

// DO NOT CHANGE THIS CODE

//

typedef struct

{

int scaled\_prob; // proportional to the probability of occurrence of this symbol

int cum\_scaled\_prob; // proportional to the probability of occurrence of this or of all previous symbols

int parent; // -1 means no parent, >= 0 gives the index of the parent

int bit; // -1 means no information, 0 or 1 means append this bit to the parent's code

char codeword[MAX\_CODEWORD\_SIZE + 1]; // the complete (inverted) Huffman code

}

symbol\_t;

typedef struct

{

int n\_symbols; // the number of symbols

int max\_bits; // maximum number of bits of a codeword

symbol\_t \*data; // the symbols and their codes (with extra data at the end --- used to construct the entire Huffman tree)

}

code\_t;

void free\_code(code\_t \*c)

{

if(c != NULL)

{

if(c->data != NULL)

free(c->data);

c->data = NULL;

free(c);

}

}

code\_t \*new\_code(int n\_symbols)

{

int i,i0,i1,n;

code\_t \*c;

//

// Refuse to handle too few or too many symbols

//

if(n\_symbols < 2 || n\_symbols > MAX\_N\_SYMBOLS)

{

fprintf(stderr,"new\_code: n\_symbols (%d) is either too small or too large\n",n\_symbols);

exit(1);

}

//

// Allocate memory for the n\_symbols symbols plus n\_symbols-1 tree nodes for the Huffman tree

//

c = (code\_t \*)malloc(sizeof(code\_t));

if(c == NULL)

{

fprintf(stderr,"new\_code: out of memory\n");

exit(1);

}

c->data = (symbol\_t \*)malloc((size\_t)(2 \* n\_symbols - 1) \* sizeof(symbol\_t));

if(c->data == NULL)

{

free(c);

fprintf(stderr,"new\_code: out of memory\n");

exit(1);

}

//

// Initialize the symbols --- at the beginning, the symbols (leaves of the Huffman tree) are disconnected

//

c->n\_symbols = n\_symbols;

for(i = 0;i < n\_symbols;i++)

{

c->data[i].scaled\_prob = 10 + (int)random() % 991; // a pseudo-random integer belonging to the interval [10,1000]

c->data[i].cum\_scaled\_prob = c->data[i].scaled\_prob; // used only to generate

if(i > 0) // symbols with the

c->data[i].cum\_scaled\_prob += c->data[i - 1].cum\_scaled\_prob; // correct probability

c->data[i].parent = -1; // currently, no parent node

c->data[i].bit = -1; // currently, no bit numbier

c->data[i].codeword[0] = '\0'; // currently, no codeword string

}

//

// Construct the Huffman code

//

// We are going to do it in a O(n^2) way --- speed is not important here

// Using min-heaps would reduce that to O(n log n), but the code would be longer and more difficult to understand

//

n = n\_symbols;

for(;;)

{

//

// Find the two "open" nodes (those with a parent equal to -1) with the smallest scaled\_prob

//

i0 = i1 = -1;

for(i = 0;i < n;i++)

if(c->data[i].parent == -1)

{ // ok, we have an open node

if(i0 < 0 || c->data[i].scaled\_prob < c->data[i0].scaled\_prob)

{ // the smallest scaled\_prob so far

i1 = i0;

i0 = i;

}

else if(i1 < 0 || c->data[i].scaled\_prob < c->data[i1].scaled\_prob)

{ // the second smallest scaled\_prob so far

i1 = i;

}

}

//

// Are we done? Yes when we cannot find two open nodes (this will happen when n == 2 \* n\_symbols - 1)

//

if(i1 < 0)

break;

//

// Merge the two open nodes (close them and create a new open node)

//

c->data[n].scaled\_prob = c->data[i0].scaled\_prob + c->data[i1].scaled\_prob;

c->data[n].cum\_scaled\_prob = -1; // not used but we initialize it anyway

c->data[n].parent = -1;

c->data[n].bit = -1;

c->data[n].codeword[0] = '\0'; // not used but we initialize it anyway

c->data[i0].parent = n; // the parent of node i0 becomes node n --- the left descendant of node n is node i0 (we do not record this information)

c->data[i0].bit = 0; // give this branch a bit of 0

c->data[i1].parent = n; // the parent of node i1 becomes node n --- the right descendant of node n is node i1 (we do not record this information)

c->data[i1].bit = 1; // give this branch a bit of 1

n++;

}

if(n != 2 \* n\_symbols - 1)

{

fprintf(stderr,"new\_code: unexpected value of n [expected %d, we got %d]\n",2 \* n\_symbols - 1,n);

exit(1);

}

//

// For each symbol, initialize its (inverted) Huffman code

//

c->max\_bits = 0;

for(n = 0;n < n\_symbols;n++)

{

i = 0; // the current code size

i0 = n; // the initial tree node index

while(c->data[i0].parent >= 0)

{

if(i >= MAX\_CODEWORD\_SIZE)

{

fprintf(stderr,"ne\_code: MAX\_CODEWORD\_SIZE is too small\n");

exit(1);

}

c->data[n].codeword[i] = '0' + c->data[i0].bit;

i++;

i0 = c->data[i0].parent;

}

c->data[n].codeword[i] = '\0'; // terminate the codeword string

if(i > c->max\_bits)

c->max\_bits = i;

}

//

// Done!

//

return c;

}

//

// Random code symbol

//

// DO NOT CHANGE THIS CODE

//

int random\_symbol(code\_t \*c)

{

int i,r;

//

// Generate an (approximately) uniformly distributed integer in the appropriate range

//

r = random() % c->data[c->n\_symbols - 1].cum\_scaled\_prob;

//

// Find the index i for which c->data[i - 1].cum\_scaled\_prob <= r < c->data[i].cum\_scaled\_prob (with c->data[-1].cum\_scaled\_prob implicitly 0)

// We are going to do it in a O(n) way --- speed is not important here

// We could have used a special version of binary search here, but the code would be longer and more difficult to understand

//

for(i = 0;i < c->n\_symbols;i++)

if(r < c->data[i].cum\_scaled\_prob)

break;

if(i == c->n\_symbols)

{

fprintf(stderr,"ramdom\_symbol: i is too large! Impossible!!! [r=%d]\n",r);

exit(1);

}

return i;

}

//

// Random message

//

// DO NOT CHANGE THIS CODE

//

void random\_message(code\_t \*c,int message\_size,int message[message\_size])

{

int i;

if(message\_size < 1 || message\_size > MAX\_MESSAGE\_SIZE)

{

fprintf(stderr,"random\_message: bad message size (%d)\n",message\_size);

exit(1);

}

for(i = 0;i < message\_size;i++)

message[i] = random\_symbol(c);

}

//

// Encode a message

//

// DO NOT CHANGE THIS CODE

//

void encode\_message(code\_t \*c,int message\_size,int message[message\_size],int max\_encoded\_message\_size,char encoded\_message[max\_encoded\_message\_size + 1])

{

int i,j,n;

char \*s;

if(message\_size < 1 || message\_size > MAX\_MESSAGE\_SIZE)

{

fprintf(stderr,"encode\_message: bad message size (%d)\n",message\_size);

exit(1);

}

n = 0; // encoded message size

for(i = 0;i < message\_size;i++)

{

if(message[i] < 0 || message[i] >= c->n\_symbols)

{

fprintf(stderr,"encoded\_message: unexpected symbol (%d)\n",message[i]);

exit(1);

}

s = c->data[message[i]].codeword;

for(j = 0;s[j] != 0;j++)

{

if(n > max\_encoded\_message\_size)

{

fprintf(stderr,"encode\_message: the encoded message is too big\n");

exit(1);

}

encoded\_message[n++] = s[j]; // concatenate the code word

}

}

encoded\_message[n] = '\0'; // terminate the string

}

//

// Global data used for decoding (to avoid passing all this information in function arguments, thus making the program more efficient)

//

struct

{

code\_t \* c; // the code being used

int \* original\_message; // the original message

int original\_message\_size; // the original message length

int max\_encoded\_message\_size; // the largest possible encoded message size

char \* encoded\_message; // the encoded message

int max\_decoded\_message\_size; // the largest possible decoded message length

int \* decoded\_message; // the decoded message (should be equal to the original message)

long number\_of\_calls; // the number of recursive function calls

long number\_of\_solutions; // the number of solutions (at the end, is all is well, must be equal to 1)

int max\_extra\_symbols; // the largest difference between the partially decoded message and the good part of the partially decoded message)

}

decoder\_global\_data;

#define \_c\_ decoder\_global\_data.c

#define \_original\_message\_ decoder\_global\_data.original\_message

#define \_original\_message\_size\_ decoder\_global\_data.original\_message\_size

#define \_max\_encoded\_message\_size\_ decoder\_global\_data.max\_encoded\_message\_size

#define \_encoded\_message\_ decoder\_global\_data.encoded\_message

#define \_max\_decoded\_message\_size\_ decoder\_global\_data.max\_decoded\_message\_size

#define \_decoded\_message\_ decoder\_global\_data.decoded\_message

#define \_number\_of\_calls\_ decoder\_global\_data.number\_of\_calls

#define \_number\_of\_solutions\_ decoder\_global\_data.number\_of\_solutions

#define \_max\_extra\_symbols\_ decoder\_global\_data.max\_extra\_symbols

//

// Recursive decoder

//

// encoded\_idx ......... index into the \_encoded\_message\_ array of the next bit to be considered

// decoded\_idx ......... index into the \_decoded\_message\_ array where the next decoded symbol will be placed

// good\_decoded\_size ... number of correct decoded symbols

//

// Decoding large messages require a large amount of stack space (one recursion level per message symbol)

// If you get a segmentation fault in our program you may need to increase the stack size (under GNU/linux, you can do it using the command "ulimit -s 16384")

// primeira tentativa

static void recursive\_decoder\_1(int encoded\_idx,int decoded\_idx,int good\_decoded\_size)

{

int b = (decoded\_idx - good\_decoded\_size);

\_max\_extra\_symbols\_ = ( b > \_max\_extra\_symbols\_)? b : \_max\_extra\_symbols\_;

\_number\_of\_calls\_++;

int possible;

int encoded\_idx\_old = encoded\_idx;

for(int i = 0; i < \_c\_->n\_symbols ; i++)//percorrer os simbolos

{

possible = 0;//consideramos o simbolo possivel

for(int p = 0; (\_c\_->data[i].codeword[p] != '\0'); p++ )//percorrer o codigo desse simbolo

{

if(\_encoded\_message\_[encoded\_idx++] != \_c\_->data[i].codeword[p])

{

possible = -1;//o codigo do simbolo e diferente logo nao e possivel e passa-se ao proximo simbolo

break;

}

}

if(possible == 0)//se o simbolo e possivel

{

if( i == \_original\_message\_[decoded\_idx])//se o simbolo for igual ao da mensagem

{

\_decoded\_message\_[decoded\_idx] = i;//guarda-se na descodificada

good\_decoded\_size++;//o good size aumenta

if(good\_decoded\_size == \_original\_message\_size\_)//se tivermos descodificado tudo

{

\_number\_of\_solutions\_++;//aumenta-se o numero de solucoes

}

}

decoded\_idx++;//descodificou-se mais um simbolo

recursive\_decoder\_1(encoded\_idx,decoded\_idx,good\_decoded\_size);

decoded\_idx--;//para simbolo seguinte tem que se voltar a repor

}

encoded\_idx = encoded\_idx\_old;//volta-se a meter o encoded no inicio para o proximo simbolo

}

}

static void recursive\_decoder(int encoded\_idx,int decoded\_idx,int good\_decoded\_size)

{

int b = (decoded\_idx - good\_decoded\_size);

\_max\_extra\_symbols\_ = ( b > \_max\_extra\_symbols\_)? b : \_max\_extra\_symbols\_;

\_number\_of\_calls\_++;

int possible;

int encoded\_idx\_old = encoded\_idx;

for(int i = 0; i < \_c\_->n\_symbols ; i++)//percorrer os simbolos

{

possible = 0;

for(int p = 0; (\_c\_->data[i].codeword[p] != '\0'); p++ )//percorrer o codigo desse simbolo

{

if(\_encoded\_message\_[encoded\_idx++] != \_c\_->data[i].codeword[p])

{

possible = -1;//o codigo do simbolo e diferente logo nao e possivel e passa-se ao proximo simbolo

break;

}

}

if(possible == 0)//se o simbolo e possivel

{

if (good\_decoded\_size == decoded\_idx && \_original\_message\_[decoded\_idx] == i)

{

good\_decoded\_size++;

}

int prev = \_number\_of\_solutions\_;

int prev\_rec = \_decoded\_message\_[decoded\_idx];

decoded\_idx++;//descodificou-se mais um simbolo

recursive\_decoder(encoded\_idx,decoded\_idx,good\_decoded\_size);

decoded\_idx--;//para simbolo seguinte tem que se voltar a repor

if(\_number\_of\_solutions\_ > prev)

prev\_rec = i;

\_decoded\_message\_[decoded\_idx] = prev\_rec;

}

encoded\_idx = encoded\_idx\_old;//volta-se a meter o encoded no inicio para o proximo simbolo

}

if (\_encoded\_message\_[encoded\_idx] == '\0')

{

\_number\_of\_solutions\_++;

//faz-se return aqui se quisermos so ver qual a mensagem descodificada corretamente, não e objetivo

}

}

typedef struct auxiliary

{ int symbol;//o simbolo que é

int position;//a posicao em que esse simbolo esta

struct auxiliary \*prev\_symbol;

}auxiliary;

#define MAX\_QS\_SIZE 1000

typedef struct

{ int top;

auxiliary items[MAX\_QS\_SIZE];

}stack\_n;

//usar ponteiros para se ter apenas um stack e nao se fazerem alteracoes so la dentro

//ser global facilita

int stack\_empty(stack\_n \*s){return ( (s->top == -1 )? 1:0 );}

int stack\_is\_full(stack\_n \*s){return ( (s->top == MAX\_QS\_SIZE-1 )? ( fprintf(stderr,"small stack")) : 0);}

auxiliary stack\_peek(stack\_n \*s) { return s->items[s->top];}

auxiliary stack\_pop(stack\_n \*s)

{

if(!stack\_empty(s))

{

s->top -= 1;

return s->items[s->top+1];

}

else

fprintf(stderr,"pop\_empty");

}

void stack\_put(stack\_n \*s,auxiliary a)

{

if(!stack\_is\_full(s))

{

s->top += 1;

s->items[(s->top)] = a;

}

}

//stack teste no link

//https://onlinegdb.com/3JXMVaeeZ

typedef struct

{ auxiliary items[MAX\_QS\_SIZE];

int front;

int rear;

int count;

}queue;

auxiliary q\_peek(queue \*q)

{ return q->items[q->front];}

int q\_empty(queue \*q)

{ return ((q->count == 0)?1:0);}

int q\_full(queue \*q)

{ return ((q->count == MAX\_QS\_SIZE)?1:0);}

void q\_put(queue \*q,auxiliary a)

{ if(!q\_full(q))

{

if(q->rear == MAX\_QS\_SIZE-1)

q->rear = -1;

q->items[++(q->rear)] = a;

q->count+=1;

}

else

fprintf(stderr,"small max size");

}

auxiliary q\_get(queue \*q)

{

if(!q\_empty(q))

{

auxiliary a = q->items[(q->front)++];

if(q->front == MAX\_QS\_SIZE)

q->front = 0;

q->count-=1;

return a;

}

else

fprintf(stderr,"empty queue");

}

// queue tester

// https://onlinegdb.com/Hy3KDRR1\_

int MAX\_CONSIDER\_ONCE;

static void bit\_by\_bit(int encoded\_idx)

{

MAX\_CONSIDER\_ONCE = 0;

stack\_n arr;

arr.top = -1;

int n\_max;

for(; \_encoded\_message\_[encoded\_idx] != '\0';encoded\_idx++)

{

stack\_n aux;

aux.top = -1;

n\_max = 0;

while( !stack\_empty(&arr) )//tirar tudo do stack e ver se cada simbolo ainda e possivel

{

auxiliary s = stack\_pop(&arr);

if(\_c\_->data[s.symbol].codeword[s.position] != '\0')

{

if(\_encoded\_message\_[encoded\_idx] == \_c\_->data[s.symbol].codeword[s.position])

{

s.position += 1;

stack\_put(&aux,s);//vai para um auxiliar para se esvaziar o original

n\_max++;//e uma possiblidade nesta bit da codificada

}

}//se for o fim entao chegou a ser um simbolo possivel

}

arr = aux;

for(int i = 0; i < \_c\_->n\_symbols ; i++)//se for começo

{

if(\_encoded\_message\_[encoded\_idx] == \_c\_->data[i].codeword[0])

{

auxiliary new;

new.symbol = i;

new.position = 1;//mete se logo 1 que para se ler logo nessa posicao

stack\_put(&arr,new);

n\_max++;

}

}

MAX\_CONSIDER\_ONCE = (MAX\_CONSIDER\_ONCE < n\_max)? n\_max:MAX\_CONSIDER\_ONCE;

}

}

static void bit\_decoder\_2(int encoded\_idx, int decoded\_idx)

{

queue arr;

arr.front = 0;

arr.rear = -1;

for(int i = 0; i < \_c\_->n\_symbols ; i++)

{

if(\_encoded\_message\_[encoded\_idx] == \_c\_->data[i].codeword[0])

{

auxiliary new;

new.symbol = i;

new.position = 1;//mete se logo 1 que para se ler logo nessa posicao

q\_put(&arr,new);

}

}

queue aux;auxiliary a;

while(1)

{

aux.front = 0;

aux.rear = -1;

encoded\_idx++;

while(!q\_empty(&arr))

{

a = q\_get(&arr);

if(\_encoded\_message\_[encoded\_idx] == \_c\_->data[a.symbol].codeword[a.position])//se o bit coincide

{

if(\_encoded\_message\_[encoded\_idx] == '\0')//se for o fim da mensagem

{

for(decoded\_idx = (\_original\_message\_size\_ - 1); decoded\_idx >= 0;decoded\_idx--)

{

printf("%d\n",a.symbol);

\_decoded\_message\_[decoded\_idx] = a.symbol;

a = \*a.prev\_symbol;

}

return;

}

a.position += 1;

q\_put(&aux,a);

}

else if(\_c\_->data[a.symbol].codeword[a.position] == '\0')//bit nao coincide;mas e um simbolo possivel do meio da mensagem

{ //ver o proximo simbolo possivel

for(int i = 0; i < \_c\_->n\_symbols ; i++)

{

if(\_encoded\_message\_[encoded\_idx] == \_c\_->data[i].codeword[0])

{

auxiliary new;

new.symbol = i;

new.position = 1;

new.prev\_symbol = &a;

q\_put(&aux,new);

}

}

}

}

arr = aux;

}

}

static void bit\_decoder\_3(int encoded\_idx, int decoded\_idx)

{

stack\_n arr;

arr.top = -1;

for(int i = 0; i < \_c\_->n\_symbols ; i++)

{

if(\_encoded\_message\_[encoded\_idx] == \_c\_->data[i].codeword[0])

{

auxiliary new;

new.symbol = i;

new.position = 1;//mete se logo 1 que para se ler logo nessa posicao

stack\_put(&arr,new);

}

}

auxiliary a;

while(1)

{

stack\_n aux;

aux.top = -1;

encoded\_idx++;

while(!stack\_empty(&arr))

{

a = stack\_pop(&arr);

if(\_encoded\_message\_[encoded\_idx] == \_c\_->data[a.symbol].codeword[a.position])//se o bit coincide

{

if(\_encoded\_message\_[encoded\_idx] == '\0')//se for o fim da mensagem

{

for(decoded\_idx = (\_original\_message\_size\_ - 1); decoded\_idx >= 0;decoded\_idx--)

{

printf("%d\n",a.symbol);

\_decoded\_message\_[decoded\_idx] = a.symbol;

a = \*a.prev\_symbol;

}

return;

}

a.position += 1;

stack\_put(&aux,a);

}

else if(\_c\_->data[a.symbol].codeword[a.position] == '\0')//bit nao coincide;mas e um simbolo possivel do meio da mensagem

{ //ver o proximo simbolo possivel

for(int i = 0; i < \_c\_->n\_symbols ; i++)

{

if(\_encoded\_message\_[encoded\_idx] == \_c\_->data[i].codeword[0])

{

auxiliary new;

new.symbol = i;

new.position = 1;

new.prev\_symbol = &a;

stack\_put(&aux,new);

}

}

}

}

arr = aux;

}

}

//

// Encode and decode driver

//

// DO NOT CHANGE THIS CODE

//

void try\_it(code\_t \*c,int message\_size,int show\_results)

{

if(message\_size < 1 || message\_size > MAX\_MESSAGE\_SIZE)

{

fprintf(stderr,"try\_it: bad message size (%d)\n",message\_size);

exit(1);

}

\_c\_ = c;

\_original\_message\_size\_ = message\_size;

\_max\_encoded\_message\_size\_ = message\_size \* c->max\_bits;

\_max\_decoded\_message\_size\_ = message\_size + 2000;

\_original\_message\_ = (int \*)malloc((size\_t)\_original\_message\_size\_ \* sizeof(int));

\_encoded\_message\_ = (char \*)malloc((size\_t)(\_max\_encoded\_message\_size\_ + 1) \* sizeof(char));

\_decoded\_message\_ = (int \*)malloc((size\_t)\_max\_decoded\_message\_size\_ \* sizeof(int));

\_number\_of\_calls\_ = 0L;

\_number\_of\_solutions\_ = 0L;

\_max\_extra\_symbols\_ = -1;

if(\_original\_message\_ == NULL || \_encoded\_message\_ == NULL || \_decoded\_message\_ == NULL)

{

fprintf(stderr,"try it: out of memory!\n");

exit(1);

}

random\_message(\_c\_,\_original\_message\_size\_,\_original\_message\_);

encode\_message(\_c\_,\_original\_message\_size\_,\_original\_message\_,\_max\_encoded\_message\_size\_,\_encoded\_message\_);

#if 0

recursive\_decoder(0,0,0);

bit\_by\_bit(0);

printf("Numero de consideracoes %d\n",MAX\_CONSIDER\_ONCE);

bit\_decoder\_2(0,0);

#else

printf("original: ");

for(int i=0;i<\_original\_message\_size\_;i++)

{

printf("%d",\_original\_message\_[i]);

}

printf("\n");

printf("messagem codificada %s \n",\_encoded\_message\_);

recursive\_decoder(0,0,0);

bit\_by\_bit(0);

printf("Numero de consideracoes %d\n",MAX\_CONSIDER\_ONCE);

bit\_decoder\_3(0,0);

printf("Decoded: ");

for(int i = 0;i<\_original\_message\_size\_ ;i++)

{

printf("%d",\_decoded\_message\_[i]);

if(\_decoded\_message\_[i] !=\_original\_message\_[i])

printf("\nERRO\n");

}

printf("\n");

#endif

//

#if 0

if(\_number\_of\_solutions\_ != 1L)

{

fprintf(stderr,"number of solutions: %ld\n",\_number\_of\_solutions\_);

fprintf(stderr,"number of function calls: %ld (%.3f per message symbol)\n",\_number\_of\_calls\_,(double)\_number\_of\_calls\_ / (double)\_original\_message\_size\_);

fprintf(stderr,"number of extra symbols: %d\n",\_max\_extra\_symbols\_);

}

#endif

if(show\_results != 0)

{

//

// print some data about this particular case (average number of calls per symbol, worst probe lookahead)

//

printf("%4d %9.3f %3d\n",\_c\_->n\_symbols,(double)\_number\_of\_calls\_ / (double)\_original\_message\_size\_,\_max\_extra\_symbols\_);

fflush(stdout);

}

free(\_original\_message\_); \_original\_message\_ = NULL;

free(\_encoded\_message\_); \_encoded\_message\_ = NULL;

free(\_decoded\_message\_); \_decoded\_message\_ = NULL;

}

//

// Main program

//

// DO NOT CHANGE THIS CODE

//

int main(int argc,char \*\*argv)

{

//

// Show code words (called with arguments -s n\_symbols seed)

//

if(argc == 4 && argv[1][0] == '-' && argv[1][1] == 's')

{

int seed,n\_symbols,i;

code\_t \*c;

n\_symbols = atoi(argv[2]);

seed = atoi(argv[3]);

srandom(seed);

c = new\_code(n\_symbols);

printf("seed: %d\n",seed);

printf("number of symbols: %d\n",c->n\_symbols);

printf("maximum bits of a code word: %d\n\n",c->max\_bits);

printf("symb freq cfreq codeword\n");

printf("---- ---- ------ --------------------\n");

for(i = 0;i < c->n\_symbols;i++)

printf("%4d %4d %6d %s\n",i,c->data[i].scaled\_prob,c->data[i].cum\_scaled\_prob,c->data[i].codeword);

printf("---- ---- ------ --------------------\n\n");

free\_code(c);

return 0;

}

//

// Encode and decode a message (called with arguments -t [n\_symbols [message\_size [seed]]])

//

if(argc >= 2 && argc <= 5 && argv[1][0] == '-' && argv[1][1] == 't')

{

int n\_symbols,message\_size,seed;

code\_t \*c;

n\_symbols = (argc < 3) ? 3 : atoi(argv[2]);

message\_size = (argc < 4) ? 10 : atoi(argv[3]);

seed = (argc < 5) ? 1 : atoi(argv[4]);

srandom(seed);

c = new\_code(n\_symbols);

try\_it(c,message\_size,1);

free\_code(c);

return 0;

}

//

// Try the first N\_MEASUREMENTS seeds (called with arguments -x n\_symbols)

//

if(argc == 3 && argv[1][0] == '-' && argv[1][1] == 'x')

{

double t,t\_min,t\_max,t\_avg,t\_data[N\_MEASUREMENTS],u\_avg;

int u,u\_min,u\_max,u\_data[N\_MEASUREMENTS];

int seed,n\_symbols,i;

code\_t \*c;

//

double m\_avg,m\_data[N\_MEASUREMENTS];

int m,m\_min,m\_max;

//

n\_symbols = atoi(argv[2]);

if(n\_symbols == 2)

{

printf("# data for MAX\_MESSAGE\_SIZE equal to %d\n",MAX\_MESSAGE\_SIZE);

printf("# data for N\_OUTLIERS equal to %d\n",N\_OUTLIERS);

printf("# data for N\_VALID equal to %d\n",N\_VALID);

printf("#\n");

printf("# number of calls per message symbol lookahead symbols max considerations\n");

printf("# ----------------------------------- --------------------- ------------------\n");

printf("# ns min avg med max min avg med max min avg med max\n");

printf("#--- -------- -------- -------- -------- ---- ------ ---- ---- --- ---- ---- ----\n");

}

if(n\_symbols < 3 || n\_symbols > MAX\_N\_SYMBOLS)

{

fprintf(stderr,"main: bad number of symbols for the -x command line option\n");

exit(1);

}

t\_min = t\_max = 0.0;

u\_min = u\_max = 0;

m\_min = m\_max = 0.0;

for(seed = 1;seed <= N\_MEASUREMENTS;seed++)

{

srandom(seed);

c = new\_code(n\_symbols);

try\_it(c,MAX\_MESSAGE\_SIZE,0);

free\_code(c);

t = (double)\_number\_of\_calls\_ / (double)MAX\_MESSAGE\_SIZE;

u = \_max\_extra\_symbols\_;

m = MAX\_CONSIDER\_ONCE;

if(seed == 1 || t < t\_min)

t\_min = t;

if(seed == 1 || t > t\_max)

t\_max = t;

if(seed == 1 || u < u\_min)

u\_min = u;

if(seed == 1 || u > u\_max)

u\_max = u;

//

if(seed == 1 || m < m\_min)

m\_min = m;

if(seed == 1 || m > m\_max)

m\_max = m;

//

for(i = seed - 1;i > 0 && t\_data[i - 1] > t;i--) // inner loop of insertion sort!

t\_data[i] = t\_data[i - 1];

t\_data[i] = t;

for(i = seed - 1;i > 0 && u\_data[i - 1] > u;i--) // inner loop of insertion sort!

u\_data[i] = u\_data[i - 1];

u\_data[i] = u;

//

for(i = seed - 1;i > 0 && m\_data[i - 1] > m;i--)

m\_data[i] = m\_data[i - 1];

m\_data[i] = m;

//

}

t\_avg = u\_avg = m\_avg = 0.0;

for(i = N\_OUTLIERS;i < N\_MEASUREMENTS - N\_OUTLIERS;i++)

{

t\_avg += t\_data[i];

u\_avg += (double)u\_data[i];

m\_avg += m\_data[i];

}

t\_avg /= (double)(2 \* N\_VALID + 1);

u\_avg /= (double)(2 \* N\_VALID + 1);

m\_avg /= (double)(2 \* N\_VALID + 1);

printf("%4d %8.3f %8.3f %8.3f %8.3f %4d %6.1f %4d %4d %3d %3.1f %3.1f %4d\n",n\_symbols,t\_min,t\_avg,t\_data[N\_OUTLIERS + N\_VALID],t\_max,u\_min,u\_avg,u\_data[N\_OUTLIERS + N\_VALID],u\_max,m\_min,m\_avg,m\_data[N\_OUTLIERS + N\_VALID],m\_max);

return 0;

}

//

// Help message

//

fprintf(stderr,"usage: %s -s n\_symbols seed # show the code words of random code\n",argv[0]);

fprintf(stderr," %s -t [n\_symbols [message\_size [seed]]] # encode and decode a message\n",argv[0]);

fprintf(stderr," %s -x n\_symbols # try the first %d seeds\n",argv[0],N\_MEASUREMENTS);

return 1;

}

1. Resultados obtidos em *Ryzen 7 3700X* com uma velocidade base de *3,59 GHz*. E 16GB de memória RAM a 3333 MHz com CL16 em dual-channel. [↑](#footnote-ref-2)