universidade de aveiro

Algoritmos e Estruturas de Dados

Trabalho Prático 3

02/02/2021
Licenciatura em Engenharia Informática

Diana Elisabete Siso Oliveira, nº 98607, P2 (33%)
Miguel Rocha Ferreira, nº 98599, P2 (33%)
Paulo Guilherme Soares Pereira, nº 98430, P2 (33%)

Índice

1. Introdução

Com o desenvolvimento do presente projeto, visamos aprimorar as nossas capacidades de programação em C bem como elaborar de forma correta um código que permita a correta descodificação de um código binário não instantâneo.

Para tal, através de informações fornecidas previamente no enunciado e dos conhecimentos adquiridos ao longo do semestre na respetiva unidade curricular, é esperado que o resultado final do *script* descodifique corretamente uma mensagem aleatória codificada inicialmente, de forma a atender às expectativas do projeto.

Assim, contemplando as ideias base do enunciado, foi desenvolvida a solução posteriormente explicada no presente relatório.

2. Estruturas

2.1. Symbol_t

Na estrutura *symbol_t* é possível encontrar cinco variáveis, das quais quatro são inteiros e uma é do tipo *char*.

A variável *scaled_prob* é um inteiro proporcional à probabilidade de ocorrência de um símbolo, enquanto que a variável *cum_sclaed_prob* é um inteiro proporcional à probabilidade de ocorrência do respetivo ou de todos os anteriores símbolos. Já a variável *parent* representa o nó pai, ou seja, se o valor atribuído for igual -1 significa que não tem um nó pai e se o valor atribuído for igual ou maior que 0 esse valor representa o índice do nó pai. Encontramos ainda a variável *bit* cujo valor associado por representar que não encontramos qualquer informação, caso seja igual a -1, ou significa que acrescentamos este *bit* ao código pai, caso seja igual a 1 ou 0. Por último, a variável *codeword* traduz-se no código de Huffman completo e invertido.

2.2. Code_t

Na estrutura *code_t* podemos verificar a existência de três variáveis, das quais duas são inteiros e uma é do tipo *symbol_t*.

A variável *n_symbols* representa, tal como o nome indica, o número de símbolos e a variável *max_bits* simboliza o número máximo de *bits* da *codeword*. Por último, a variável *data* é um ponteiro para a estrutura *symbol_t* usada para construir a árvore de *Huffman* contendo os símbolos e os respetivos códigos.

2.3. Decoder_global_data

A estrutura *decoder_global_data* é utilizada para a descodificação e, para tal, possui dez variáveis, das quais seis são inteiros, duas são do tipo *long*, uma é do tipo *char* e uma é do tipo *code_t*.

As duas variáveis do tipo *long*, *number_of_calls* e *number_of_solutions*, significam, tal como o próprio nome diz, o número de chamadas da função recursiva e o número de soluções encontradas. Já a variável do tipo *char*, *encoded_message*, é um ponteiro que aponta para a zona de memória onde ficará armazenada a mensagem codificada, enquanto que as variáveis do tipo *int*, *original_message* e *decoded_message*, também são ponteiros que apontam para a zona de memória onde encontramos a mensagem original e a mensagem descodificada, respetivamente. Salienta-se que as mensagens original e

descodificada deverão ser semelhantes. Ainda sobre ponteiros, encontramos a variável *c*, do tipo *code_t*, que aponta para a zona de memória onde se encontra o código a ser usado. Temos também as variáveis *original_message_size*, *max_extra_symbols*, *max_encoded_message_size* e *max_decoded_message_size*, todas inteiros, representando respetivamente o tamanho da mensagem original, o maior número de símbolos extras, o tamanho máximo da mensagem codificada e o tamanho máximo da mensagem descodificada.

3. A função *recursive_decoder*

O programa que nos é fornecido é composto por várias funções que contribuem para o funcionamento do programa. No entanto, como a solução que realizámos consistiu apenas em construir a função *recursive_decoder*, é apenas desta que iremos falar.

A função começa por incrementar a variável *_number_of_calls_*, que corresponde ao número de vezes que a função recursiva é chamada.

De seguida iteramos pelo número de símbolos e, para cada símbolo, definimos uma variável de controlo, chamada *PossibleSymbol*, a 1.

A seguir, iteramos por cada *bit* da *codeword* do símbolo até chegarmos ao *bit* "\0" e comparamos com o *bit* de mesmo índice da mensagem codificada, verificando se são iguais. Caso algum não seja, a variável *PossibleSymbol* é colocada a 0, quebrando assim o ciclo, visto que basta um *bit* da *codeword* não corresponder à mensagem codificada para que a *codeword* não seja compatível.

Caso *PossibleSymbol* seja igual a 1, ou seja, os *bits* da *codeword* forem iguais aos da mensagem codificada, é possível chamar a função de forma recursiva e incrementar o número de soluções. Neste caso começamos por colocar no índice *decoded_idx* da variável *_decoded_message_* o valor do símbolo *i* correspondente à *codeword* compatível.

Agora, para finalizar, verificamos se a mensagem codificada ainda possui bits para serem descodificados ou se já foi totalmente descodificada.

No primeiro caso, vamos ainda verificar se o símbolo a analisar está certo, comparando com o símbolo na posição *decoded_idx* na *_original_message_* e, caso esteja, incrementamos a variável *good_decoded_size*. De seguida, chamamos a função recursiva com *encoded_idx+k*, como índice do primeiro elemento do resto da mensagem codificada, *decoded_idx+1* e *good_decoded_size*.

No segundo caso, como não há mais *bits*, significa que chegamos à solução. Incrementamos então a variável *_number_of_solutions_*.

Em caso de *dead-end* é também calculado o número de *_max_extra_symbols_*. Caso o valor da diferença de *decoded_idx* e de *good_decoded_size* seja maior que o número atual de *_max_extra_symbols_* o seu valor é atualizado para o valor dessa mesma diferença.

Verifique alguns exemplos dos resultados obtidos utilizando o código com diferentes *inputs* na próxima página.

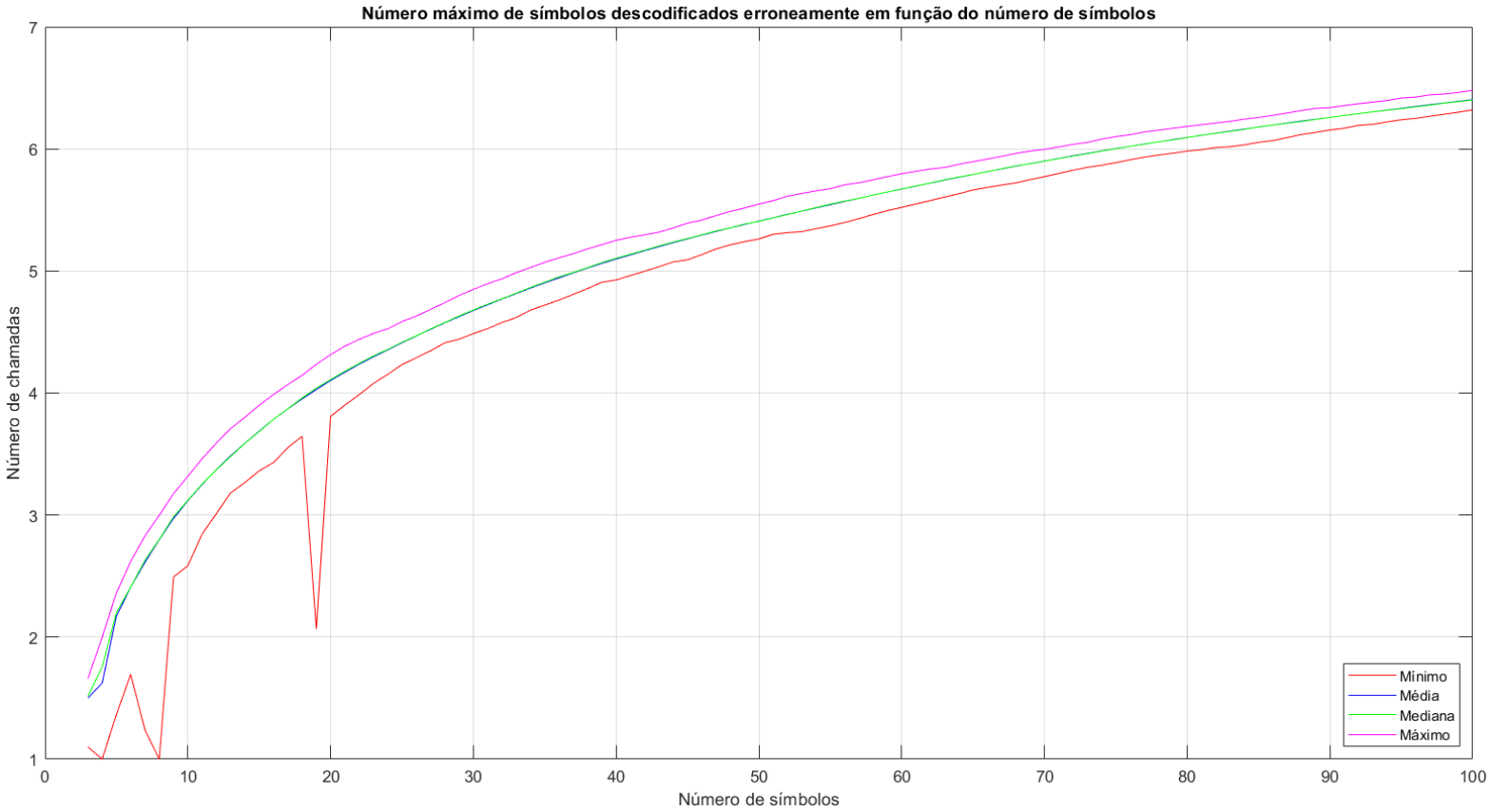
```
miguelferreira@miguel-mint:~/Desktop/PrinterGoesBrrr$ ./A03 -t 5 3 5
Encoded 011000
Decoded  023
Original 023
    5     1.667   1
```

```
miguelferreira@miguel-mint:~/Desktop/PrinterGoesBrrr$ ./A03 -t 12 9 4
Encoded 011100100111110100111110111
Decoded  7811396383
Original 7811396383
   12     1.000   0
```

```
miguelferreira@miguel-mint:~/Desktop/PrinterGoesBrrr$ ./A03 -t 6 4 10
Encoded 011010111
Decoded  4223
Original 4223
    6     1.250   1
```

4. Análise dos resultados obtidos

4.1. Número de chamadas da função por símbolo



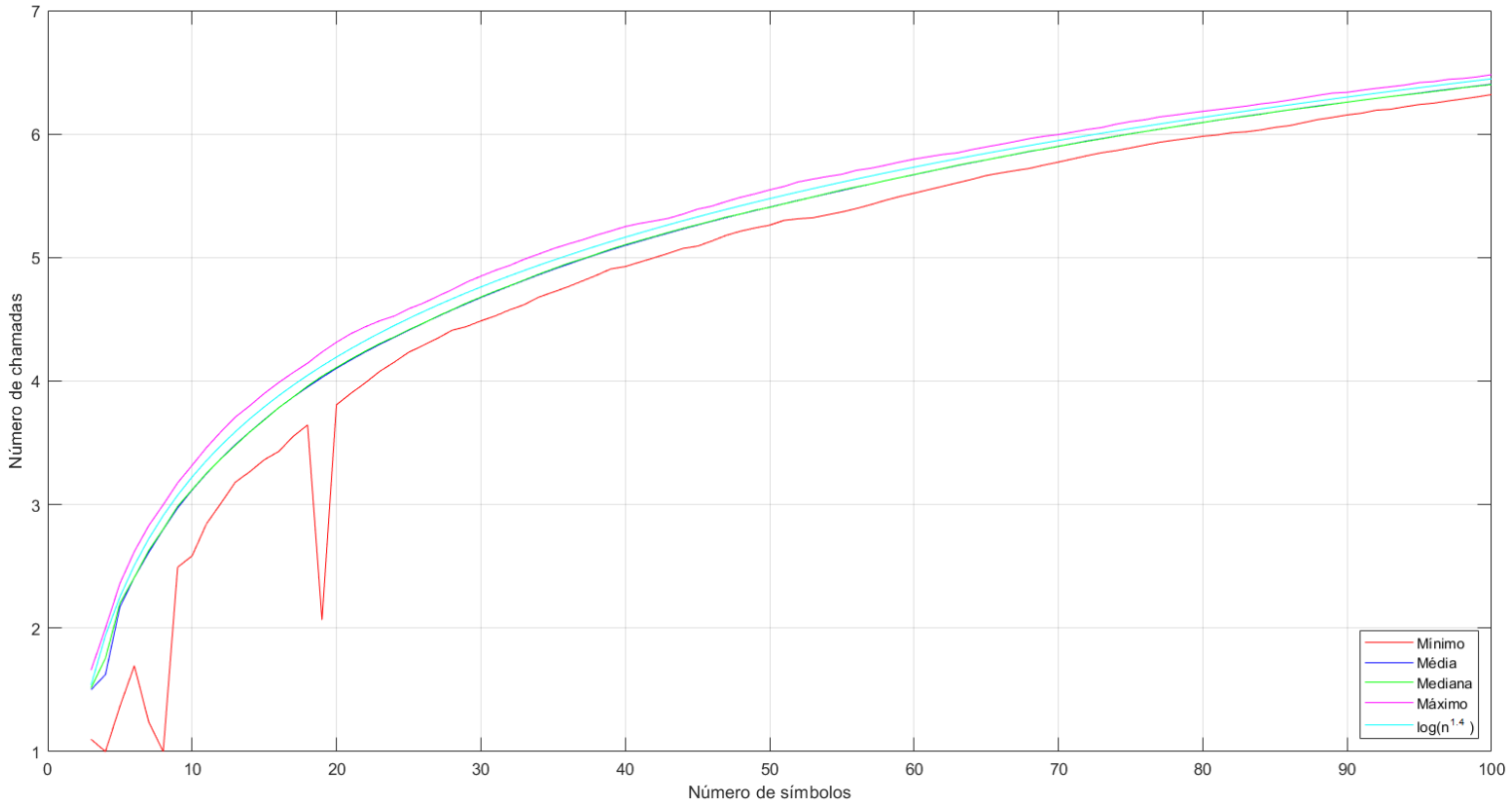
Por observação do gráfico, podemos averiguar que os valores da média e da mediana são bastante semelhantes, pelo que podemos afirmar com segurança que os dados são geralmente uniformes, o que torna a média uma medida confiável.

Verificamos ainda que a função cresce mais rápido no intervalo de símbolos [3, 30] do que no intervalo [30, 100], pelo que podemos afirmar também que, quando o número de símbolos tende para infinito, a função tende a ficar mais estável no que diz respeito ao número de chamadas.

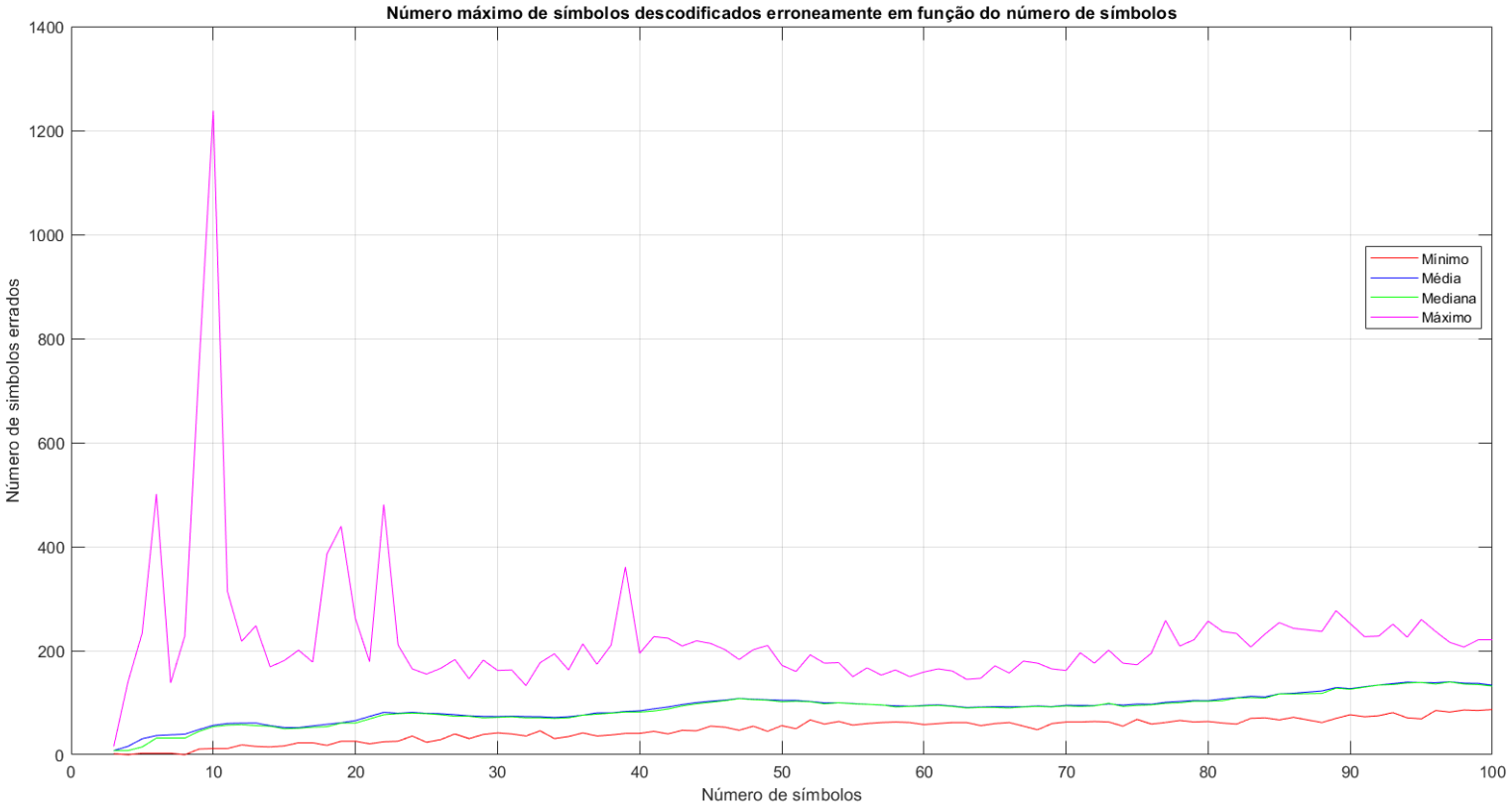
Ainda é possível concluir que à medida que o número de símbolos aumenta, a diferença entre o máximo e o mínimo deixa de ser tão acentuada.

Individualmente, é notório que inicialmente a reta correspondente ao mínimo valor enfrenta alguns picos e depressões, diferente das restantes que se mantêm estáveis durante todo o processo, no entanto a mesma tende a estabilizar-se posteriormente.

Podemos ainda deduzir que a complexidade da função aproxima-se de $\Theta(log(n))$pelo seguinte gráfico:



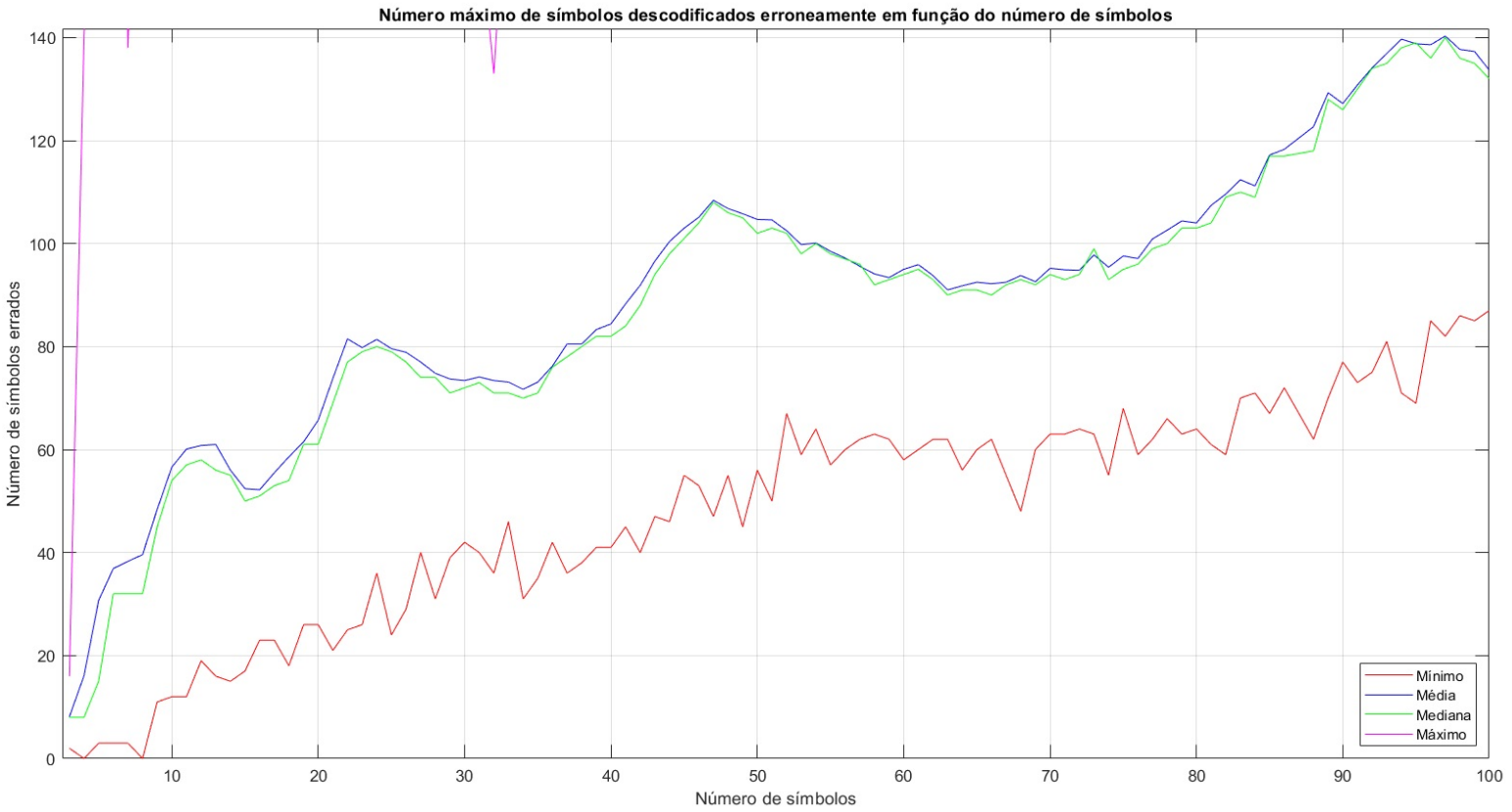
4.2. Número máximo de símbolos descodificados erroneamente



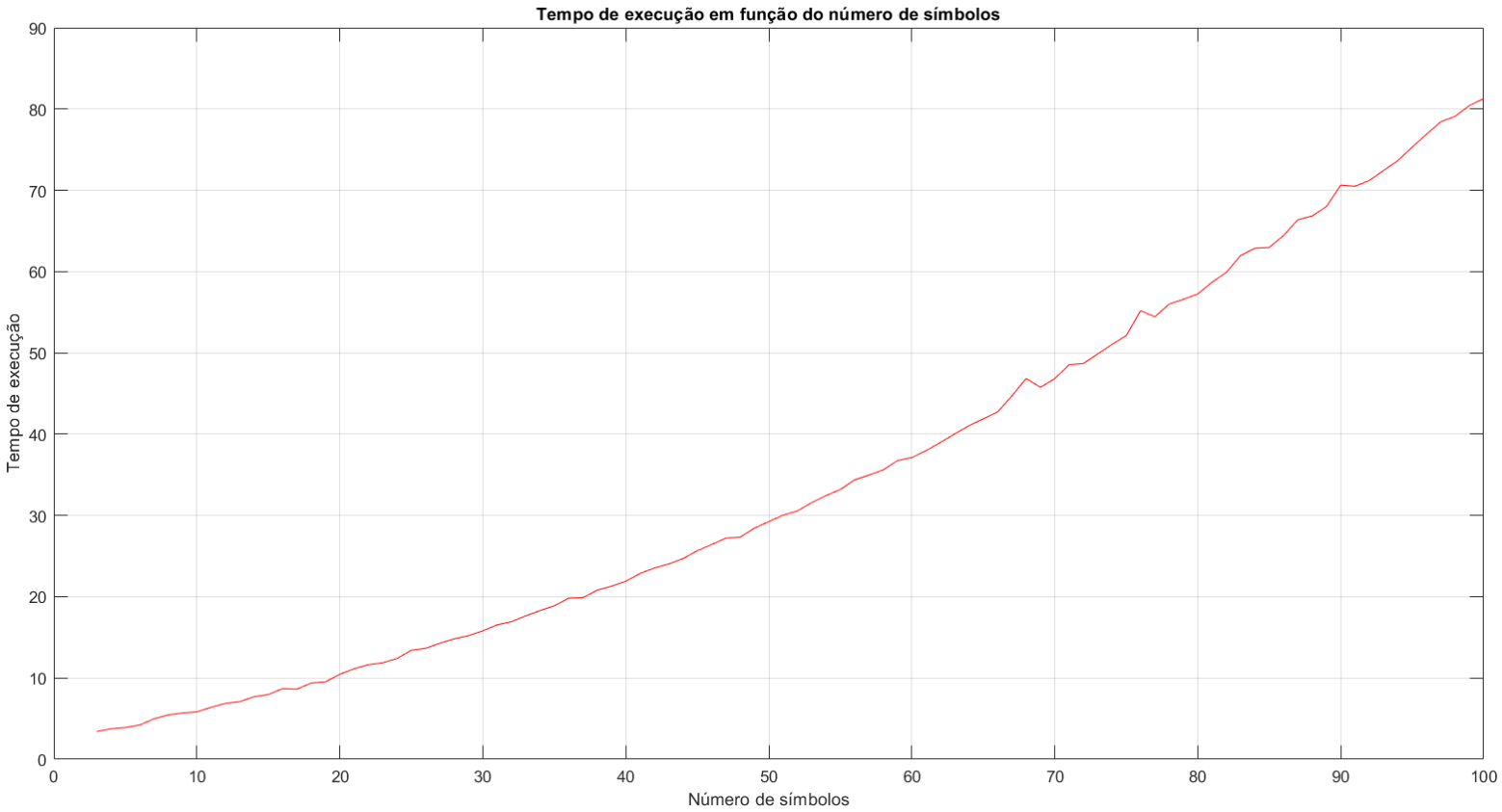
Por observação do gráfico, podemos averiguar que os valores da média e da mediana são bastante semelhantes, pelo que podemos afirmar com segurança que os dados são geralmente uniformes, o que torna a média uma medida confiável.

Individualmente, é possível notar que a função corresponde ao máximo não assume um padrão constante, tendo mais oscilações para um número pequeno de símbolos. O mesmo não se verifica nas restantes retas, contudo não podemos dizer que o crescimento delas é constante, apenas que não possui picos e depressões tão acentuadas, como podemos observar com uma ampliação do gráfico.

De forma geral, podemos concluir que o número de símbolos errados cresce à medida que o número de símbolos aumenta, no entanto, e mais uma vez por observação do gráfico ampliado, podemos averiguar que não é algo constante.



4.3. Tempo de execução do programa



$$y = 0.005186x^2 + 0.2714x + 2.819$$

Ao executarmos o ficheiro *do_all.bash*, obtivemos impressos no terminal os tempos de execução para cada valor de n, com n a variar num intervalo de 3 a 100, correspondentes à execução do programa com a seguinte chamada "./A03 -x n", sendo n o valor de n referido anteriormente.

Neste gráfico podemos observar que, apesar de pequenos picos e depressões, o tempo de execução, de forma geral, assemelha-se a uma função exponencial de ordem quadrática (2). Isto revela que, para cada n, o tempo de execução será aproximadamente n^2.

5. Anexo

5.1. Código em C

(O código a vermelho foi o código alterado na execução do trabalho)

```
//
// AED, 2020/2021
//
// Decoding a non-instantaneous binary code
//
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
//
// Compile time parameters
//
#ifndef MAX_N_SYMBOLS
# define MAX_N_SYMBOLS         100 // 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_si
ze + 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_s
ize
#define _max_encoded_message_size_
decoder_global_data.max_encoded_messag
e_size
#define _encoded_message_
decoder_global_data.encoded_message
#define _max_decoded_message_size_
decoder_global_data.max_decoded_messag
e_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_solution
s
#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")
//
static void recursive_decoder(int
encoded_idx,int decoded_idx,int
good_decoded_size)
{ //
 _number_of_calls_++;
 int k;
 for(int i = 0;i < _c_->n_symbols;i++)
 {
   int PossibleSymbol=1;
   for
(k=0;_c_->data[i].codeword[k]!='\0';k+
+)
   {
     if(_c_->data[i].codeword[k]
!=_encoded_message_[encoded_idx+k])
     {
       PossibleSymbol=0;
       break;
     }
   }
   if (PossibleSymbol)
   {
_decoded_message_[decoded_idx]=i;
     if (_encoded_message_[encoded_idx
+ k] != '\0')
     {
       if
(_original_message_[decoded_idx] == i
&& decoded_idx == good_decoded_size)
       {
         good_decoded_size++;
       }
       recursive_decoder(encoded_idx +
k , decoded_idx +1 ,
good_decoded_size);
     }
     else
     {
       _number_of_solutions_ ++;
     }
   }
 }

f(decoded_idx-good_decoded_size>_max_e
xtra_symbols_)
 {
_max_extra_symbols_=decoded_idx-good_d
ecoded_size;
 }
}
//
// 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_siz
e_ * 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_s
ize_,_original_message_);
encode_message(_c_,_original_message_s
ize_,_original_message_,_max_encoded_m
essage_size_,_encoded_message_);
 recursive_decoder(0,0,0);
 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_);
 }
 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_e
xtra_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].codewor
d);
   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_MEASUREME
NTS],u_avg;
   int
u,u_min,u_max,u_data[N_MEASUREMENTS];
   int seed,n_symbols,i;
   code_t *c;
   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\n");
     printf("#
---------------------------------
--------------------\n");
```

```
     printf("# ns       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;
   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_;
     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;
     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;
   }
   t_avg = u_avg = 0.0;
   for(i = N_OUTLIERS;i <
N_MEASUREMENTS - N_OUTLIERS;i++)
   {
     t_avg += t_data[i];
     u_avg += (double)u_data[i];
   }
   t_avg /= (double)(2 * N_VALID + 1);
   u_avg /= (double)(2 * N_VALID + 1);
   FILE *dados;

    dados = fopen("dados.txt","a");
    if(dados == NULL)
    {
      fprintf(dados,"Unable to create
file (maybe it already exists? If so,
delete it!)\n");
      exit(1);
    }

    fprintf(dados,"%4d  %8.3f %8.3f
%8.3f %8.3f  %4d %6.1f %4d
%4d\n",n_symbols,t_min,t_avg,t_data[N_
OUTLIERS +
N_VALID],t_max,u_min,u_avg,u_data[N_OU
TLIERS + N_VALID],u_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;
}
```

5.2. Código em MATLAB

```
fid = fopen('dadosA03.txt');
simbolos = zeros(97,1);
u_min = zeros(97,1);
u_max = zeros(97,1);
u_avg = zeros(97,1);
u_data = zeros(97,1);
for i=1:97
    linha = fgetl(fid);
    linha = split(linha);
    simbolos(i) = str2double(linha{1});
    u_min(i) = str2double(linha{6});
    u_avg(i) = str2double(linha{7});
    u_data(i) = str2double(linha{8});
    u_max(i) = str2double(linha{9});
end
plot(simbolos,u_min, 'r-', simbolos, u_avg,
'b-', simbolos, u_data, 'g-', simbolos,
u_max, 'm-');
legend('Mínimo', 'Média', 'Mediana',
'Máximo', 'Location','Southeast');
set(gcf, 'color', 'w');
title('Número máximo de símbolos
descodificados erroneamente em função do
número de símbolos');
xlabel('Número de símbolos');
ylabel('Número de simbolos errados');
grid;
fclose(fid);
%usamos o mesmo código mas adaptado para
fazer os outros gráficos
```