Algoritmos e Estruturas de Dados
Licenciatura em Engenharia Informática
Ano Letivo 2020/2021 2ºAno 1º Semestre

Relatório do Trabalho Prático 3

Recursively decoding a non-instantaneous binary code

Trabalho realizado por:
Diogo Cruz (98595) - 30%
Gonçalo Leal (98008) - 40%
Sophie Pousinho (97814) - 30%

Índice

Introdução - Trabalho Prático

O objetivo deste trabalho é decodificar, de forma recursiva, mensagens geradas aleatoriamente que foram codificadas com um determinado número de bits através de símbolos pertencentes a um dicionário, também gerados aleatoriamente pelo programa.

Para além disto, devemos considerar que a codificação fornecida no dicionário é não instantânea, ou seja, à partida não sabemos quando temos de parar de analisar a codificação para obter um símbolo, o que significa que o código é ambíguo uma vez que a mesma codificação pode ter mais que uma representação.

No decorrer desta descodificação pretendemos analisar as mensagens descodificadas e o número de chamadas à função recursiva, por símbolo da mensagem, até terminar a análise da mensagem ou até descobrir um beco sem saída, ou seja, uma descodificação inválida.

Introdução - Teoria

Comecemos por perceber melhor como funciona a instantaneidade de um código binário:

Um código instantâneo é um código com que é possível descodificar imediatamente cada símbolo a partir do momento que este nos é apresentado. Estes códigos podem ser obtidos através de árvores binárias (ex. Código de Huffman, código de Shannon-Fano, etc.)

Neste trabalho foi-nos proposto um código, como já foi mencionado anteriormente, não instantâneo, ou seja, só conseguimos descodificar um símbolo através dos bits seguintes. Estes códigos já não são possíveis de ser obtidos através de árvores.

Símbolo	Código
A	0
B	01
C	001

Este código é **não instantâneo**.
Sempre que surge um novo bit 0, é necessário esperar pelos bits seguintes (à partida, não se sabe quantos) para se identificar a chegada de um novo símbolo.

symbol	codeword
A	0
B	011
C	01
D	111

This code is uniquely decodable (it is our first code in reverse, so decoding from the end to the beginning is easy!), but it is not instantaneous. If the beginning of an encoded message is 0111 it is not possible to decide without further bits if the message starts with AD (encoded as 0 111) or with BD (encoded as 011 111).

Código - Função Recursiva

Como a codificação é ambígua temos de escolher um caminho e, à medida que vamos avançando tentamos perceber se estamos no caminho correto ou não. Se tivermos escolhido o caminho errado então vamos chegar a um ponto onde a nossa função não vai conseguir identificar nenhum símbolo que corresponda à sequência de bits que falta descodificar.
Após uma pesquisa exaustiva de todas as possibilidades, só um dos caminhos escolhidos deve levar à solução, ou seja, só um caminho permite transformar todos os bits no respetivos símbolos da mensagem original.

---- Recursive Decoder ---

Parâmetros de entrada:

- *encoded_idx:* posição atual na mensagem codificada.
- *decoded_idx:* posição atual na mensagem descodificada (nº de símbolos descodificados).
- *good_decoded_size:* número de símbolos descodificados corretamente.

Explicação

Numa primeira fase, são feitas cópias locais das variáveis passadas como parâmetros da função e a variável que indica o número de bits que estamos a considerar para o código de um símbolo, *symb_size*, é inicializada a 1. Assim, começa uma sequência de bits de tamanho symb_size na mensagem codificada que corresponde a um dos códigos do nosso dicionário.

De seguida, aumenta-se o número de chamadas à função recursiva, apenas para dados estatísticos.

```
int initial_encoded_idx = encoded_idx;
int decoded_pos = decoded_idx;
int correct_decoded_size = good_decoded_size;
int symb_size = 1;

_number_of_calls_++;
```

Figura 1: Variáveis locais e aumento do número de chamadas à função

Como já foi mencionado, o código é ambíguo e vamos chegar a uma altura em que o caminho que estamos a percorrer é errado. Então, pretendemos saber quantos símbolos errados descodificamos até não conseguirmos avançar mais naquele caminho, esse valor fica guardado na variável *_max_extra_symbols_*. Podemos obter esse valor fazendo a diferença entre os símbolos descodificados, *decoded_pos*, e os símbolos descodificados corretamente, *correct_decoded_size*. Se esta diferença for maior que o máximo até ao momento, passa a ser o novo máximo.

```
if (decoded_pos - correct_decoded_size > _max_extra_symbols_){
  _max_extra_symbols_ = decoded_pos - correct_decoded_size;
}
```

Figura 2: Atualização do número máximo de símbolos descodificados erradamente

Em relação à parte da descodificação exatamente, para descodificar a mensagem, temos de ir procurando os símbolos que tenham um código igual à sequência de bits de tamanho *symb_size* que estamos a considerar.

Para tal, usamos um ciclo while que termina quando o tamanho da sequência de bits, ou seja, o *symb_size*, for maior que o tamanho máximo que um código pode ter, *max_bits*, ou quando a mensagem codificada terminar, *_encoded_message_[initial_encoded_idx + symb_size -1] == 0.*

```
// se (symb_size) > _c_->max_bits então não é possível descodificar mais
// verificar se a mensagem já acabou
while (symb_size <= _c_->max_bits && _encoded_message_[initial_encoded_idx + symb_size - 1] != 0){
```

Figura 3: Verificação do ciclo while

Enquanto as condições do while se verificam, percorremos todos os símbolos do nosso dicionário.

```
for (int j = 0 ; j < _c_->n_symbols; j++){
  // length de um símbolo
  int i, h;
  for (i = 0; _c_->data[j].codeword[i]; i++);
```

Figura 4: Passagem pelos elementos do dicionário

Se encontrarmos algum código com tamanho *symb_size* que seja igual à sequência de bits que temos então, adicionamos o símbolo correspondente à mensagem descodificada.

Se, até ao momento, a mensagem descodificada estiver correta e o símbolo que está a ser adicionado corresponder ao símbolo daquela posição na mensagem original, então, verificamos se já chegámos ao fim da mensagem e aumentamos o número de soluções e damos return, se essa condição se verificar. Se não, chamamos a nossa função recursiva para continuar a descodificar a mensagem, aumentando o *enconded_idx* para *encoded_idx* + *symb_size,* e incrementamos os outros dois parâmetros da função em uma unidade.

Se, por outro lado, a mensagem descodificada estiver errada, mas ainda for possível continuar a descodificar, chamamos a nossa função recursiva aumentando o *encoded_idx* para o valor recebido + *symb_size* e incrementando o *decoded_size*. Neste caso o *correct_decode_size* mantém-se.

```
// o símbolo _c_->data[j].codeword é do tamanho do símbolo que estamos a considerar?
if (i == symb_size){
  // as strings são do mesmo tamanho, por isso quando _c_->data[j].codeword[h] acabar já comparamos os dois símbolos
  for (h = 0; _c_->data[j].codeword[h] && _encoded_message_[initial_encoded_idx + h] &&
  _c_->data[j].codeword[h] == _encoded_message_[initial_encoded_idx + h]; h++);

  // se o i e o h forem iguais quer dizer que os símbolos são iguais
  if (i == h){
    _decoded_message_[decoded_pos] = j;
    int idx = initial_encoded_idx + symb_size;
    // se o decoded_idx - good_decoded_size não for zero quer dizer que a mensagem já está errada,
    // se não está veerifica se está certa
    if (decoded_pos - correct_decoded_size == 0 && j == _original_message_[decoded_pos]){
      // o decoded_pos será sempre menor que o tamanho da mensagem original por ser um index
      if (decoded_pos == _original_message_size_ - 1){
        // printf("--------------------SOLUÇÃO ENCONTRADA----------------------\n");
        _number_of_solutions_++;
      }

      recursive_decoder(idx, decoded_pos+1, correct_decoded_size+1);
    }
    // a mensagem não está certa, mas é possível continuar a descodificar
    else{
      // printf("Já está errada\n");
      recursive_decoder(idx, decoded_pos+1, correct_decoded_size);
    }
    // printf("endoded_idx = %d; decoded_idx = %d; good_decoded_size = %d\n", initial_encoded_idx + symb_size,
    // decoded_pos, correct_decoded_size);
  }

}
```

Figura 5: Tratamento da mensagem descodificada

O symb_size é incrementado a cada iteração no ciclo while.

```
symb_size++;
```

Figura 6: Incremento do número de bits que estamos a considerar para o código de um símbolo

Por fim, quando o ciclo while acabar damos return. No final, esperamos ter encontrado apenas uma solução.

--- Alternativa ao uso da biblioteca string.h ---

De forma a não usar a biblioteca string.h, que não estava incluída no código fornecido, criámos formas alternativas às funções strlen e strcmp.

- strlen()

```
for (i = 0; _c_->data[j].codeword[i]; i++);
```

Figura 7: Comprimento de uma string

Uma forma de descobrir o comprimento de uma string é através de um ciclo for. É necessário definir o i previamente para que seja possível usar o seu valor fora do ciclo. O ciclo termina quando o caractere na posição i da string for '\0'. Embora esse caractere não conte para o tamanho da string, como i começa em 0 então iremos obter o tamanho correto da string.

- strcmp()

```
for (h = 0; _c_->data[j].codeword[h] && _encoded_message_[initial_encoded_idx + h]
&& _c_->data[j].codeword[h] == _encoded_message_[initial_encoded_idx + h]; h++);
```

Figura 8: Comparar duas string

Para comparar duas strings e concluir que são iguais o comprimento das strings tem de ser igual e todos os caracteres têm de ser os mesmos. Neste caso, este ciclo for é antecedido de um if que verifica se as duas strings têm o mesmo tamanho. Para verificar se são iguais, percorremos ambas as strings até que uma delas termine (deverão terminar ao mesmo tempo, mas é uma segunda verificação) e enquanto os caracteres da posição h das strings sejam iguais.

Os códigos das figuras 7 e 8 devem ser usados em conjunto. Apenas se i e h forem iguais é que podemos afirmar que as strings são realmente iguais.

Resultados

Obtenção de resultados

Os resultados foram gerados usando as constantes:

```
#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
```

Figura 7: Constantes usadas

Para obtermos os resultados corremos o programa desenvolvido através do script bash *do_all.bash* fornecido pelo professor. Depois, criamos um script bash (*get_results.bash*) que fosse buscar estes valores aos vários ficheiros gerados por cada chamada à função.

```
#! /bin/bash

for n in {100..3}; do
  if [ -e stop_request ]; then
    exit 0
  fi
  f=$(printf %04d $n)
  export TIMEFORMAT="$n done in %3Us"
  if [ ! -e $f ]; then
    time ./A03 -x $n >$f
  fi
done
echo All done
```

Figura 8: Script do_all.bash

```
#! /bin/bash

for n in {3..100}; do
  f=$(printf %04d $n)
  sed -n '6p' < $f >> results.txt;
done
echo All done
```

Figura 9: Script get_results.bash

Por fim, escrevemos um script em matlab que lesse o ficheiro results.txt gerado a partir do script *get_results.bash* e gerasse os gráficos e as tabelas apresentadas mais à frente nesta secção do relatório.

```
results = readcell('results.txt');

n_seeds = [results{:, 1}];
n_call_per_symbol_min = [results{:, 2}];
n_call_per_symbol_avg = [results{:, 3}];
n_call_per_symbol_med = [results{:, 4}];
n_call_per_symbol_max = [results{:, 5}];
lookahead_symb_min = [results{:, 6}];
lookahead_symb_avg = [results{:, 7}];
lookahead_symb_med = [results{:, 8}];
lookahead_symb_max = [results{:, 9}];
```

Figura 10: Script matlab parte 1 - leitura dos dados

```
plot(n_seeds, n_call_per_symbol_min, 'r--');
hold on;
plot(n_seeds, n_call_per_symbol_avg, 'g-x');
hold on;
plot(n_seeds, n_call_per_symbol_max, 'b-o');
hold off;

xlabel('Number of seeds');
ylabel('Number of calls');
title('Number of calls to the recursive function by number of seeds');
legend('min', 'avg', 'max');
% este é logaritmico

%-----------------------------------------------
figure(2);
plot(n_seeds, lookahead_symb_min, 'r--');
hold on;
plot(n_seeds, lookahead_symb_avg, 'g-x');
hold on;
plot(n_seeds, lookahead_symb_max, 'b-o');
hold off;

xlabel('Number of seeds');
ylabel('Number of lookahead symbols');
title('Number of lookahead symbols by number of seeds');
legend('min', 'avg', 'max');
```

Figura 11: Script matlab parte 2 - gráficos

```
T1 = table(n_seeds', n_call_per_symbol_min', n_call_per_symbol_avg', n_call_per_symbol_med', n_call_per_symbol_max');
T1.Properties.VariableNames = {'N Seeds' 'Min Calls Per Symbol' 'Avg Calls Per Symbol' 'Med Calls Per Symbol' 'Max Calls Per Symbol'};
T1
filename = 'tabela1.xlsx';
writetable(T1,filename,'Sheet',1);

T2 = table(n_seeds', lookahead_symb_min', lookahead_symb_avg', lookahead_symb_med', lookahead_symb_max');
T2.Properties.VariableNames = {'N Seeds' 'Min Lookahead Symbols' 'Avg Lookahead Symbols' 'Med Lookahead Symbols' 'Max Lookahead Symbols'};
T2
filename = 'tabela2.xlsx';
writetable(T2,filename,'Sheet',1);
```

Figura 12: Script matlab parte 3 - tabelas de resultados

Análise de Resultados

N Seeds	N Seeds	N Seeds	N Seeds	N Seeds
3	1,1	1,501	1,516	1,66
4	1	1,624	1,756	1,999
5	1,367	2,172	2,201	2,362
6	1,695	2,41	2,41	2,62
7	1,239	2,613	2,629	2,83
8	1	2,799	2,801	2,999
9	2,493	2,973	2,986	3,176
10	2,585	3,118	3,122	3,318
11	2,845	3,251	3,254	3,462
12	3,011	3,373	3,371	3,59
13	3,182	3,486	3,481	3,709
14	3,267	3,589	3,588	3,801
15	3,363	3,688	3,685	3,9
16	3,43	3,784	3,784	3,988
17	3,552	3,87	3,871	4,069
18	3,645	3,952	3,959	4,145
19	2,067	4,029	4,04	4,235
20	3,808	4,103	4,11	4,314

21	3,901	4,171	4,178	4,384
22	3,986	4,236	4,242	4,439
23	4,079	4,297	4,304	4,488
24	4,153	4,354	4,357	4,526
25	4,233	4,411	4,416	4,585
26	4,289	4,467	4,468	4,63
27	4,346	4,522	4,525	4,686
28	4,411	4,575	4,577	4,74
29	4,441	4,626	4,631	4,8
30	4,487	4,676	4,68	4,849
31	4,527	4,723	4,728	4,896
32	4,577	4,77	4,772	4,936
33	4,618	4,815	4,818	4,986
34	4,679	4,86	4,865	5,028
35	4,72	4,903	4,908	5,071
36	4,762	4,944	4,951	5,108
37	4,809	4,984	4,988	5,142
38	4,856	5,024	5,027	5,181
39	4,908	5,062	5,068	5,215
40	4,927	5,098	5,104	5,251
41	4,963	5,132	5,137	5,276
42	4,998	5,166	5,17	5,296
43	5,035	5,199	5,204	5,318
44	5,074	5,232	5,236	5,352
45	5,092	5,263	5,266	5,392
46	5,134	5,294	5,296	5,416
47	5,18	5,324	5,328	5,454
48	5,214	5,353	5,353	5,488

49	5,24	5,381	5,384	5,516
50	5,262	5,408	5,406	5,548
51	5,301	5,436	5,435	5,576
52	5,314	5,464	5,462	5,613
53	5,322	5,49	5,491	5,635
54	5,346	5,517	5,519	5,656
55	5,369	5,543	5,547	5,674
56	5,397	5,57	5,572	5,706
57	5,428	5,595	5,595	5,723
58	5,463	5,622	5,622	5,747
59	5,494	5,647	5,647	5,773
60	5,521	5,671	5,672	5,797
61	5,549	5,696	5,697	5,816
62	5,577	5,72	5,721	5,835
63	5,605	5,745	5,747	5,848
64	5,633	5,768	5,77	5,874
65	5,664	5,791	5,791	5,896
66	5,685	5,814	5,813	5,917
67	5,704	5,836	5,837	5,939
68	5,722	5,859	5,861	5,963
69	5,749	5,879	5,878	5,982
70	5,773	5,901	5,9	5,996
71	5,799	5,922	5,921	6,016
72	5,825	5,944	5,942	6,037
73	5,849	5,963	5,961	6,053
74	5,866	5,983	5,982	6,08
75	5,888	6,003	6,002	6,101
76	5,911	6,021	6,021	6,116

77	5,933	6,04	6,039	6,139
78	5,95	6,058	6,058	6,153
79	5,965	6,077	6,075	6,169
80	5,982	6,094	6,093	6,184
81	5,993	6,112	6,113	6,198
82	6,011	6,129	6,128	6,212
83	6,018	6,146	6,144	6,226
84	6,033	6,162	6,16	6,244
85	6,054	6,179	6,18	6,257
86	6,068	6,196	6,196	6,275
87	6,096	6,211	6,21	6,293
88	6,118	6,228	6,225	6,314
89	6,135	6,243	6,242	6,332
90	6,155	6,258	6,258	6,338
91	6,169	6,274	6,274	6,355
92	6,193	6,289	6,289	6,37
93	6,201	6,304	6,304	6,383
94	6,221	6,318	6,317	6,397
95	6,239	6,332	6,33	6,417
96	6,25	6,348	6,345	6,425
97	6,268	6,362	6,359	6,443
98	6,284	6,376	6,375	6,45
99	6,301	6,39	6,388	6,463
100	6,32	6,404	6,401	6,481

Tabela 1: Chamadas à função por símbolo

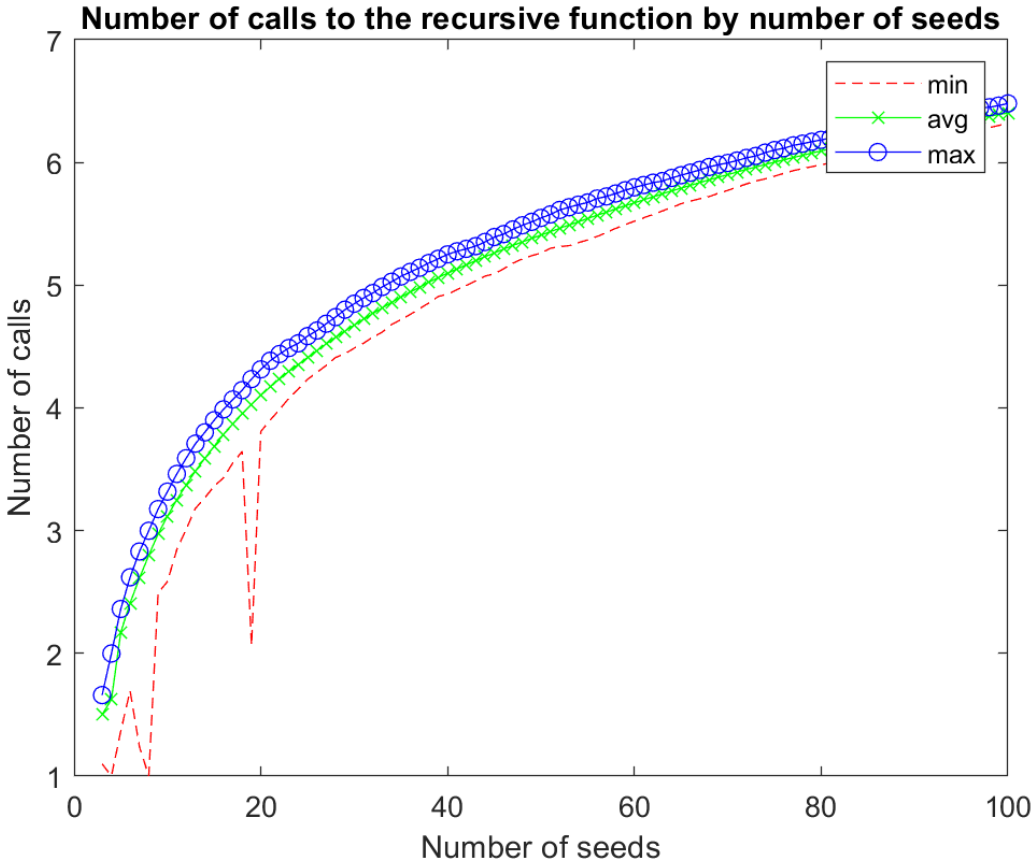

Figura 13: Chamadas à função por símbolo

Através da observação do gráfico percebemos que o crescimento do número de chamadas à função por símbolo à medida que o número de seeds aumenta vai crescendo de forma logarítmica. Talvez, se correrrmos o código para um maior número de casos (por exemplo até um alfabeto de 1000 símbolos) iremos chegar a um ponto em que o número de chamadas à função por símbolo quase não varia. Para o número máximo e médio de chamadas o gráfico segue uma linha contínua, sem grandes variações visíveis a olho nú. No entanto, para o número mínimo observamos duas mudanças abruptas. Provavelmente, se gerássemos novos resultados o gráfico seria diferente, porque os resultados estão muito dependentes da aleatoriedade da mensagem gerada. O facto de o número de chamadas à função por símbolo terem um valor mínimo tão baixo pode dever-se ao caso de a mensagem encriptada não ser tão ambígua quanto se desejava. O mesmo podia acontecer para o caso máximo, mas a média como é feita a partir de um número elevado de medições (201) à partida não terá grandes variações.

A partir de 90 símbolos no alfabeto vemos que o número máximo, médio e mínimo começam a convergir para o mesmo valor de número de chamadas por símbolo.

N Seeds	Min Lookahead Symbols	Avg Lookahead Symbols	Med Lookahead Symbols	Max Lookahead Symbols
3	2	8,6	9	19
4	0	16,5	8	141
5	3	31,3	15	235
6	3	37,6	32	502
7	4	39	33	139
8	0	40,2	32	228
9	11	49,1	46	747
10	13	57,1	54	1239
11	13	60,6	58	315
12	20	61,3	58	219
13	16	61,5	57	249
14	16	56,6	55	169
15	17	53	51	182
16	23	52,7	51	201
17	24	55,9	54	179
18	19	59,1	55	387
19	26	62,1	62	439
20	27	66,2	61	263
21	22	74,4	70	179
22	26	82,1	78	481
23	26	80,3	79	211
24	36	81,9	80	166

25	24	80,2	79	156
26	30	79,4	78	166
27	40	77,5	74	183
28	32	75,3	74	146
29	40	74,1	72	183
30	43	73,9	73	162
31	41	74,5	73	163
32	37	73,8	72	134
33	47	73,7	72	178
34	32	72,1	70	195
35	36	73,5	71	164
36	42	76,6	76	214
37	37	81	78	175
38	39	81	80	211
39	42	83,9	82	361
40	41	84,9	82	196
41	46	88,7	85	229
42	41	92,4	89	224
43	47	97,1	94	209
44	47	100,9	99	219
45	56	103,6	103	214
46	53	105,7	105	202
47	47	108,9	108	184
48	56	107,2	106	203
49	46	106,3	106	211
50	56	105,3	103	173
51	51	105	104	161
52	67	103	103	192

53	60	100,3	99	177
54	64	100,6	100	177
55	58	99	98	151
56	60	97,7	97	168
57	62	96,1	96	154
58	63	94,5	93	164
59	63	94	94	151
60	58	95,6	94	159
61	60	96,4	96	165
62	63	94,3	93	162
63	63	91,5	91	146
64	57	92,2	91	147
65	60	93	92	171
66	62	92,8	91	158
67	55	93	93	181
68	48	94,3	94	176
69	61	93,2	92	167
70	64	95,6	94	163
71	63	95,4	94	197
72	65	95,3	94	177
73	64	98,3	99	202
74	55	95,9	94	176
75	69	98,1	95	173
76	60	97,6	97	195
77	63	101,4	100	259
78	66	103,1	100	210
79	64	105	104	221
80	64	104,6	104	257

81	61	108	105	237
82	60	110,1	109	233
83	70	112,9	111	207
84	71	111,7	109	233
85	68	117,7	117	254
86	73	118,9	118	243
87	71	119,3	116	272
88	62	123,2	119	238
89	71	129,8	128	277
90	77	127,7	126	253
91	73	131,3	131	227
92	76	134,6	135	228
93	82	137,3	136	252
94	71	140,2	139	226
95	69	139,3	139	261
96	87	139,1	136	237
97	82	140,8	140	216
98	86	138,1	137	207
99	86	137,7	135	221
100	88	134,1	132	222

Tabela 2: Quantidade símbolos descodificados erradamente

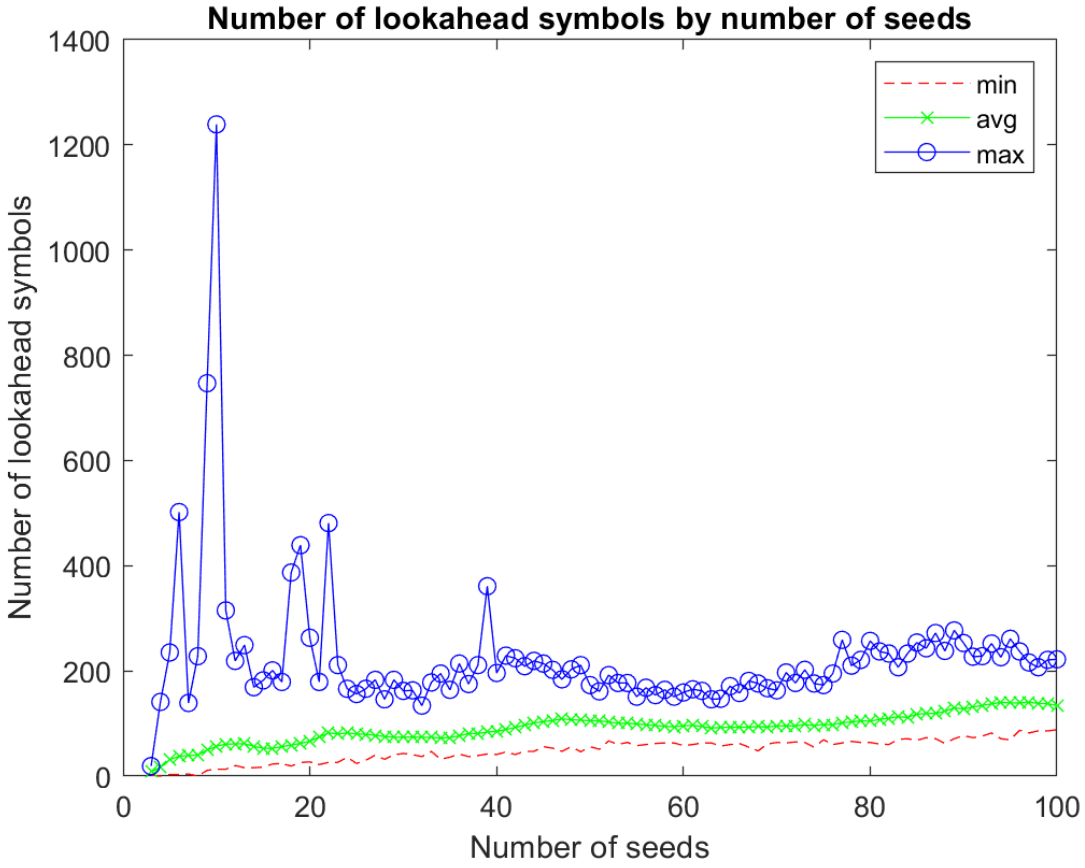

Figura 14: Símbolos descodificados erradamente

O número de símbolos descodificados erradamente tem um padrão diferente ao do número de chamadas à função por símbolo. Neste caso, o gráfico tem picos nos pontos do tipo 1.5 x potência de base 2. Ao contrário do número de chamadas à função por símbolo, neste gráfico é o número máximo de lookahead symbols que tem uma maior variação face à média. No entanto, vemos que com o aumento do número de símbolos no alfabeto estes valores tendem a estabilizar. Talvez, se aumentássemos o número de amostras, este valor seguisse o gráfico da média.

Caraterísticas do PC

- HP EliteBook 840 G3
- Intel(R) Core(™) i7-6500U CPU @ 2.50GHz 2.59GHz
- 16.0GB de RAM

Foi usada uma máquina virtual através da Oracle VM VirtualBox com o Linux Ubuntu instalado. Esta máquina tem as seguintes características:

- 8GB de RAM
- 2 dos 4 processadores lógicos do host

Conclusão

Em suma, consideramos que este trabalho, tal como os anteriores, foi uma oportunidade para percebermos melhor o funcionamento das funções recursivas, já que foi um tema ainda pouco abordado ao longo do nosso percurso académico.

Em termos da solução encontrada para este problema, poderíamos ter atingido valores mais credíveis se tivéssemos aumentado o número de amostras, por exemplo, correndo o código de 3 até 1000 símbolos no alfabeto em vez de apenas até 100, mas, mesmo assim, foi possível ganharmos uma perceção dos valores que se pretendiam.

Em relação à complexidade computacional, poderia ser melhorada, algo que possivelmente a nossa experiência ao longo do tempo em programação nos ajudará a aperfeiçoar.

Para além dos valores que obtivemos, também teria sido interessante analisar a mediana, o tempo máximo, o tempo médio e o tempo mínimo que é necessário para descodificar a mensagem em função do número de símbolos do nosso alfabeto.

Anexos

```
346 static void recursive_decoder(int encoded_idx,int decoded_idx,int good_decoded_size)
347 {
348
349   int initial_encoded_idx = encoded_idx;
350   int decoded_pos = decoded_idx;
351   int correct_decoded_size = good_decoded_size;
352   int symb_size = 1;
353
354   _number_of_calls_++;
355
356   if (decoded_pos - correct_decoded_size > _max_extra_symbols_){
357     _max_extra_symbols_ = decoded_pos - correct_decoded_size;
358   }
359
360   // se (symb_size) > _c_->max_bits então não é possível descodificar mais
361   // verificar se a mensagem já acabou
362   while (symb_size <= _c_->max_bits && _encoded_message_[initial_encoded_idx + symb_size - 1] != 0){
363
364     for (int j = 0 ; j < _c_->n_symbols; j++){
365       // length de um símbolo
366       int i, h;
367       for (i = 0; _c_->data[j].codeword[i]; i++);
368
369       // o símbolo _c_->data[j].codeword é do tamanho do símbolo que estamos a considerar?
370       if (i == symb_size){
371         // as strings são do mesmo tamanho, por isso quando _c_->data[j].codeword[h] acabar já comparamos os dois símbolos
372         for (h = 0; _c_->data[j].codeword[h] && _encoded_message_[initial_encoded_idx + h] && _c_->data[j].codeword[h] == _encoded_message_[initial_encoded_idx + h]; h++);
373
374         // se o i e o h forem iguais quer dizer que os símbolos são iguais
375         if (i == h){
376           _decoded_message_[decoded_pos] = j;
377           int idx = initial_encoded_idx + symb_size;
378           // se o decoded_idx - good_decoded_size não for zero quer dizer que a mensagem já está errada,
379           // se não está verifica se está certa
380           if (decoded_pos - correct_decoded_size == 0 && j == _original_message_[decoded_pos]){
381             // o decoded_pos será sempre menor que o tamanho da mensagem original por ser um index
382             if (decoded_pos == _original_message_size_ - 1){
383               // printf("-------------------SOLUÇÃO ENCONTRADA--------------------\n");
384               _number_of_solutions_++;
385             }
386
387             recursive_decoder(idx, decoded_pos+1, correct_decoded_size+1);
388           }
389           // a mensagem não está certa, mas é possível continuar a descodificar
390           else{
391             recursive_decoder(idx, decoded_pos+1, correct_decoded_size);
392           }
393         }
394
395       }
396     }
397
398     symb_size++;
399
400   }
```

Figura 15: Função completa