Universidade de Aveiro

Algoritmos e Estruturas de Dados

Recursively decoding a non-instantaneous binary code

Professores:

Tomás Oliveira e Silva (tos@ua.pt)
Pedro Lavrador (plavrador@ua.pt)

Pedro Sobral, 98491 – 40%
André Freixo, 98495 – 30%
Marta Fradique, 98629 – 30%

08/02/2021

Índice

1 - Introdução

No âmbito da unidade curricular de Algorítmos e Estruturas de Dados, foi-nos proposto a realização deste trabalho prático, onde de um modo geral, espera-se que de uma forma recursiva se consiga descodificar um código binário fornecido, sendo que existe um e um só código correto, dessa forma a solução possível na descodificação de qualquer código é unica. O trabalho foi auxilido por um repositório GIT, onde todo o código fonte pode ser consultado neste link.

O algoritmo por nós a desenvolver, deverá consistir em fazer corresponder um código binário a um determinado símbolo. O nosso objetivo principal é receber uma mensagem e decodificar a mesma com os símbolos que esta compõe. Por exemplo, e de acordo com a figura abaixo(Fig. 1) ao símbolo um, o código correspondente é o "001". E caso recebemos uma mensagem por exemplo 10111001, a decodificação seria "231".

symb	codeword
0	0
1	001
2	101
3	11

1 - Exemplo de tabela com os símbolos e respetivo código binário

Para alcançar o resultado pretendido, o algoritmo deve analisar um *bit* de cada vez e prever a que *codeword* este pode pertencer, como podemos verificar no exemplo dado, pois ao analisar o primeiro *bit* "1" no caso, este pode pertencer ao *symbol "2" ou "3"*. De seguida, avançamos para o próximo bit que é um "0", concluímos então que o primeiro símbolo é o "2" adicionando este à resposta, aplicamos este racioínio sucessivamente até ser descodificada a mensagem toda.

Definimos também como objetivos: consolidar os conceitos aprendidos nas aulas teóricas, criar gráficos para analisarmos os resultados que alcançamos, criar uma função que descodifica a mensagem em tempo real e por último queremos ainda melhorar a forma como trabalhamos com a linguagem C e desenvolver um algoritmo que funcione de forma eficiente e optimizada.

2 - Compilação e Execução

Para compilar os programas, é vital ter um compilador de C instalado na máquina (p.e. gcc). O projeto foi desenvolvido com recurso ao IDE VSCode, para implementação do código C, e ao Matlab, para a realização de gráficos usados no relatório.

Para compilar o programa, é necessário utilizamos o *makefile*, da seguinte maneira:

```
make A03                    #compile without decode in real-time
make A03_RT                 #compile with decode in real-time
```

Para executar, há as seguintes possibilidades:

```
./A03 -s n_symbols seed                 # show the code words of random code
./A03 -t [n_symbols [message_size [seed]]]  # encode and decode a message
./A03 -x n_symbols                      # try the first 201 seeds
```

Para que conseguíssemos obter resultados mais rápido, fizemos uso do script *do_all.bash* fornecido na página online da unidade curricular. Para isso atribuímos permissões e executamos o *script:*

```
chmod u+x do_all.bash       #para atribuir permissões
./do_all.bash               #para executar o script
```

3 - Função recursive_decoder()

Para resolver o problema proposto, construímos a função recursive_decoder(), como argumentos passamos 3 variáveis:

- **encoded_idx**, índice do array _encoded_message_ do próximo bit a ser analisado;
- **decoded_idx**, índice do array _decoded_message_ onde irá ficar guardado o próximo símbolo a ser descodificado;
- **good_decoded_size**, número de símbolos descodificados corretamente.

Para construirmos a nossa função, iniciamos um contador _number_of_calls_ do número de vezes que a função recursive_decoder() é chamada, incrementamos em 1 o valor da variável _number_of_calls_.

De seguida, e para sabermos se devemos atualizar o valor da variável _max_extra_symbols_, criamos uma expressão condicional, de modo a que sempre que o decoded_idx menos o good_decoded_size for maior que o _max_extra_symbols_ atualizamos o _max_extra_symbols_ para o valor da diferença supra mencionada.

```
330  static void recursive_decoder(int encoded_idx, int decoded_idx, int good_decoded_size) {
331      _number_of_calls_++;  //increase by one, each time the function is called
332
333      if ((decoded_idx - good_decoded_size) > _max_extra_symbols_) {  //update the max_extra_symbols,
334          _max_extra_symbols_ = (decoded_idx - good_decoded_size);
335      }
```

*1 - Expressão condicional para update da variável _max_extra_symbols_*

Temos um condição de paragem, condição essa que se verifica sempre que o array _encoded_message_ está no fim, significando isso que a descodificação chegou ao fim, dentro da expressão condicional, atualizamos a variável _number_of_solutions_ para o valor 1, como só há uma solução em todas as possibilidades, esta terá então de tomar esse referido valor. Ainda executamos um return, para sair.

```
337     //* Terminal condition, it means that message is already decoded
338     if (_encoded_message_[encoded_idx] == '\0') {  //if the last inde
339         _number_of_solutions_++;                   //increase by one,
340
341         return;
```

*2 - Condição de paragem e atualização da variável _number_of_solutions_*

Chegamos agora à parte da função onde é feita a descodificação, a lógica para que a decodificação seja feita de modo certo é simples: percorrer o array _encoded_message_ enquanto se percorrem os codeword de cada símbolo, enquanto o bit do codeword for igual ao bit do _encoded_message num determinado índice, avançamos sendo que considera-se que um símbolo é descodificado quando o codeword chega ao fim.

Para pôr isto em prática, implementamos um ciclo for, que itera tantas vezes quantos símbolos houverem, sendo que este for tem associado a si a variável i, dentro do ciclo inicializamos a variável j a zero, e implementamos um novo ciclo, desta vez um ciclo while. Este ciclo, é iterado sempre que o bit do codeword j do símbolo i, é igual ao ao bit do array _decoded_message_ do índice encoded_idx + j.

Enquanto esta condição se verificar, dentro do while, é avaliada a condição para saber se o codeword chegou ao fim, pois caso se verifique que o codeword não tem mais bits, conseguimos descodificar um símbolo, no entanto é importante realçar neste momento que, descodificar um símbolo não significa que este tenha sido feito de forma correta, pois pode haver mais que uma possibilidade de descodificação para um determinado conjunto de bits de um codeword. A partir do momento que um símbolo é descodificado, adicionamos ao array _decoded_message_ no índice decoded_idx a variável i, sendo que o i é o símbolo que foi descodificado.

```
344     for (int i = 0; i < _c_->n_symbols; i++) {  //for cycle to go through
345         int j = 0;
346         while (_c_->data[i].codeword[j] == _encoded_message_[encoded_idx + j]) {
347             if (_c_->data[i].codeword[++j] == '\0') {
348                 _decoded_message_[decoded_idx] = i;
349                 //* confirm if the symbol decoded is equal to the symbol original,
```

3 - Início do ciclo for e do ciclo while

Nesta parte do algoritmo, temos 2 situações possíveis, o símbolo que foi descodificado foi bem descodificado, ou foi mal descodificado, quando falamos em ser “mal ou bem" descodificado referimo-nos a saber se estamos num ramo que nos irá levar a um deadend ou não, respetivamente. Para sabermos se o nosso símbolo nos levará a um deadend, fazemos 2 comparações dentro de uma expressão

condicional, condições essas que comparam o conteúdo do array _original_message_ e do array _decoded_message_ nos índices decoded_idx, sendo que o conteúdo destes arrays nesse determinado índice terá de ser igual, e o valor das variáveis good_decoded_size e decoded_idx serem também iguais.

Se estas 2 condições se verificarem, significa que o símbolo descodificado não nos levará a um deadlock, e neste caso chamamos a função recursive_decoder(), onde o encoded_idx, fica atualizado com o valor encoded_idx mais o j, o decoded_idx, com o valor de decoded_idx mais um, e o good_decoded_size mais um.

Caso as 2 condições referidas anteriormente não se verifiquem, então o símbolo levará-nos a um deadend, porém para obter as variávis de quantos lookaheads obtivemos, fazemos também esta opção chamando a função recursive_decoder(), onde o encoded_idx, fica atualizado com o valor encoded_idx mais o j, o decoded_idx, com o valor de decoded_idx mais um, e o good_decoded_size mantem o mesmo valor.

Neste bloco de código, temos ainda duas condições, que avaliam o valor da macro N, sendo verdadeiras, é feita a impressão de toda a descodificação em tempo real, que será explicado no próximo ponto do relatório.

```
                if (_original_message_[decoded_idx] == _decoded_message_[decoded_idx] && (good_decoded_size) == decoded_idx) {
                    if (N == 1) {
                        print_decode_real_time(decoded_idx);
                    }
                    recursive_decoder(encoded_idx + j, decoded_idx + 1, good_decoded_size + 1);
                } else {
                    if (N == 1) {
                        print_decode_real_time(decoded_idx);
                    }
                    recursive_decoder(encoded_idx + j, decoded_idx + 1, good_decoded_size);
                }
                break;
            }
        }
    }
}
```

4 - Expressões condicionais

4 - Decode em real-time

Relativamente à parte opcional, pela nossa interpretação, concluímos que o que é pedido é que seja visivel ao utilizador ver a descodificação em tempo real a ser feita, desse modo implementamos uma pequena função, a função print_decode_real_time(Fig.6), esta função aceita como argumento o decoded_idx.

```
321  static void print_decode_real_time(int decoded_idx) {
322      usleep(100000);
323      printf("\r");
324      for (int k = 0; k <= decoded_idx; k++) {
325          printf("%d", _decoded_message_[k]);
326      }
327      fflush(stdout);
328  }
```

5 - Função print_decode_real_time()

Dentro da função executamos um usleep(), com o valor 100000, para assim atrasar um pouco o processo de modo a que seja mais fácil a visualização e compreensão do que está a ser impresso. Implementamos um ciclo for, para printar os novos simbolos que vão sendo adicionados ao array _decoded_message_, ao fim do ciclo usamos a função fflush, com o argumento stdout. A execução desta função pode ser vista no neste link[1], exemplificando deste modo muito bem o funcionamento da mesma, para valores escolhidos arbitrariamente.

[1] - por motivos de guardar o relatório em.pfd, não nos é possível emplementar o gif nesse mesmo formato

De forma a que esta opção de visualização da descodificação não seja feita sempre que o programa principal fosse executado, pensamos numa maneira simples de a implementar. Definimos a macro N, no início do programa, e na nossa função recursiva temos uma condição que é validada sempre que a macro está definida com o valor 1. Para definirmos o valor da macro, de modo a querermos ou não usar a descodificação em tempo real, fizemos umas pequenas alterações no makefile do programa, adicionamos mais uma opção A03_RT, onde a macro fica definida com o valor 1 quando se compila com essa opção, caso se faça a compilação "normal", a macro fica com o valor 0, e nesse sentido não é feito qualquer tipo de descodificação em tempo real.

```
36  #ifndef N
37  #define N 0
38  #endif
```

7 – Macro N

```
1  clean:
2          rm -f a.out A03 *.pdf
3
4  A03:    A03.c rng.c
5          cc -Wall -O2 A03.c -o A03
6
7  A03_RT: A03 rng.c
8          cc -DN=1 -Wall -O2 A03.c -o A03
```

8 - Makefile

5 - Outras alterações no código

De maneira a guardar o tempo de execução da opção "-x" para um determinado número de símbolos, fizemos algumas pequenas alterações no código, tais como: fazer um "#include" do ficheiro elapsed_time.h que nos dará acesso à função cpu_time() e foi fornecida pelos Professores no primeiro trabalho prático ,adicionamos a variável cpu_time, e criamos a macro _cpu_time_ que corresponde à variável decoder_global_data.cpu_time, e na parte do código que em que tratamos da opção "-x", fizemos uso da função cpu_time(), de modo a calcular o delta de tempo relativamente à execução do programa. Na printf onde sãp impressas os resultados da opção "-x" adicionamos também o tempo de execução, como pode ser visto na figura seguinte.

```
490         _cpu_time_ = cpu_time();
491         for (seed = 1; seed <= N_MEASUREMENTS; seed++) {
492             srandom(seed);
493             c = new_code(n_symbols);
494             try_it(c, MAX_MESSAGE_SIZE, 0);
495             free_code(c);
496             t = (double)_number_of_calls_ / (double)MAX_MESSAGE_SIZE;
497             u = _max_extra_symbols_;
498             if (seed == 1 || t < t_min)
499                 t_min = t;
500             if (seed == 1 || t > t_max)
501                 t_max = t;
502             if (seed == 1 || u < u_min)
503                 u_min = u;
504             if (seed == 1 || u > u_max)
505                 u_max = u;
506             for (i = seed - 1; i > 0 && t_data[i - 1] > t; i--)  // i
507                 t_data[i] = t_data[i - 1];
508             t_data[i] = t;
509             for (i = seed - 1; i > 0 && u_data[i - 1] > u; i--)  // i
510                 u_data[i] = u_data[i - 1];
511             u_data[i] = u;
512         }
513         t_avg = u_avg = 0.0;
514         for (i = N_OUTLIERS; i < N_MEASUREMENTS - N_OUTLIERS; i++) {
515             t_avg += t_data[i];
516             u_avg += (double)u_data[i];
517         }
518         t_avg /= (double)(2 * N_VALID + 1);
519         u_avg /= (double)(2 * N_VALID + 1);
520         _cpu_time_ = cpu_time() - _cpu_time_;
```

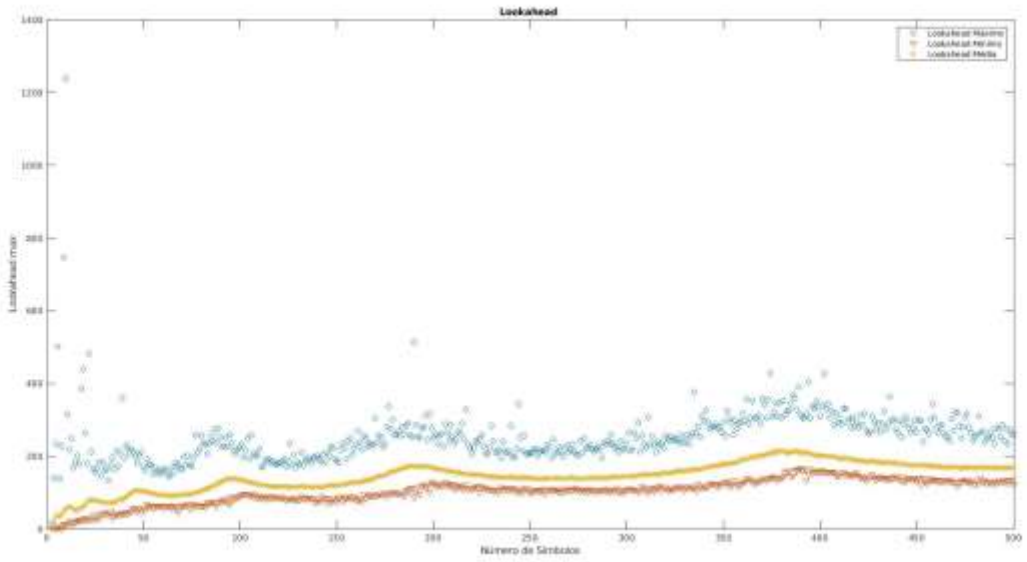
9 – Alterações ao código

6 - Resultados

Para obtermos os resultados de uma maneira mais rápida, falando em termos de tempo total de execução, utilizamos o script do_all.bash, que nos permite executar vários programas em paralelo.

Após resolvermos o problema fizemos gráficos com os dados que consideramos mais pertinentes. Para isso, usamos o comando *cat 0??? > output.txt*,e dessa forma guardamos num único ficheiro todos os dados resultantes da execuçao do programa, que pode ser consultado aqui. Os gráficos foram feitos em MatLab, visto que é uma ferramenta já por nós conhecida.

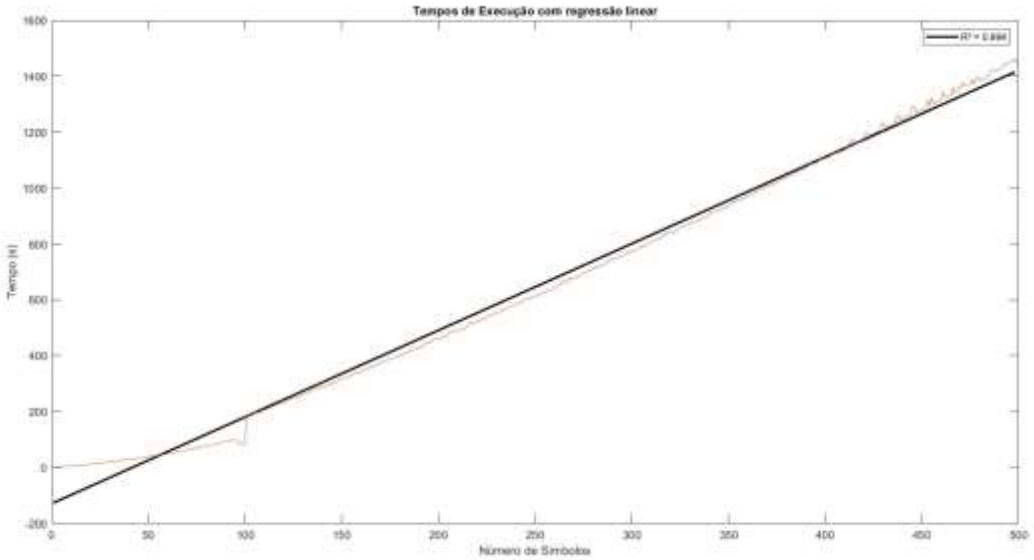
O gráfico onde está representada a comparação entre entre o Número de Símbolos e Lookaheads é o seguinte(Fig. 10):



10 - Comparação entre entre o Número de Símbolos e Lookaheads

Quanto a este gráfico podemos tirar algumas conclusões, a primeira e talvez a que salta mais à vista é a sua forma ondulada, que tem picos em valores aproximados do dobro do pico anterior, por exemplo, há um pico sensivelmente para n = 50, depois para n=100, n=200, n=400, tentamos ainda perceber o porque desta ondulação, porém não conseguimos encontrar nenhuma explicação lógica e/ou matemática para tal acontecimento. Outra conclusão plausivel de se tirar é que quanto maior for o n, menor é a probabilidade do outlier ser grande/afastado do normal, como podemos ver até valores de n = 50, os Lookahead máximos são bastande desfazados da suposta linha “normal”, que seria o Lookahead Médio. À medida que se percorre o eixo do x, mais se percebe que os 3 valores do Lookahead, segue uma forma ondulada muito semelhante.

O gráfico que representa os Tempos de execução vs Números Símbolos é o seguinte(Fig.11):



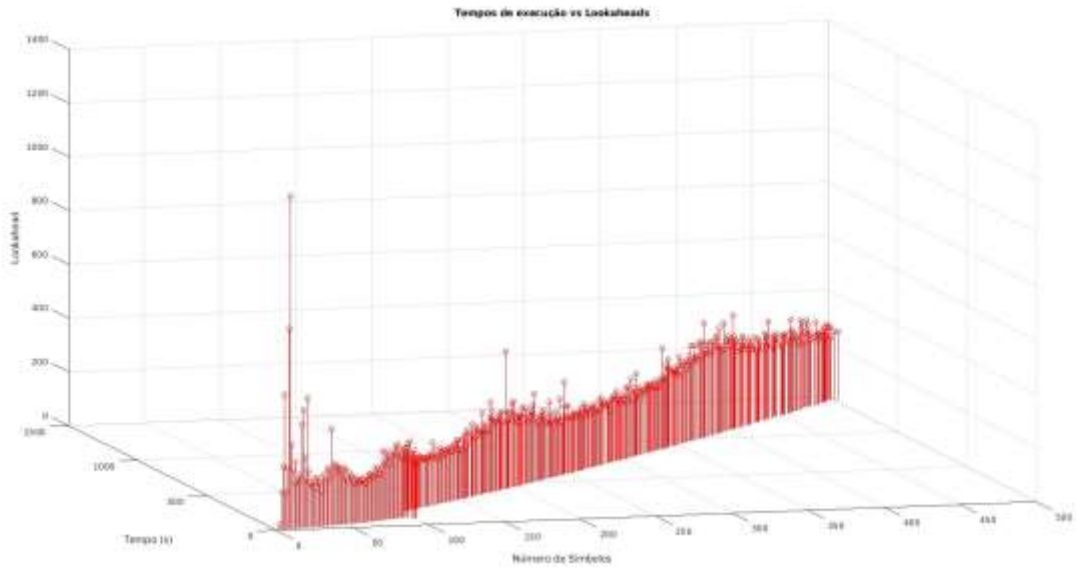
11 - Tempos de execução vs Números Símbolos

Relativamente ao gráfico dos Tempos de Execução, este cresce de forma linear, no entanto, apresenta uma pequena distorção na zona do cem no eixo do x, isto é explicado por terem sido obtidos resultados para o valor de n até cem, sendo que esses resultados foram obtidos em quatro terminais, os restantes valores até n igual a quinhentos, foram obtidos mais tarde, sendo que foram obtidos em oito terminais. Como o número de executáveis a correr em paralelo era considerável, nota-se um ligeiro aumento do tempo unitário de cada resultado, porém no tempo total de execução de n=3 até n=500 os ganhos temporais são extraordinários.

O tempo total até cem, foram uns consideráveis dezanove minutos com quatro terminais ativos, e o tempo total de execução desde n=3 até n=500 foram sensivelmente dez horas, com oito terminais ativos.

Implementamos ainda uma regressão linear no gráfico, sendo que obtivemos um R2 = 0.994, sendo portanto um valor bastante satisfatório, e conclusivo à questão de justificação do crescimento linear dos tempos de execução.

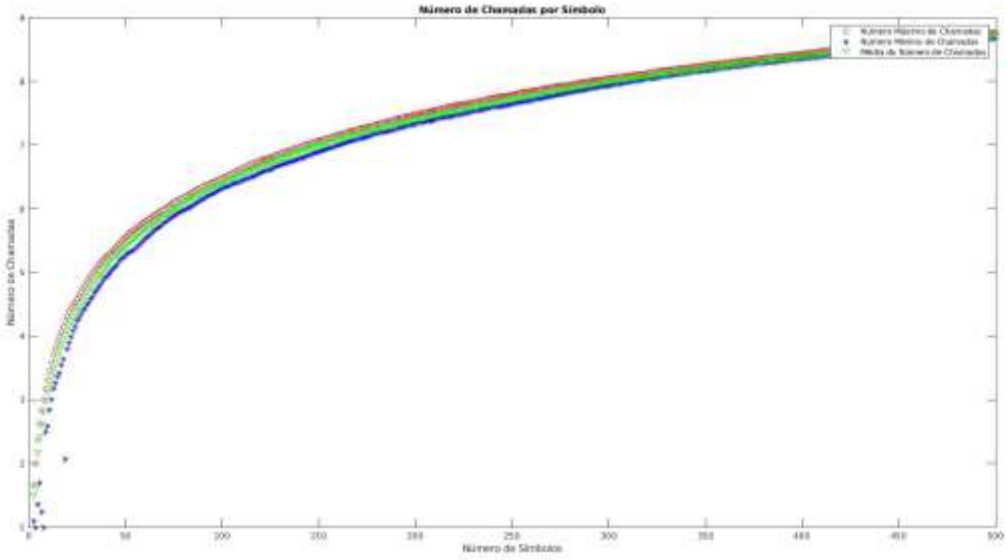
O gráfico 3D que representa os Tempos de execução vs Lookaheads é o seguinte(Fig.12):



12 - Tempos de execução vs Lookaheads

Decidimos ainda fazer um gráfico com 3 eixos, de modo a que fosse possivel visualizar de forma fácil, a interligação entre as variáveis, Número de Símbolos, Tempo, Lookahead. Neste gráfico não há muito mais a retirar que já não se tenha concluido, no entanto demonstra de maneira eficiente o comportamento destas variáveis. Os Lookaheads, apresentam uma trajetória ondulatória, e o tempo de execução mantem os aspeto linear à medida que o n aumenta, mantém a nossa incerteza para justificar a ondulação já referida, sendo que conseguimos observar uma sequencia de quando os picos das “onde” acontecem, em valores de n aproximados: 25, 100, 200, 400.

O gráfico que representa o *Número de Chamadas por Símbolo*, é o seguinte(Fig.13)



13 - Número de Chamadas por Símbolo

Ao analisarmos o gráfico do *Número de Chamadas por Símbolo*, podemos concluir que este apresenta aproximadamente um crescimento de ordem ***log n***, visto que, inicialmente para um pequeno *Número de Símbolos* , o *Número de Chamadas* aumenta de forma muito acentuada. À medida que o valor do *Número de Símbolos* sobe o *Número de Chamadas* continua a crescer mas não de forma tão saliente. Uma razão no nosso entender plausivel para o rápido crescimento para valores de *n* pequenos, é sem duvida a baixa complexidade da mensagem em si, comparada com messagens para valores maiores de *n*, neste caso para valores até 500.

7 - Apêndice

Overview completo das 2 funções por nós implementadas:

14- Função recursive_decoder

```c
static void print_decode_real_time(int decoded_idx) {
    usleep(100000);
    printf("\r");
    for (int k = 0; k <= decoded_idx; k++) {
        printf("%d", _decoded_message_[k]);
    }
    fflush(stdout);
}
```

15 - Função print_decode_real_time

Código MatLab, implementado:

```
file = load("output.txt");

n = file(:, 1);

time = file(:, 10);

lookaheadMAX = file(:, 9);
lookaheadMIN = file(:, 6);
lookaheadAVG = file(:, 7);

callsSymbolMAX = file(:, 5);
callsSymbolMIN = file(:, 2);
callsSymbolAVG = file(:, 3);

figure(1)
x = (1:length(n))';
Rtime = fitlm(x, time);
pptime = polyfit(x, time, 1);
plot(x, polyval(pptime, x), '-k', "lineWidth", 2);
hold on;
plot(n, time);
hold off;
title 'Tempos de Execução com regressão linear'
xlabel 'Número de Símbolos'
ylabel 'Tempo (s)'
legend ('R² = 0.994')

figure(2)
plot(n, lookaheadMAX, "o");
hold on;
plot(n, lookaheadMIN, "v");
plot(n, lookaheadAVG, "*");
hold off;
title 'Lookahead '
xlabel 'NÃºmero de Símbolos'
ylabel 'Lookahead max'
legend ("Lookahead MÃ¡ximo","Lookahead MÃnimo","Lookahead MÃ©dia");

figure(3)
plot(n, callsSymbolMAX, "or");
hold on;
plot(n, callsSymbolMIN, "*b");
plot(n, callsSymbolAVG, "vg");
title 'NÃºmero de Chamadas por SÃmbolo'
xlabel 'NÃºmero de SÃmbolos'
ylabel 'NÃºmero de Chamadas'
legend ("NÃºmero MÃ¡ximo de Chamadas","NÃºmero MÃnimo de Chamadas","MÃ©dia do NÃºmero de Chamadas");

figure(4)
stem3(n, time, lookaheadMAX, "or");
title 'Tempos de execuÃ§Ã£o vs Lookaheads'
xlabel 'NÃºmero de SÃmbolos'
```

16 - Código MatLab implementado

8 - Conclusão

Com a realização deste trabalho conseguimos criar um script que descodifica recursivamente um código binário não instantâneo. Recorremos ainda à criação de vários gráficos para analisar variáveis como o *Tempo de Execução*, o *Número de Chamadas por Símbolo*, os *Lookaheads* e o *Número de símbolos* e como estas variam em consonância.

Consolidamos também os conceitos aprendidos nas aulas teóricas, criamos uma função que descodifica a mensagem em tempo real e por último melhoramos a forma como trabalhamos com a linguagem C e desenvolvemos um algoritmo que funciona de forma eficiente e optimizada.

De forma geral cumprimos com os objetivos previamente definidos e complementamos os conceitos teóricos com os práticos.

9 - Bibliografia

Silva, Tomás Oliveira e. Lecture notes: Algorithms and Data Structute (AED -Algoritmos e Estruturas de Dados), LEI, MIEC, 2020/2021