Algoritmo e Estrutura de Dados

Licenciatura em Engenharia Informática

Recursively Decoding a Non-instantaneous Binary Code

Professores:

Tomás Oliveira e Silva

Pedro Lavrador.

Data: 08/02/2021

1. Trabalho realizado por:

Daniel Figueiredo, nº 98498, 33,33%

Eva Bartolomeu, nº 98513, 33,33%

Eduardo Fernandes, nº 98512, 33,33%

Índice

1. Introdução

No âmbito da cadeira de Algoritmos e Estruturas de Dados, foi-nos proposto um trabalho acerca de Recursively Decoding a Non-instantaneous Binary Code.

Este problema baseia-se num conjunto de símbolos, em que cada símbolo pode ser codificado pelo seu codeword (um conjunto de bits). Uma mensagem descodificada é constituída por símbolos, e uma mensagem codificada é composta por bits.

Perante estes dados, é nos pedido a descodificação de uma mensagem codificada. Além disso, propõe-se o cálculo do número médio de chamadas da função por símbolo na mensagem, e o número máximo de símbolos descodificados incorretamente.

Os objetivos deste relatório são:

- Desenvolver um código C eficiente capaz de resolver o problema proposto;
- Obter bons valores para os números de símbolos regredidos;
- Elaborar um gráfico do número mínimo, médio e máximo de chamadas por símbolo da mensagem;
- Construir um gráfico que mostra o Lookahead symbols;
- Interpretar gráficos.

2. Método de resolução

A nossa função *recursive_decoder(),* é chamada com três argumentos sendo eles: *encoded_idx, decoded_idx* e *good_decoded_size.* O argumento *encoded_idx* refere-se à posição atual na mensagem codificada, o *decoded_idx*, por sua vez, refere-se à posição atual na mensagem descodificada (número de símbolos descodificados) e o *good_decoded_size* refere-se a quantos símbolos, até ao momento já foram descodificados corretamente.

Para a construção da nossa função, decidimos iniciar um contador *_number_of_calls_*++, que nos vai permitir observar quantas vezes a função *recursive_decoder*() vai ser chamada e sempre que esta for chamada incrementamos o valor 1 ao valor de *_number_of_calls_*.

De seguida, fomos calcular a variável *_max_extra_symbols_*, que se refere ao número de símbolos máximo a mais que a função foi calculando até se aperceber que estava a seguir um caminho errado e não iria encontrar a solução por esse caminho. Fizemos uma expressão condicional, de modo a que sempre que (*decoded_idx - good_decoded_size*) for maior que *_max_extra_symbols_* vamos atualizar o valor de *_max_extra_symbols_* para (*decoded_idx - good_decoded_size*).

A seguir a esta expressão condicional, temos outra expressão condicional que nos permite saber se já percorremos a mensagem codificada toda, verificando se na posição *encoded_idx* do *array _encoded_message_* encontramos o operador terminal '\0' e caso esta condição se verificar, atualizamos a variável *_number_of_solutions_* com um incremente de 1, havendo apenas uma solução.

```
static void recursive_decoder(int encoded_idx, int decoded_idx, int good_decoded_size)
{
  _number_of_calls_++;

  if ((decoded_idx - good_decoded_size) > _max_extra_symbols_)
  {
    _max_extra_symbols_ = (decoded_idx - good_decoded_size);
  }

  if (_encoded_message_[encoded_idx] == '\0')
  {
    _number_of_solutions_++;

  }
```

Figura 1 - Parte inicial da função *recursive_decoder()*

De seguida, passamos à parte da descodificação da mensagem codificada, em que o método utilizado, foi o facto de percorrermos todos os símbolos e verificar se o *codeword* de cada um destes símbolos era igual a *_encoded_message_* num determinado índice, verificando se o símbolo foi bem descodificado caso o *codeword* do símbolo que estava a ser avaliado chegar ao fim (‘\0’).

Deste modo, implementamos um ciclo *for*, que percorre *_c_->symbols,* percorrendo assim todos os símbolos existentes, ficando os símbolos com o valor da iteração do ciclo *for*. Dentro deste *for*, foi implementado outro ciclo *for*, que nos permite percorrer os *codeword* do símbolo que estamos a iterar. Sendo que, este *for* termina caso não haja correspondência com o *bit* da mensagem codificada, *_encoded_message_* de índice *_encoded_idx_ + j*, com o *bit* correspondente no *codeword* do símbolo. Caso esta condição no ciclo *for* seja verificada, então dentro do *for,* encontramos uma expressão condicional que vai verificar se o *bit* que iremos encontrar na iteração seguinte (*_c_->data[i].codeword[j+1]*) é o *bit* que contém o terminador ‘\0’, e caso se verifique essa condição, vamos colocar, na mensagem descodificada, *_decoded_message_* no índice *decoded_idx,* o símbolo que estávamos a iterar sobre.

Depois de colocar o símbolo na mensagem descodificada, iremos observar que podemos ter duas situações possíveis, sendo elas o facto de o caminho que estamos a percorrer nos leva para a solução certa ou nos leva para a solução errada. Para descobrirmos se o caminho escolhida será a solução certa ou não, fizemos uma expressão condicional que verifica se o conteúdo de *_original_message_* e de *_decoded_message,* no índice *decoded_idx,* são iguais ou não, fazendo também um *and* visto que *good_decoded_size* e *decoded_idx* têm de ser iguais. Caso estas condições se verifiquem, então podemos concluir que estamos a ir no caminho certo, até ao momento, e vamos chamar a função *recursive_decoder(),* com os argumentos *encoded_idx + j + 1,* visto que, até aquele momento verificamos que a descodificação estava a ser bem feita, mais o argumento *decoded_idx + 1,* pois vamos avançar para outro índice na mensagem descodificada e também o argumento *good_decoded_size + 1,* caso, posteriormente, tenhamos de voltar atrás no caminho seguido. Caso as condições não se verifiquem, então, vamos chamar a função recursiva com os mesmo argumentos, à exceção do argumento *good_decoded_size* em que não adicionamos o valor 1, visto que, o caminho escolhido foi o errado.

```
for (int i = 0; i < _c_->n_symbols; i++)
{
  for (int j = 0; _c_->data[i].codeword[j] == _encoded_message_[encoded_idx + j]; j++)
  {
    if (_c_->data[i].codeword[j+1] == '\0')
    {
      _decoded_message_[decoded_idx] = i;
      if (_original_message_[decoded_idx] == _decoded_message_[decoded_idx] && good_decoded_size == decoded_idx)
      {
        recursive_decoder(encoded_idx + j+1, decoded_idx + 1, good_decoded_size + 1);
      }
      else
      {
        recursive_decoder(encoded_idx + j+1, decoded_idx + 1, good_decoded_size);
      }

      break;
    }

  }
 }
}
```

Figura 2 - Chamada da função recursivamente.

3. Resultados

Antes de executar o script fornecido pelo professor, “do_all.bash” (Pág. 10, Fig. 5), por uma questão de organização, decidimos acrescentar no script uma criação de uma pasta “results”, colocando lá dentro os ficheiros todos criados neste script.

De forma a obter futuros gráficos, criamos um script “results.bash” (Pág. 10, Fig. 6) que gera um ficheiro de nome “results.txt”, contendo todas as linhas dos ficheiros da pasta “results”.

3.1. Gráfico do número de chamadas por símbolo da mensagem

Para a apresentação do gráfico do número de chamadas por símbolo da mensagem, usamos no gnuplot os comandos retratados na Figura 3.

```
eva@eva-TUF-GAMING-FX504GD-FX80GD:~/Documentos/AED/A03$ gnuplot

        G N U P L O T
        Version 5.2 patchlevel 2    last modified 2017-11-01

        Copyright (C) 1986-1993, 1998, 2004, 2007-2017
        Thomas Williams, Colin Kelley and many others

        gnuplot home:     http://www.gnuplot.info
        faq, bugs, etc:   type "help FAQ"
        immediate help:   type "help"  (plot window: hit 'h')

Terminal type is now 'qt'
gnuplot> data_file="results.txt"
gnuplot> set title "Number of calls per message symbol"
gnuplot> set logscale xy
gnuplot> set grid
gnuplot> set xlabel "Number of symbols"
gnuplot> plot data_file u 1:5 w p t 'number of calls per message symbol (max)' ,\
>     data_file u 1:3 w p t 'number of calls per message symbol (avg)' ,\
>     data_file u 1:2 w p t 'number of calls per message symbol (min)'
```

Figura 3 - Comandos utilizados no gnuplot, para construir o gráfico do número de chamadas por símbolo da mensagem.

Obtendo assim o Gráfico 1, em que o eixo das abscissas é o número de símbolos e o das ordenadas representa o número de chamadas por símbolo da mensagem. O símbolo '+' de cor roxa representa o número máximo de chamadas por símbolo, o símbolo 'x' de cor verde simboliza o número médio de chamadas por símbolo, e por fim, o '*' de cor azul retrata o número mínimo de chamadas por símbolo.

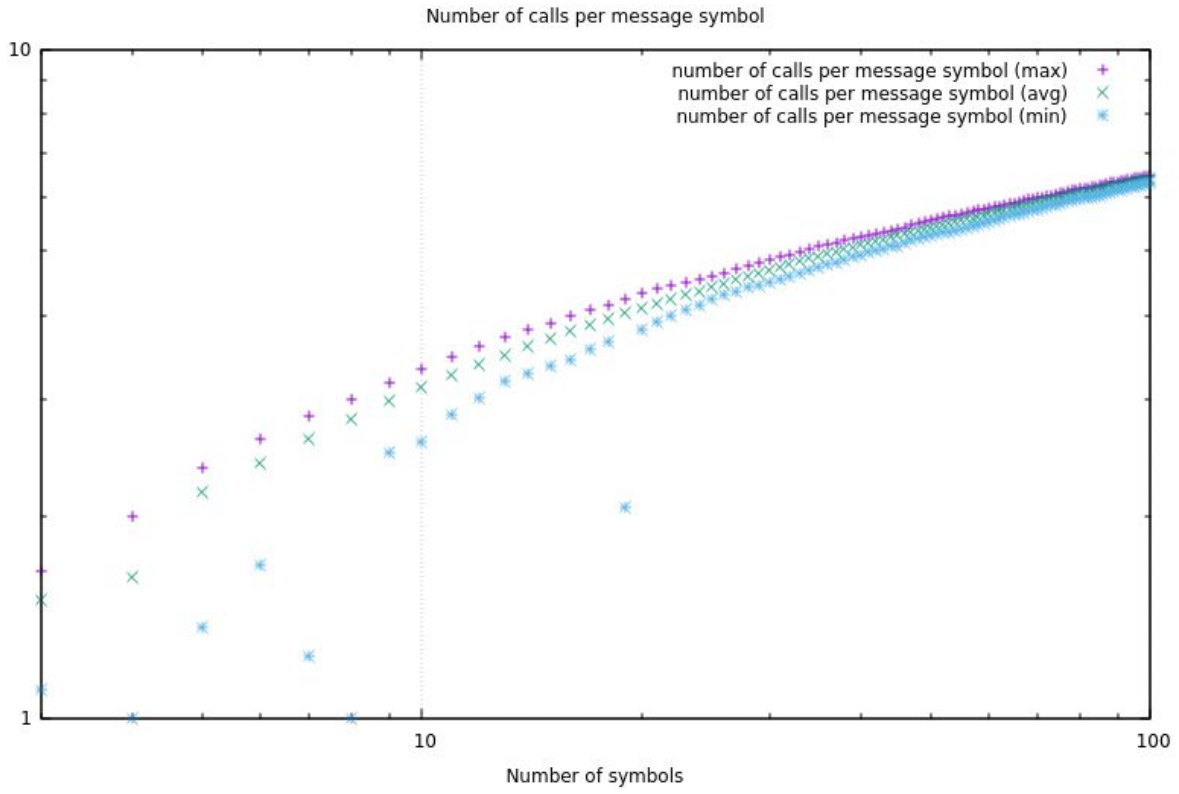

Gráfico 1: Número de chamadas por símbolo da mensagem.

Neste gráfico é visível que quanto maior for o número de símbolos, maior será o número médio, mínimo e máximo de chamadas por símbolo. Este aumento é cada vez menos significativo, à medida que o número de símbolos aumenta.

Portanto, com o crescimento do número de símbolos, o número médio de chamadas por símbolo torna-se cada vez mais preciso.

3.2. Gráfico do Lookahead symbols

Para a apresentação do gráfico do número de símbolos regredidos após se ter verificado que aquela forma de decodificação era errada, usamos no gnuplot os comandos retratados na Figura 4.

```
eva@eva-TUF-GAMING-FX504GD-FX80GD:~/Documentos/AED/A03$ gnuplot

        G N U P L O T
        Version 5.2 patchlevel 2    last modified 2017-11-01

        Copyright (C) 1986-1993, 1998, 2004, 2007-2017
        Thomas Williams, Colin Kelley and many others

        gnuplot home:     http://www.gnuplot.info
        faq, bugs, etc:   type "help FAQ"
        immediate help:   type "help"  (plot window: hit 'h')

Terminal type is now 'qt'
gnuplot> data_file="results.txt"
gnuplot> set title "Lookahead symbols"
gnuplot> set logscale xy
gnuplot> set grid
gnuplot> set xlabel "Number of symbols"
gnuplot> plot data_file u 1:9 w p t 'lookahead symbols (max)' ,\
>data_file u 1:7 w p t 'lookahead symbols (avg)' ,\
>data_file u 1:6 w p t 'lookahead symbols (min)'
```

Figura 4 - Comandos utilizados no gnuplot, para construir o gráfico com o número de símbolos que foram codificados numa alternativa errada.

Consequentemente dos comandos executados, obtivemos o Gráfico 2 que possui no eixo das abcissas o número de símbolos como também, possui no eixo das ordenadas o número de símbolos que foram descodificados mas, que no entanto, se encontravam numa alternativa errada de descodificação. O símbolo '+' que possui a cor roxa indica o valor máximo de símbolos descodificados num caminho de descodificação errada, o símbolo 'x' de cor verde representa o valor médio deste mesmo fator bem como o símbolo '*' de cor azul, que simboliza o respetivo valor mínimo.

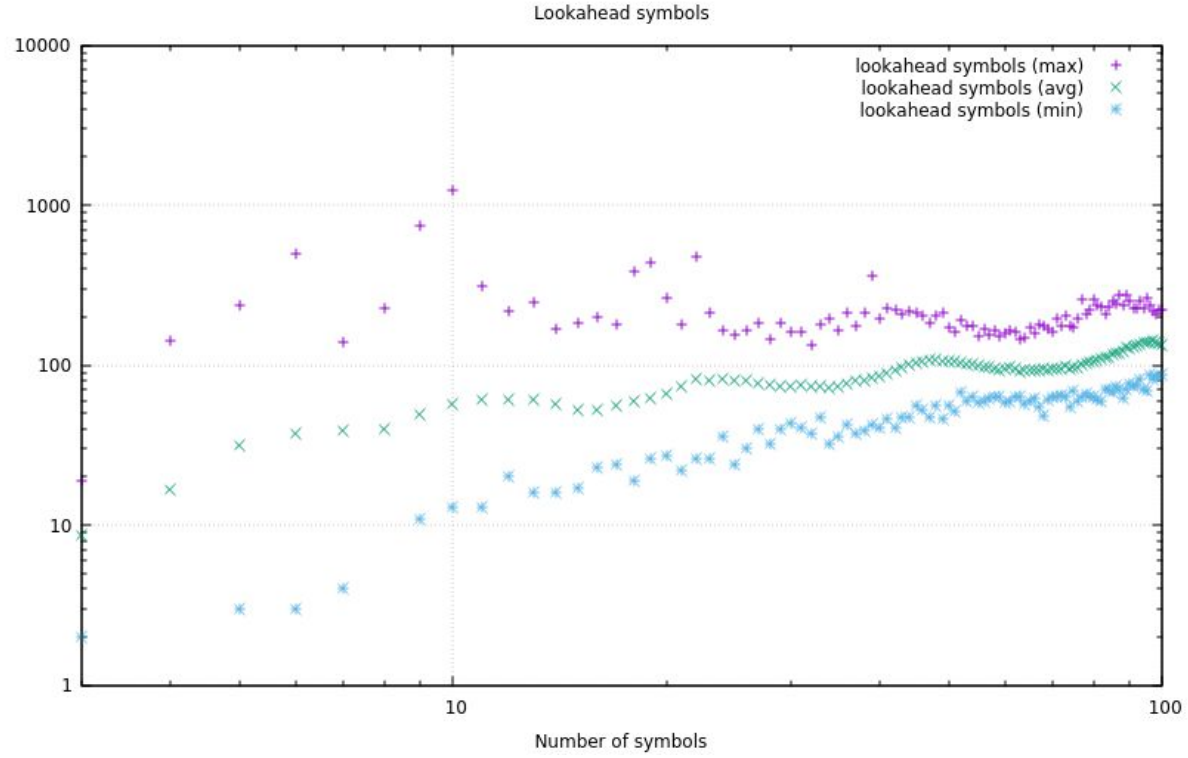

Gráfico 2: Número de símbolos descodificados em uma alternativa de descodificação errada.

Com a projeção gráfica dos valores resultantes do programa, é evidente que quanto maior o número de símbolos é, o valor de símbolos descodificados em um caminho errado também o será. Do mesmo modo, à medida que se verifica o aumento destes valores, é possível verificar que a discrepância entre os máximos, médios e mínimos valores de símbolos descodificados, é cada vez menor. Verifica-se que o intervalo de valores é cada vez mais pequeno e cada vez há menos variações dos mesmos.

Este valor de símbolos varia bastante consoante o aumento do número de símbolos, sendo que há um momento em que este valor atinge um máximo equivalente a *1239* símbolos descodificados para *n=10*, e um valor mínimo de *0* para *n=4*.

4. Apêndice

```
#! /bin/bash

mkdir results

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

Figura 5: Código bash de do_all.bash.

```
cat results/* > results.txt
```

Figura 6: Código bash de results.bash.

5. Conclusão

Com este trabalho podemos concluir que **fortalecemos os nossos conhecimentos acerca da recursividade em uma função bem como a descodificação de um código binário não instantâneo** que fizeram parte deste trabalho prático.

Além disso, conseguimos melhorar o nosso conhecimento em linguagem C embora, já não seja tão grande esse aprimoramento devido ao facto de trabalharmos com esta linguagem há algum tempo.

Na nossa opinião, os objetivos do trabalho prático foram atingidos com sucesso. **Obtivemos bons valores para o número de símbolos regredidos e foram construídos e interpretados gráficos com informação diretamente relacionados com o trabalho.** Estes gráficos possuem informação crucial para o entendimento do código escrito ajudando na interpretação dos valores obtidos e como estes variam em função das variáveis independentes.