ALGORITMOS E ESTRUTURAS DE DADOS

RECURSIVE MESSAGE DECODING

ALEXANDRE SERRAS (97505) (1/3)

JOÃO REIS (98474) (1/3)

RICARDO RODRIGUEZ (98388) (1/3)

Turma P1

Prof. Tomás Oliveira e Silva

2020/2021

Índice

Introdução

Para a realização deste trabalho foi-nos proposto descodificar uma mensagem encriptada apresentada em código binário não-instantâneo.

Cada símbolo descodificado do alfabeto pode ser representado em binário por uma sequência específica de bits á qual designamos de “binary codeword”. Esta lista de símbolos, bem como a sua respetiva codificação, pode ser visualizada ao executar o programa ./A03 -s seguido de dois parâmetros: o número de símbolos desejados e a seed que queremos utilizar.

Desta forma, o objetivo principal é desenvolver uma função recursiva “recursive_decoder” em linguagem C que consegue traduzir a mensagem codificada, retornando o número de soluções possíveis para o mesmo problema, bem como o maior número de bits mal codificados em sequência.

Para executar o programa principal, escrevemos ./A03 -t seguido de três argumentos: o número de símbolos, o tamanho da mensagem descodificada original e, por último, a seed.

Explicação do código

```
348  static void recursive_decoder(int encoded_idx,int decoded_idx,int good_decoded_size)
349  {
350    int controlo=0;
351    int contador=0;
352    int verifica=0;
353    int i;
354    for ( i= 0; i< _original_message_size_;i++ ){
355      if (i < decoded_idx ){
356        if(_decoded_message_[i] != _original_message_[i] ){
357            controlo=1;
358            verifica=i;
359            break;
360        }
361        else{
362          verifica=i;
363        }
364      }
365    }
366    contador=decoded_idx-verifica;
367    if(controlo == 0 && decoded_idx == _original_message_size_ ){
368      _number_of_solutions_++;
369      return ;
370    }
371    int  ndeco=decoded_idx,nencode_idx=encoded_idx, senencode_idx=encoded_idx;
372    int teste=0;
373    int bitLidos=1,k=0;
374    char bits[_c_->max_bits + 1];
```

```
375    while (bitLidos <= _c_->max_bits && _encoded_message_[nencode_idx] != 0 ){
376      bits[k]=_encoded_message_[nencode_idx];
377      bits[k+1]='\0';
378      for(int p = 0;p < _c_->n_symbols;p++){
379        teste=strcmp(bits,_c_->data[p].codeword);
380          if (teste== 0) {
381              _decoded_message_[ndeco]=p;
382              _number_of_calls_++;
383              recursive_decoder(senencode_idx+bitLidos,ndeco+1, good_decoded_size);
384          }
385      }
386      bitLidos++;
387      k++;
388      nencode_idx++;
389    }
390    if (contador > _max_extra_symbols_ ){
391      _max_extra_symbols_=contador;
392    }
393     return ;
394  }
```

Inicialmente, para resolver o problema, criaram-se 3 variáveis às quais chamamos de controlo, verifica e contador. A primeira variável controlo vai basicamente servir de controlo para a resolução do problema e as últimas 2 vão ser utilizadas para verificarmos quantos símbolos foram descodificados de forma errada e, no final, caso o valor desta seja superior à variável max_extra_symbols, este valor é atualizado.

De seguida, vamos percorrer a mensagem original, já sabendo quantos símbolos foram descodificados utilizando a variável global decoded_index. Caso verifiquemos que temos um valor diferente entre a mensagem descodificada e a mensagem original, a variável de controlo passa automaticamente para 1, o verifica fica com o valor da posição atual, ou seja, a variável i e, uma vez que a mensagem original difere da mensagem descodificada, acabamos o ciclo for com um break.

No entanto, caso o valor de verifica seja diferente do valor de decoded_index, significa que temos, até agora, todos os símbolos bem descodificados, atualizando sempre a variável verifica com o valor atual de i, passando para a próxima iteração.

Quando finalmente ultrapassarmos a estrutura for loop, vamos querer guardar na variável contador a diferença entre o decoded_idx e a variável verifica. Este valor representa a contagem de símbolos consecutivos mal descodificados e será utilizada posteriormente no código.

Depois, caso a variável de controlo seja igual a 0, ou seja, se não houve nenhum símbolo mal descodificado e o tamanho mensagem original seja igual ao index atual da mensagem descodificada, isto quer dizer que encontrámos uma solução face ao problema designado e incrementamos uma unidade ao valor de _number_of_solutions_, invocando um return para sair da função.

Seguidamente, inicializamos e declaramos uma variável ndeco que vai armazenar o valor atual de decoded_idx e outras duas nencode_idx e senencode_idx que vão guardar o valor de encoded_idx. Igualmente, inicializamos e declaramos a variável teste e k com valor nulo e outra variável bitLidos com valor 1 que, tal como o próprio nome indica, vai guardar o nome de bits que acabaram de ser descodificados da mensagem encodificada. Por último, como precisamos de guardar os bits lidos para formar um símbolo, criamos uma nova variável bits, um array de caracteres com tamanho máximo igual ao número máximo possível para codificar um símbolo (max_bits) mais um.

Posteriormente, enquanto que o número de bits lidos (bitLidos) não ultrapassar o número máximo de bits que codificam um símbolo e a mensagem codificada na posição nencode_idx for diferente de zero, pois verificamos que a mensagem codificada é um array com os códigos ASCII de 0 e 1 logo 48 e 49 e que depois de acabar a mensagem o array continua mas com vários 0's,logo quando encontramos um 0 sabemos que chegamos ao final, executamos uma série de instruções.

Estas são guardar na posição k do array de caracteres bits o bit que está a ser lido da mensagem codificada, inserindo na posição seguinte o caractere especial '\0' que marca o final da lista. Depois, percorremos todos os símbolos possíveis e verificamos, através do uso da função strcmp, se a sequência de bits lidos atual (armazenado em bits) corresponde ao "binary codeword" do símbolo que está a ser verificado. Caso sejam iguais, a variável teste terá o valor 0 e o próximo if será executado.

Uma vez que conseguimos codificar um símbolo, colocaremos na posição ndeco do array de _decoded_message_ o novo símbolo decifrado tal como pretendemos e incrementamos a variável _number_of_calls_. Como identificámos um novo símbolo a partir da mensagem codificada e queremos agora decifrar os restantes bits da mesma, chamamos a mesma função recursiva com os parâmetros de entrada atualizados: o novo encoded_idx será a soma da posição em que começamos a verificar os bits da mensagem codificada pelo número de bits lidos, o decoded_idx terá o seu valor

incrementado por uma unidade devido à adição de um novo símbolo à mensagem descodificada e, por último, a variável good_decoded_size terá o mesmo valor.

Após o for loop acabar, serão incrementados por uma unidade o número de bits lidos, o valor de k e o nencode_idx, uma vez que não foi criado nenhum símbolo descodificado a partir dos bits que possuímos, passando a verificar o próximo bit da mensagem codificada e este será inserido na próxima posição do array bits.

Na circunstância em que o valor da variável contador equacionada anteriormente seja superior ao valor atual de _max_extra_symbols_, então esta última atualizará o seu valor com o novo máximo (contador).

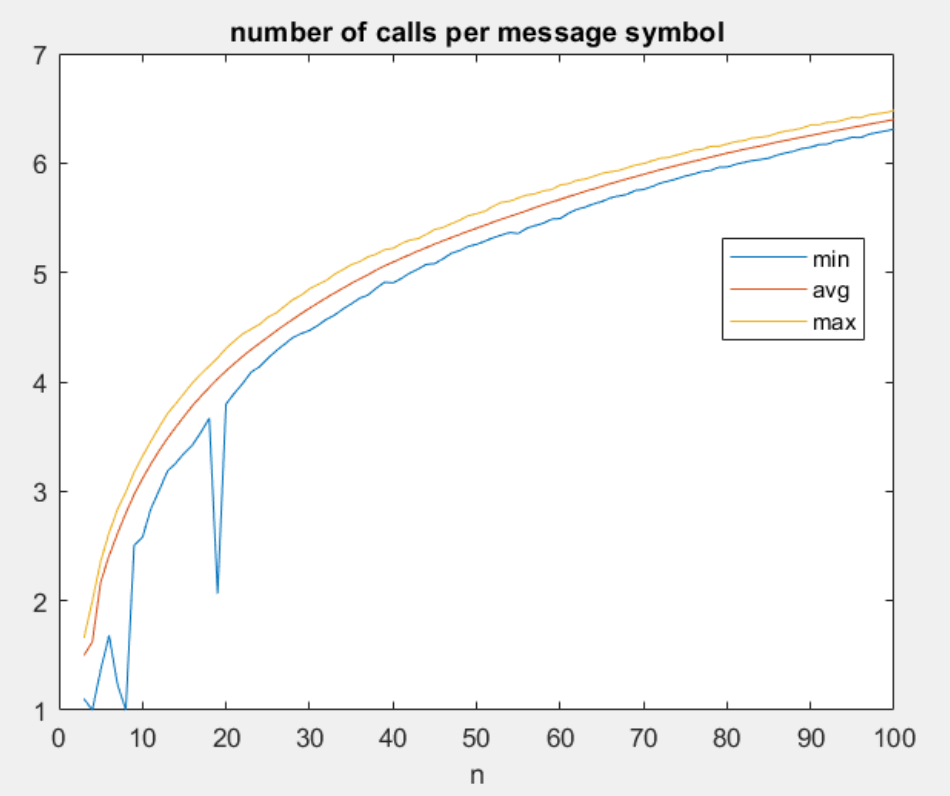
Resultados

Resultados obtidos ao correr o programa com a opção -t

```
alexandre@alexandre-X555LN:~/A03$ ./A03 -t 100 500 201
Número de soluções: 1
 100      6.104  61
```

Gráfico sobre o número de chamadas

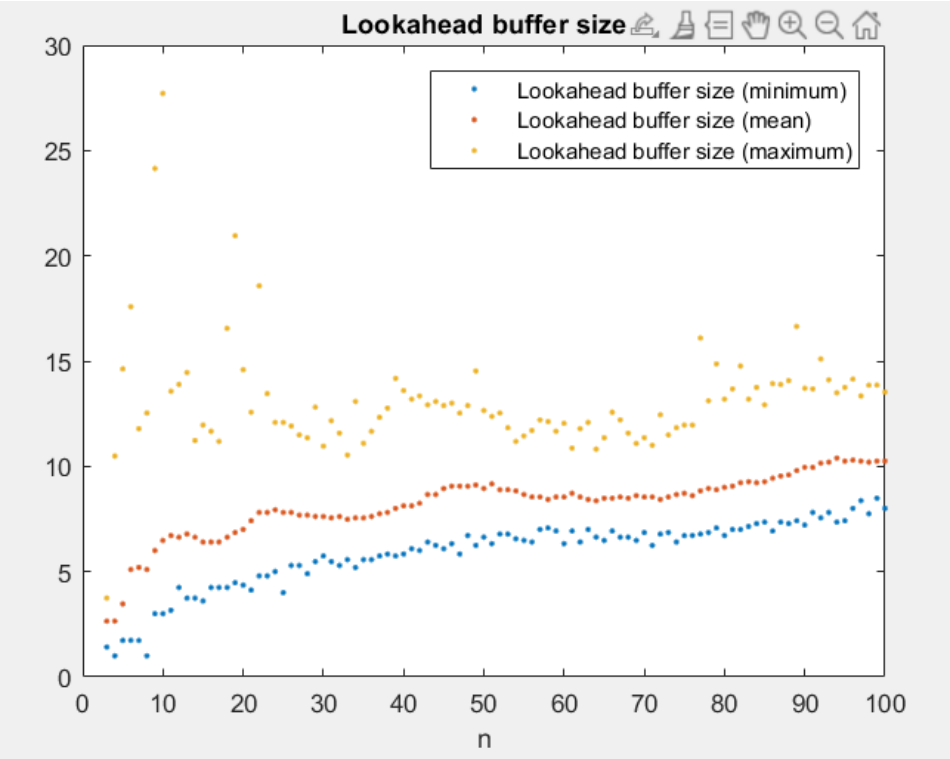
- Salientar que usamos mensagens de tamanho 10 000 e estes testes fazem referência às primeiras 201 seeds, pois corrermos o script bash dado pelo professor.



Ao analisar o gráfico, verificamos que o número de chamadas por símbolo crescem de forma logarítmica, algo que não estávamos muito à espera que fosse acontecer

Gráfico Lookahead buffer size

- Salientar que usamos mensagens de tamanho 10 000 e estes testes fazem referência às primeiras 201 seeds, pois corrermos o script bash dado pelo professor.



Neste gráfico, o eixo dos y sofreu uma distorção pela função raiz quadrada.

E conseguimos verificar observando o gráfico, que apesar de à medida que o valor de n aumenta tem a tendência do lookahead crescer, existem caso como podemos verificar para n=12 que cresce mas depois para valores de n = 13 e 14 que o valor decresce, formando uma espécie de onda.

Mas como o grupo já esperava, à medida que o valor de n cresce, regra geral, maior vai ser o tamanho da variável lookahead.

Conclusão

A realização deste trabalho permitiu-nos desenvolver o nosso raciocínio para resolver problemas de grande escala como este através do uso de funções recursivas. É impressionante o facto de poucas linhas de código conseguirem descodificar uma mensagem extensa em várias mensagens descodificadas diferentes de forma relativamente eficiente e rápida.

Este tipo de problemas acaba por ser muito interessante para alunos como nós que ainda não tem muita experiência tanto ao nível da programação como do trabalho em grupo pois acabamos por desenvolver inúmeras capacidades.

Anexo

Nesta secção achamos benéfico apenas colocar a parte do código que foi feito pelo grupo.

```
348 static void recursive_decoder(int encoded_idx,int decoded_idx,int good_decoded_size)
349 {
350   int controlo=0;
351   int contador=0;
352   int verifica=0;
353   int i;
354   for ( i= 0; i<_original_message_size_;i++ ){
355     if (i < decoded_idx ){
356       if(_decoded_message_[i] != _original_message_[i] ){
357           controlo=1;
358           verifica=i;
359           break;
360       }
361       else{
362         verifica=i;
363       }
364     }
365   }
366   contador=decoded_idx-verifica;
367   if(controlo == 0 && decoded_idx == _original_message_size_ ){
368     _number_of_solutions_++;
369     return ;
370   }
371   int  ndeco=decoded_idx,nencode_idx=encoded_idx, senencode_idx=encoded_idx;
372   int teste=0;
373   int bitLidos=1,k=0;
374   char bits[_c_->max_bits + 1];
```

```
375     while (bitLidos <= _c_->max_bits && _encoded_message_[nencode_idx] != 0 ){
376       bits[k]= _encoded_message_[nencode_idx];
377       bits[k+1]='\0';
378       for(int p = 0;p < _c_->n_symbols;p++){
379         teste=strcmp(bits,_c_->data[p].codeword);
380           if (teste== 0) {
381               _decoded_message_[ndeco]=p;
382               _number_of_calls_++;
383               recursive_decoder(senencode_idx+bitLidos,ndeco+1, good_decoded_size);
384           }
385       }
386       bitLidos++;
387       k++;
388       nencode_idx++;
389     }
390     if (contador > _max_extra_symbols_ ){
391       _max_extra_symbols_=contador;
392     }
393      return ;
394   }
```