DESCODIFICAÇÃO RECURSIVA DE UM CÓDIGO BINÁRIO

Algoritmos e Estruturas de Dados (40437)

2020/2021

universidade de aveiro

Trabalho realizado por:
Tiago Alvim 95584 50%
Vasco Costa 97746 50%

Introdução

O trabalho proposto consiste na descodificação de uma mensagem representada binariamente através de símbolos que a formam e nas várias possibilidades subjacentes, nomeadamente, o número de vezes que a função descodificadora tem de ser chamada, numa implementação recursiva, que facilita a sua solução, fazendo utilização de *depth-first* de forma eficaz e simples. E no estudo de ao final de quantos símbolos será possível verificar-se que o símbolo que se colocou está de facto errado. Isto acontece visto que, existem símbolos que apresentam partes em comum, no entanto, apenas um permite que a mensagem seja corretamente descodificada.

A codificação e a descodificação de mensagens são algo que sempre se apresentaram muito relevantes em diversas matérias, na guerra por exemplo. Sendo que uma das mais famosas máquinas de codificação e descodificação é a Enigma que era utilizada para mensagens de guerra durante a 2ª Guerra Mundial, acredita-se que também graças a ter-se conseguido decifrar as diversas cifras o conflito tenha acabado mais cedo.

Atualmente o ramo da criptografia é da mais alta importância e utilizada diariamente uma vez que permite que a informação pessoal seja guardada de quem a tenta utilizar, desde dados bancários a mensagens pessoais. No entanto, é necessário ter em atenção a sua aplicação. Por vezes faz-se o compromisso entre segurança e facilidade de uso, para o uso quotidiano não seria prático a utilização de uma palavra passe bastante grande com caracteres especiais, inclusive, para desbloquear o telemóvel todas as vezes que se pretendesse utilizar. Tornando-se numa cifra mais complexa e difícil ou até impossível de decifrar sem a chave, obrigando a uma abordagem *brute-force* que obrigaria a vários anos para a descodificar. A obtenção de uma palavra passe de *Wi-Fi* do tipo *WPA2*, dos mais utilizados, como nas casas de cada um, poderá levar desde segundos a milhares de anos consoante o método utilizado. A utilização de dicionários grandes e diversos poderá facilitar tal tarefa como a paralelização do processo em placas gráficas.

Abordagem computacional

recursive_decoder_1

Para resolver este problema a nossa primeira abordagem começou por percorrer todos os símbolos, passando a analisar-se o código binário desse mesmo símbolo. Na ocorrência de algum caracter diferente ao da mensagem codificada resulta nesse símbolo não ser possível pelo que se passa a analisar o símbolo seguinte.

Caso o símbolo seja possível, então comparamos com a mensagem original, se corresponde então é porque se descodificou um símbolo correto incrementando *good_decoded_size*. Aumentando o índice da mensagem descodificada, (*decoded_idx*), levando a chamar a mesma função para analisar o resto da mensagem. E como existe a possibilidade de haver mais do que um símbolo possível, nesse local, tem que se diminuir o *decoded_idx* e voltar a repor *encoded_idx* para se poder analisar com as condições iniciais, *backtracking*. No entanto, consideramos que isto não é o método mais desejado dado que, estava-se a comparar com a mensagem original e a adicionar apenas os símbolos corretos. Continuando-se a fazer a recursão tendo apenas atenção aos índices para se poder concretizar o estudo do número de chamadas à função e quantos símbolos teriam que ser percorridos até se chegar à conclusão de que era um símbolo incorreto. Para se fazer isto, ao início da função

verifica-se se já nos encontramos com símbolos extra, *good_decoded_size* será menor que *decoded_idx,* incrementando o número de chamadas à função.

recursive_decoder

Para evitar o ato menos honesto referido anteriormente alterou-se a função para se ir acrescentando sempre o símbolo possível à mensagem descodificada. Alcançando-se o final da mensagem codificada caso seja um símbolo de facto correto deixando-se esse permanecer, o número de soluções terá que ser superior ao que se tinha antes. Na eventualidade de este não ser o correto então coloca-se o símbolo anterior.

Para uma estimativa de complexidade computacional, afirmamos que terá um polinómio quadrático para representar o seu crescimento. Isto por se ter um ciclo para n símbolos todas as vezes, com uma passagem por x *bits* de cada símbolo sendo que se avança x *bits* também na mensagem codificada, acontecendo um certo número de vezes até se chegar ao final, tendo apenas uma constante associada.

bit_by_bit

Com o intuito de se realizar a contagem do maior número de casos a ter em consideração num dado instante esta função nasceu. Teve-se, no entanto, uma abordagem diferente à sugerida pelos docentes uma vez que já nos encontrávamos focados na ideia da descodificação *bit* a *bit*. Daí se fazer uso de uma pilha em vez de se fazerem alterações à função anterior, *recursive_decoder.*

Na idealização de um uso prático para esta função tivemos em mente não se saber se nos encontrávamos no início da mensagem, o que alterou bastante o raciocínio. Referimos prático no sentido de apenas se ter começado a fazer a leitura após o início ou caso tenha acontecido algum erro de transmissão da mensagem.

Deste modo a cada *bit* verifica-se se o início de cada símbolo é possível e caso possa ser então adiciona-se a uma pilha. Se houver algo na pilha, então, para o *bit* anterior um dado símbolo era possível pelo que tem que se verificar se o mesmo agora ainda o é. Em cada um destes casos verificamos que há mais uma possibilidade, e faz-se comparação com o valor obtido anteriormente. Para se poder fazer um estudo detalhado de como esta variação ocorre tiveram que se fazer algumas alterações no restante código. Por isso se inclui todo o código na secção código.

Não sabendo se os resultados aqui obtidos estão corretos para o ponto seguinte não se os decidiu utilizar. Daí a utilização de números grandes para garantirmos que não resultaria daí algum problema, à custa de maior uso de memória.

bit_decoder_2 e *bit_decoder_3*

O objetivo desta função era realizar a descodificação *bit* a *bit*, realizando a descoberta de forma *breadth-first.* Para o efeito realiza-se uma passagem inicial por todos os símbolos e se o *bit* inicial do símbolo corresponder ao primeiro da mensagem então coloca-se um objeto *auxiliary* numa pilha. O objeto *auxiliary* é caracterizado por ter a indicação do símbolo que representa, da posição em que nos encontramos a ler o *bit* do símbolo e um ponteiro para o símbolo possível anterior que originou este.

Seguidamente tiram-se todos os símbolos da pilha e verifica-se se o bit seguinte ainda corresponde ao da mensagem. E caso corresponda então guarda-se numa estrutura secundária para se poder esvaziar a primeira. E assim sucessivamente. Chegando ao ponto em

que é o final de um símbolo concluimos que esse é possível na mensagem pelo que temos que procurar quais serão os símbolos que poderão ser o começo. Ao chegar ao fim da mensagem codificada, através do ponteiro na estrutura *auxiliary* consegue-se descodificar a mensagem do fim para o princípio.

No entanto, não conseguimos obter resultados corretos. Por vezes a mensagem descodificada apresenta sempre o mesmo símbolo. Isto leva-nos a crer que não conseguimos implementar corretamente o uso de ponteiros e endereços para se conseguir fazer a atribuição dos símbolos corretos à mensagem descodificada.

Também se tentou fazer a implementação deste método com uma fila, mas os resultados obtidos foram idênticos. Esta tentativa foi pensada por usualmente se utilizarem filas em procuras *breadth-first* mas nem sempre é necessário fazer recurso a tal, como verificámos. Também seria irrelevante a estrutura de armazenamento utilizada uma vez que se retiravam todos os elementos, mas teve-se mais interesse em fazer também a implementação de estruturas de armazenamento de forma a se aprender mais.

Análise[1]

Os dados seguintes foram obtidos utilizando o comando *-x* para diversos números de símbolos, com a utilização de 100 testes para cada, com 80 medições válidas e 20 descartadas.

Número de chamadas à função

Um aspeto que pode ser relevante na análise destas situações será possivelmente o número de vezes que a função é chamada recursivamente e como é possível visualizar, o resultado obtido é de facto muito curioso, verificar que tem o aspeto de uma função logarítmica e de facto, fazendo a aproximação a um polinómio do tipo $a * \log(b * n)$ obteve-se um r^2 de 0,9999 com o polinómio $1.438 * \log(0.8635 * n)$ e não assumindo que depende de um valor para b obteve-se $1.401 * \log(n)$ com um valor de r^2 de 0.9992. Assim, pode-se assumir que para valores superiores o mesmo se verificará. Isto poderá tornar-se significativo para a descodificação para um número de símbolos superior uma vez que um crescimento de ordem logarítmico é comparativamente lento com um crescimento de ordem n, por exemplo. Como se verifica de forma clara, o nível de recursões necessárias para a descodificação cresce de forma logarítmica com o número de símbolos levando a que uma duplicação de n não

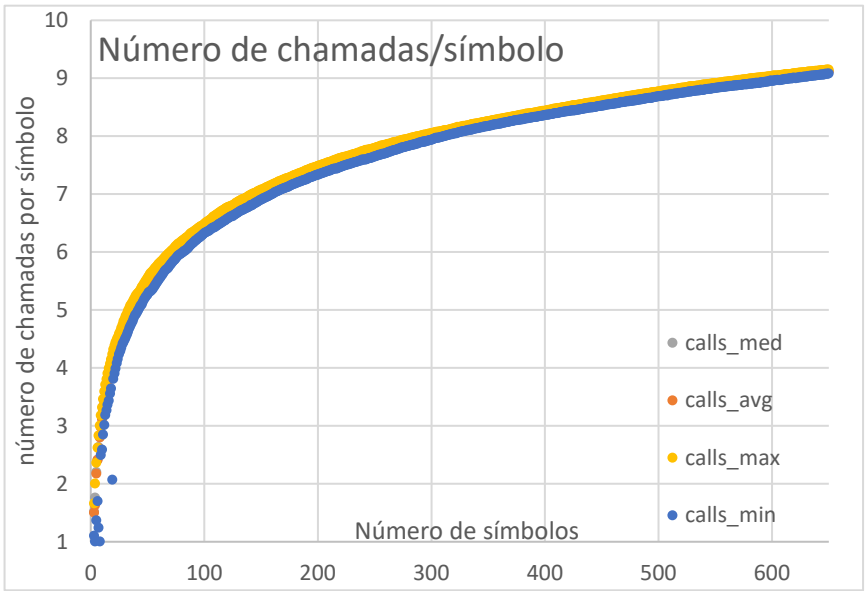

Gráfico 1

[1] Resultados obtidos em *Ryzen 7 3700X* com uma velocidade base de *3,59 GHz*. E 16GB de memória RAM a 3333 MHz com CL16 em dual-channel.

necessite do dobro de memória para o stack da recursão, algo que se torna bastante necessário a ter em consideração.

Número máximo de símbolos

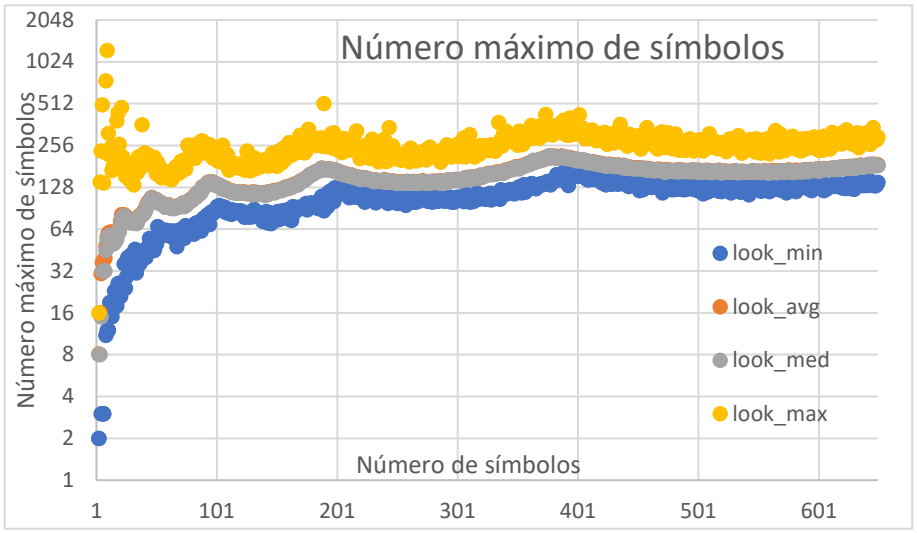

Gráfico 2

Para esta situação a obtenção de um polinómio que consiga representar o crescimento deste número de casos é de facto mais complexo e poderá nem fazer grande sentido. Uma vez que se verificam a existência de picos e para o número máximo de símbolos analisados até se chegar à conclusão de que a mensagem estava errada é bastante variável, de 16 a 1238, para pouca variação no número de símbolos. No entanto, os picos referidos ocorrem aproximadamente, tanto para o número máximo como para o número mínimo e para a mediana num mesmo número de n. Sem consistência no crescimento de n.

Através da visualização do gráfico é possível verificar-se que *look_max* começa por ser bastante disperso passando a agrupar-se para um número superior. Assim como *look_min* e o conjunto destes passa a ter uma dispersão menor, aproximando-se dos valores da mediana.

Na análise destes resultados para um número de símbolos entre 0 e 100, aproximadamente, é possível verificar-se que esta se poderá assemelhar a uma função logarítmica. No entanto, na tentativa da realização de uma aproximação todas as tentativas apresentaram um valor para r^2 abaixo de 0,90 pelo que se acabou por excluir a hipótese de este elemento ter um crescimento polinomial, assim como os outros.

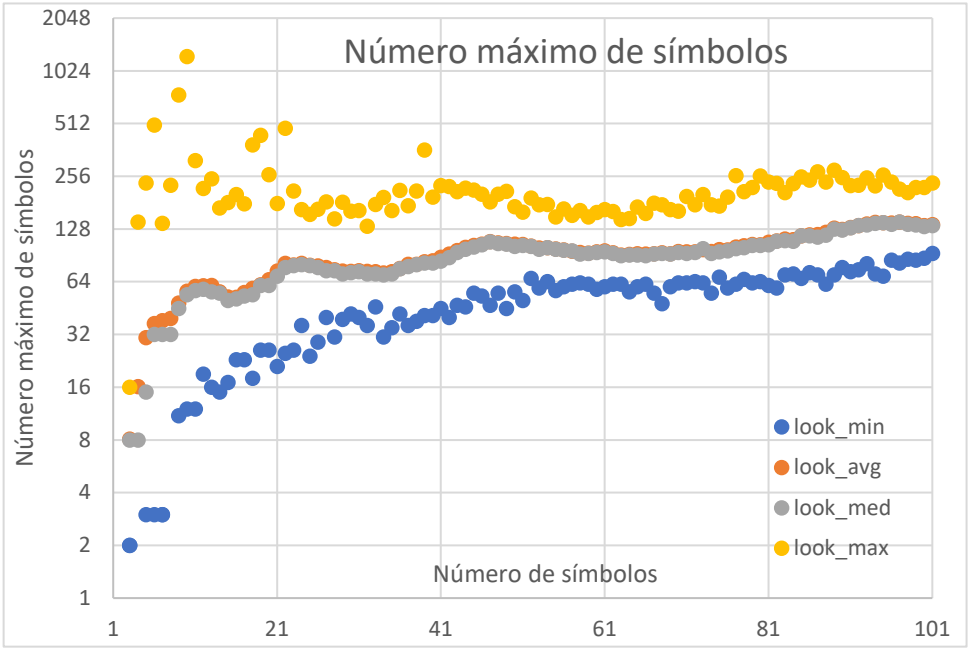

Gráfico 3

bit_by_bit

Fazendo a alteração na parte do código na parte de análise de forma a que este faça a mesma análise para esta função obtêm-se os resultados que se conseguem visualizar no gráfico seguinte.

Através de uma análise ao seguinte gráfico consegue-se verificar que existe um crescimento linear dos resultados obtidos com o crescimento do número de símbolos. De forma explicita verificamos que segue uma reta de declive 1 e ordenada na origem de menos 1. $y = n - 1$, com r^2 de 1. Assim sendo, podemos considerar um aumento necessário do tamanho do *stack* de forma linear.

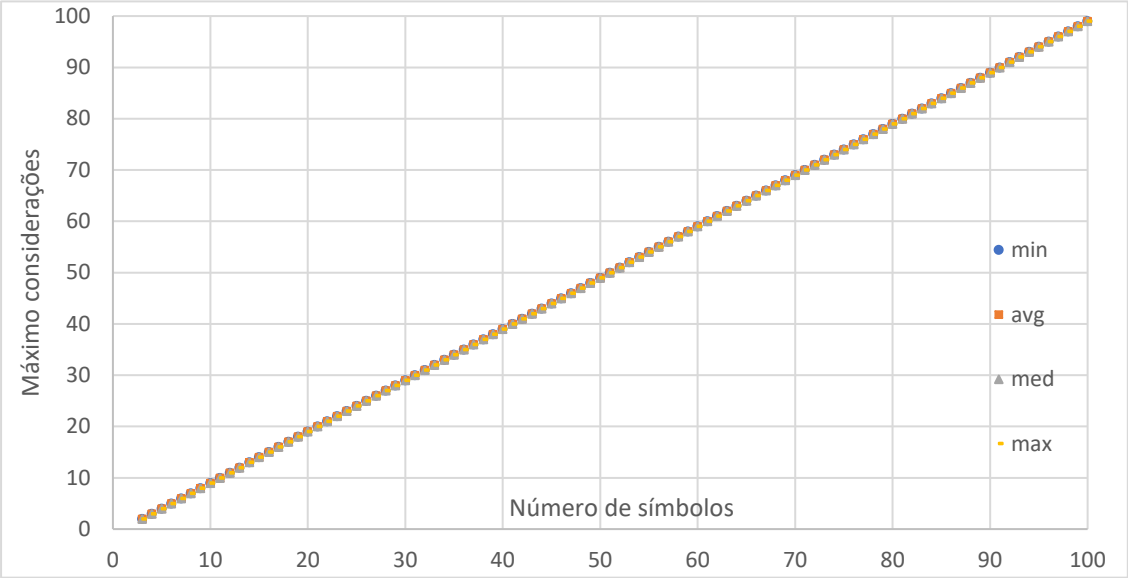

Gráfico 4

Na tentativa do ponto seguinte optou-se por usar um número bastante grande de forma a não se encontrarem erros devido a um tamanho da estrutura de armazenamento pequeno.

Posteriormente, testou-se se de facto era necessário o uso de um valor tão grande e verificámos que a aproximação linear seria provavelmente a melhor opção. Uma vez que a utilização de um número inferior ao aqui observado indicava que era necessário adicionar mais um elemento à estrutura sendo tal impossível, valores muito superiores apenas fariam uso de mais memória consumindo recursos desnecessariamente. Posto isto, a aposta de um crescimento linear é o desejado sugerindo-se a alteração da ordenada na origem para um valor arbitrário positivo.

Conclusão

Através da realização deste projeto conseguimos verificar que a codificação e descodificação de mensagens são bastante pertinentes nos dias de hoje. Apresentado pormenores que à partida poderiam não ser considerados, como a profundidade necessária a alcançar para a obtenção de uma solução correta, demonstrando também a desvantagem natural de uma abordagem *depth-first*.

Com uma solução *breadth-first* a funcionar corretamente seria interessante analisar e comparar a execução com o método recursivo implementado, tanto em termos de desempenho como de uso de recursos.

Referências

Material de aula disponibilizado no eLearning

Código

```
//
// AED, 2020/2021
//
// Decoding a non-instantaneous binary code
//

#include <stdio.h>
#include <stdlib.h>

//
// Compile time parameters
//
#ifndef MAX_N_SYMBOLS
# define MAX_N_SYMBOLS        1000 // 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_size + 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_size
#define _max_encoded_message_size_  decoder_global_data.max_encoded_message_size
#define _encoded_message_           decoder_global_data.encoded_message
#define _max_decoded_message_size_  decoder_global_data.max_decoded_message_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_solutions
#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")


// primeira tentativa
static void recursive_decoder_1(int encoded_idx,int decoded_idx,int good_decoded_size)
{
   int b = (decoded_idx - good_decoded_size);
            _max_extra_symbols_ = ( b > _max_extra_symbols_)? b : _max_extra_symbols_;

   _number_of_calls_++;

            int possible;
   int encoded_idx_old = encoded_idx;
            for(int i = 0; i < _c_->n_symbols ; i++)//percorrer os simbolos
            {
      possible = 0;//consideramos o simbolo possivel
      for(int p = 0; (_c_->data[i].codeword[p] != '\0'); p++ )//percorrer o codigo desse simbolo
      {
         if(_encoded_message_[encoded_idx++] != _c_->data[i].codeword[p])
         {
            possible = -1;//o codigo do simbolo e diferente logo nao e possivel e passa-se ao proximo simbolo
            break;
         }
      }

      if(possible == 0)//se o simbolo e possivel
      {

         if( i == _original_message_[decoded_idx])//se o simbolo for igual ao da mensagem
```

```
        {
          _decoded_message_[decoded_idx] = i;//guarda-se na descodificada
          good_decoded_size++;//o good size aumenta
          if(good_decoded_size == _original_message_size_)//se tivermos descodificado tudo
          {
            _number_of_solutions_++;//aumenta-se o numero de solucoes
          }
        }
        decoded_idx++;//descodificou-se mais um simbolo
        recursive_decoder_1(encoded_idx,decoded_idx,good_decoded_size);
        decoded_idx--;//para simbolo seguinte tem que se voltar a repor
      }
      encoded_idx = encoded_idx_old;//volta-se a meter o encoded no inicio para o proximo simbolo
    }
}

static void recursive_decoder(int encoded_idx,int decoded_idx,int good_decoded_size)
{
   int b = (decoded_idx - good_decoded_size);
            _max_extra_symbols_ = ( b > _max_extra_symbols_)? b : _max_extra_symbols_;

   _number_of_calls_++;

   int possible;
   int encoded_idx_old = encoded_idx;
             for(int i = 0; i < _c_->n_symbols ; i++)//percorrer os simbolos
             {
      possible = 0;
      for(int p = 0; (_c_->data[i].codeword[p] != '\0'); p++ )//percorrer o codigo desse simbolo
      {
        if(_encoded_message_[encoded_idx++] != _c_->data[i].codeword[p])
        {
          possible = -1;//o codigo do simbolo e diferente logo nao e possivel e passa-se ao proximo simbolo
          break;
        }
      }
      if(possible == 0)//se o simbolo e possivel
      {
        if (good_decoded_size == decoded_idx && _original_message_[decoded_idx] == i)
        {
          good_decoded_size++;
        }
        int prev = _number_of_solutions_;
        int prev_rec = _decoded_message_[decoded_idx];
        decoded_idx++;//descodificou-se mais um simbolo
        recursive_decoder(encoded_idx,decoded_idx,good_decoded_size);
        decoded_idx--;//para simbolo seguinte tem que se voltar a repor
        if(_number_of_solutions_ > prev)
          prev_rec = i;
        _decoded_message_[decoded_idx] = prev_rec;
      }
      encoded_idx = encoded_idx_old;//volta-se a meter o encoded no inicio para o proximo simbolo
   }
   if (_encoded_message_[encoded_idx] == '\0')
   {
     _number_of_solutions_++;
    //faz-se return aqui se quisermos so ver qual a mensagem descodificada corretamente, não e objetivo
   }
}

typedef struct auxiliary
{  int symbol;//o simbolo que é
   int position;//a posicao em que esse simbolo esta
```

```
   struct auxiliary *prev_symbol;
}auxiliary;
#define MAX_QS_SIZE 1000
typedef struct
{   int top;
   auxiliary items[MAX_QS_SIZE];
}stack_n;
//usar ponteiros para se ter apenas um stack e nao se fazerem alteracoes so la dentro
//ser global facilita
int stack_empty(stack_n *s){return ( (s->top == -1 )? 1:0 );}

int stack_is_full(stack_n *s){return ( (s->top == MAX_QS_SIZE-1 )? ( fprintf(stderr,"small stack")) : 0);}

auxiliary stack_peek(stack_n *s) { return s->items[s->top];}

auxiliary stack_pop(stack_n *s)
{
   if(!stack_empty(s))
   {
      s->top -= 1;
      return s->items[s->top+1];
   }
   else
      fprintf(stderr,"pop_empty");
}

void stack_put(stack_n *s,auxiliary a)
{
   if(!stack_is_full(s))
   {
      s->top += 1;
      s->items[(s->top)] = a;
   }
}
//stack teste no link
//https://onlinegdb.com/3JXMVaeeZ
typedef struct
{   auxiliary items[MAX_QS_SIZE];
   int front;
   int rear;
   int count;
}queue;
auxiliary q_peek(queue *q)
{ return q->items[q->front];}
int q_empty(queue *q)
{ return ((q->count == 0)?1:0);}
int q_full(queue *q)
{ return ((q->count == MAX_QS_SIZE)?1:0);}
void q_put(queue *q,auxiliary a)
{ if(!q_full(q))
  {
   if(q->rear == MAX_QS_SIZE-1)
      q->rear = -1;
   q->items[++(q->rear)] = a;
   q->count+=1;
  }
  else
   fprintf(stderr,"small max size");
}
auxiliary q_get(queue *q)
{
   if(!q_empty(q))
   {
```

```
        auxiliary a = q->items[(q->front)++];
        if(q->front == MAX_QS_SIZE)
            q->front = 0;
        q->count-=1;
        return a;
    }
    else
        fprintf(stderr,"empty queue");
}
// queue tester
// https://onlinegdb.com/Hy3KDRR1_

int MAX_CONSIDER_ONCE;
static void bit_by_bit(int encoded_idx)
{
    MAX_CONSIDER_ONCE = 0;
    stack_n arr;
    arr.top = -1;

    int n_max;

    for(; _encoded_message_[encoded_idx] != '\0';encoded_idx++)
    {
        stack_n aux;
        aux.top = -1;
        n_max = 0;
        while( !stack_empty(&arr) )//tirar tudo do stack e ver se cada simbolo ainda e possivel
        {
            auxiliary s = stack_pop(&arr);
            if(_c_->data[s.symbol].codeword[s.position] != '\0')
            {
                if(_encoded_message_[encoded_idx] == _c_->data[s.symbol].codeword[s.position])
                {
                    s.position += 1;
                    stack_put(&aux,s);//vai para um auxiliar para se esvaziar o original
                    n_max++;//e uma possiblidade nesta bit da codificada
                }
            }//se for o fim entao chegou a ser um simbolo possivel
        }
        arr = aux;
        for(int i = 0; i < _c_->n_symbols ; i++)//se for começo
        {
            if(_encoded_message_[encoded_idx] == _c_->data[i].codeword[0])
            {
                auxiliary new;
                new.symbol = i;
                new.position = 1;//mete se logo 1 que para se ler logo nessa posicao
                stack_put(&arr,new);
                n_max++;
            }
        }
        MAX_CONSIDER_ONCE = (MAX_CONSIDER_ONCE < n_max)? n_max:MAX_CONSIDER_ONCE;
    }
}

static void bit_decoder_2(int encoded_idx, int decoded_idx)
{
    queue arr;
    arr.front = 0;
    arr.rear = -1;
    for(int i = 0; i < _c_->n_symbols ; i++)
    {
        if(_encoded_message_[encoded_idx] == _c_->data[i].codeword[0])
```

```
      {
         auxiliary new;
         new.symbol = i;
         new.position = 1;//mete se logo 1 que para se ler logo nessa posicao
         q_put(&arr,new);
      }
   }
queue aux;auxiliary a;
while(1)
{
   aux.front = 0;
   aux.rear = -1;

   encoded_idx++;
   while(!q_empty(&arr))
   {
      a = q_get(&arr);
      if(_encoded_message_[encoded_idx] == _c_->data[a.symbol].codeword[a.position])//se o bit coincide
      {
         if(_encoded_message_[encoded_idx] == '\0')//se for o fim da mensagem
         {
            for(decoded_idx = (_original_message_size_ - 1); decoded_idx >= 0;decoded_idx--)
            {
               printf("%d\n",a.symbol);
               _decoded_message_[decoded_idx] = a.symbol;
               a = *a.prev_symbol;
            }
            return;
         }
         a.position += 1;
         q_put(&aux,a);
      }
      else if(_c_->data[a.symbol].codeword[a.position] == '\0')//bit nao coincide;mas e um simbolo possivel do meio
da mensagem
      {  //ver o proximo simbolo possivel
         for(int i = 0; i < _c_->n_symbols ; i++)
         {
            if(_encoded_message_[encoded_idx] == _c_->data[i].codeword[0])
            {
               auxiliary new;
               new.symbol = i;
               new.position = 1;
               new.prev_symbol = &a;
               q_put(&aux,new);
            }
         }
      }
   }
   arr = aux;
}
}
static void bit_decoder_3(int encoded_idx, int decoded_idx)
{
   stack_n arr;
   arr.top = -1;

   for(int i = 0; i < _c_->n_symbols ; i++)
   {
      if(_encoded_message_[encoded_idx] == _c_->data[i].codeword[0])
      {
         auxiliary new;
         new.symbol = i;
         new.position = 1;//mete se logo 1 que para se ler logo nessa posicao
```

```
            stack_put(&arr,new);
        }
    }
auxiliary a;
while(1)
{
    stack_n aux;
    aux.top = -1;

    encoded_idx++;
    while(!stack_empty(&arr))
    {
        a = stack_pop(&arr);
        if(_encoded_message_[encoded_idx] == _c_->data[a.symbol].codeword[a.position])//se o bit coincide
        {
            if(_encoded_message_[encoded_idx] == '\0')//se for o fim da mensagem
            {
                for(decoded_idx = (_original_message_size_ - 1); decoded_idx >= 0;decoded_idx--)
                {
                    printf("%d\n",a.symbol);
                    _decoded_message_[decoded_idx] = a.symbol;
                    a = *a.prev_symbol;
                }
                return;
            }
            a.position += 1;
            stack_put(&aux,a);
        }
        else if(_c_->data[a.symbol].codeword[a.position] == '\0')//bit nao coincide;mas e um simbolo possivel do meio
da mensagem
        {   //ver o proximo simbolo possivel
            for(int i = 0; i < _c_->n_symbols ; i++)
            {
                if(_encoded_message_[encoded_idx] == _c_->data[i].codeword[0])
                {
                    auxiliary new;
                    new.symbol = i;
                    new.position = 1;
                    new.prev_symbol = &a;
                    stack_put(&aux,new);
                }
            }
        }
    }
    arr = aux;
}
}
//
// 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_size_ * 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_size_,_original_message_);

encode_message(_c_,_original_message_size_,_original_message_,_max_encoded_message_size_,_encoded_mess
age_);

 #if 0
  recursive_decoder(0,0,0);
  bit_by_bit(0);
  printf("Numero de consideracoes %d\n",MAX_CONSIDER_ONCE);
  bit_decoder_2(0,0);
 #else
  printf("original: ");
  for(int i=0;i<_original_message_size_;i++)
  {
     printf("%d",_original_message_[i]);
  }
  printf("\n");
  printf("messagem codificada %s \n",_encoded_message_);

  recursive_decoder(0,0,0);
  bit_by_bit(0);
  printf("Numero de consideracoes %d\n",MAX_CONSIDER_ONCE);
  bit_decoder_3(0,0);

  printf("Decoded:  ");
  for(int i = 0;i<_original_message_size_ ;i++)
  {
    printf("%d",_decoded_message_[i]);

    if(_decoded_message_[i] !=_original_message_[i])
       printf("\nERRO\n");

  }
  printf("\n");

 #endif
 //
 #if 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_);
 }
 #endif
 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_extra_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].codeword);
    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_MEASUREMENTS],u_avg;
int u,u_min,u_max,u_data[N_MEASUREMENTS];
int seed,n_symbols,i;
code_t *c;

//
double m_avg,m_data[N_MEASUREMENTS];
int m,m_min,m_max;
//
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  max considerations\n");
 printf("#    ------------------------------------  ---------------------  ------------------\n");
 printf("# ns      min     avg     med     max   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;
m_min = m_max = 0.0;
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_;
 m = MAX_CONSIDER_ONCE;
 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;
 //
 if(seed == 1 || m < m_min)
  m_min  = m;
 if(seed == 1 || m > m_max)
  m_max  = m;
 //
 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;
 //
 for(i = seed - 1;i > 0 && m_data[i - 1] > m;i--)
  m_data[i] = m_data[i - 1];
 m_data[i] = m;
 //
```

```
    }
    t_avg = u_avg = m_avg = 0.0;
    for(i = N_OUTLIERS;i < N_MEASUREMENTS - N_OUTLIERS;i++)
    {
      t_avg += t_data[i];
      u_avg += (double)u_data[i];
      m_avg += m_data[i];
    }
    t_avg /= (double)(2 * N_VALID + 1);
    u_avg /= (double)(2 * N_VALID + 1);
    m_avg /= (double)(2 * N_VALID + 1);
    printf("%4d  %8.3f %8.3f %8.3f %8.3f  %4d %6.1f %4d %4d  %3d %3.1f %3.1f
%4d\n",n_symbols,t_min,t_avg,t_data[N_OUTLIERS + N_VALID],t_max,u_min,u_avg,u_data[N_OUTLIERS +
N_VALID],u_max,m_min,m_avg,m_data[N_OUTLIERS + N_VALID],m_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;
}
```