

**Trabalho Prático 2**

**Word Ladder**

**Licenciatura em Engenharia Informática**

**Relatório**

**Turma P8**

**Professores:**

Joaquim Madeira ([*jmadeira@ua.pt*](mailto:jmadeira@ua.pt))

Tomás Oliveira da Silva ([*tos@ua.pt*](mailto:tos@ua.pt))

**Realizado por:**

Diogo Falcão Nº108712 – **50%**

José Gameiro Nº108840 – **50%**

08/01/2023

***Índice:***

[***1.*** ***Introdução*** 3](#_Toc124099423)

[***2.*** ***Desenvolvimento*** 4](#_Toc124099424)

[**2.1.** **Estruturas** 4](#_Toc124099425)

[**2.2.** **Funções** 6](#_Toc124099426)

[*2.2.1.* *allocate\_adjency\_node ()* 6](#_Toc124099427)

[*2.2.2.* *free\_adjency\_node ()* 6](#_Toc124099428)

[*2.2.3.* *allocate\_hash\_table\_node ()* 7](#_Toc124099429)

[*2.2.4.* *free\_hash\_table\_node ()* 7](#_Toc124099430)

[*2.2.5.* *hash\_table\_create ()* 8](#_Toc124099431)

[*2.2.6.* *hash\_table\_free ()* 8](#_Toc124099432)

[*2.2.7.* *hash\_table\_grow ()* 9](#_Toc124099433)

[*2.2.8.* *find\_word ()* 10](#_Toc124099434)

[*2.2.9.* *print\_hash\_table ()* 12](#_Toc124099435)

[**2.3.** **Resultados** 14](#_Toc124099436)

[***3.*** ***Código*** 19](#_Toc124099437)

[**3.1.** **Código em C** 19](#_Toc124099438)

[**3.2.** **Código em Matlab** 28](#_Toc124099439)

[***4.*** ***Conclusão*** 29](#_Toc124099440)

# ***Introdução***

No âmbito da cadeira de Algoritmos e Estruturas de Dados (AED), foi-nos proposto realizar uma *Word Ladder* ou uma *escada de palavras*. Para isto, foi necessário recorrer a conceitos que foram lecionados nas aulas teóricas e práticas tais como: a implementação de métodos para o desenvolvimento de uma *Hash Table*, a utilização de *Linked Lists* e propriedades e teoremas de grafos.

Uma *Word Ladder* ou uma *escada de palavras* consiste em selecionar duas palavras de dentro de um conjunto constituído por várias palavras diferentes e, para estas duas palavras, ser possível partir-se de uma e chegar à outra alterando apenas uma letra em cada passo. Tal como referido, as palavras encontram-se todas num conjunto, por outras palavras, numa *Hash Table*.

Uma *Hash Table* é uma estrutura de dados que usa funções hash para armazenar e recuperar dados de forma rápida. As funções são usadas para converter os dados numa chave de hash, que é então usada para armazenar os dados na tabela de hash. É possível ocorrerem colisões de hash, em que ao utilizar a mesma função de hash em uma ou mais chaves, é gerado o mesmo valor de hash para cada chave. Quando isto acontece é criada uma *Linked List* para armazenar as diferentes chaves com o mesmo valor de hash. Uma *Linked List* é, também, uma estrutura de dados de listagem linear que é composta por nós. Cada nó contém um campo para armazenar dados e um ponteiro para o próximo nó presente na lista, à exceção do último nó, que aponta para *NULL* (indicando o final da lista). Desta forma, as *Linked List* ajudam a melhorar o desempenho das pesquisas.

Este relatório irá demonstrar e explicar as funções criadas para a implementação da *Hash Table*, o raciocínio que tivemos para a sua implementação, bem como os testes que fizemos para perceber se a nossa implementação estava correta.

# ***Desenvolvimento***

Para a implementação da *Word Ladder*, foi-nos disponibilizado um conjunto de ficheiros, no qual se encontrava um script, designado por *word\_ladder.c*. Este encontrava-se com várias funções incompletas para a implementação de uma *Hash Table* e de um grafo em C. Também se encontravam presentes cinco ficheiros de texto que foram utilizados para testar a nossa *Hash Table*:

* ***wordlist-big-latest.txt*:** contém um número muito elevado de palavras com diferentes tamanhos;
* ***wordlist-four-letters.txt*:** contém várias palavras com tamanho de quatro letras, que foram retiradas do ficheiro ***wordlist-big-latest.txt***;
* ***wordlist-five-letters.txt*:** contém várias palavras com tamanho de cinco letras, que foram retiradas do ficheiro ***wordlist-big-latest.txt***;
* ***wordlist-six-letters.txt*:** contém várias palavras com tamanho de seis letras, que foram retiradas do ficheiro ***wordlist-big-latest.txt***;
* ***teste.txt*:** foi um ficheiro criado por nós, que contém um número de palavras reduzido, usado maioritariamente para pequenos testes usando a nossa implementação.

De seguida, iremos explicar algum conteúdo que se encontra já implementado no ficheiro word\_ladder.c, bem como o código que adicionámos às funções que se encontravam incompletas.

## **Estruturas**

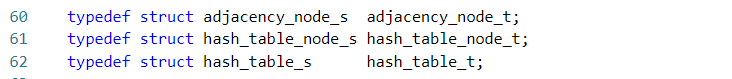


Fig.1 – Estruturas criadas

Foram criadas três estruturas para a implementação da *Hash Table* e do grafo:

* ***adjency\_node\_s* ou *adjency\_node\_t***: A primeira estrutura denomina-se por *adjency\_node\_s*, que representa um nó de uma lista de adjacência. Esta estrutura é utilizada para a representação do grafo, em que cada nó da lista armazena um vértice e um conjunto de vértices adjacentes a esse vértice. Tem como atributos a variável *next* que é um ponteiro que é do tipo de ***adjency\_node\_t*** e a variável *vertex* que também é um ponteiro que é do tipo ***hash\_table\_node\_t*.**

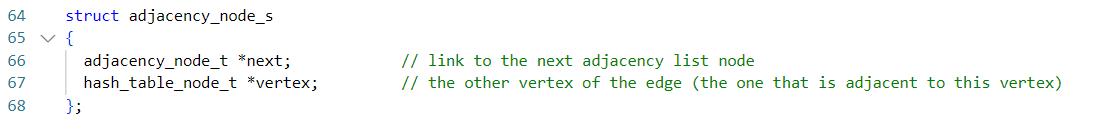


Fig.2 – Estrutura *adjency\_node\_s*

* ***hash\_table\_node\_s* ou *hash\_table\_node\_t***: esta estrutura representa um nó da *Hash Table*. Tem como atributos:

- ***word*:** será a palavra que irá ficar guardada no nó, esta é um vetor de caracteres que podem ter como tamanho máximo 32 caracteres;

- ***next***: é um ponteiro para outro nó da *Hash Table*, que é usado para ligar os nós através de uma *Linked List* dentro da *Hash Table*;

- ***head***: é um ponteiro para o primeiro nó da lista de adjacência;

- ***visited***: uma variável do tipo inteiro que indica se o vértice foi visitado ou não;

- ***previous***: representa um ponteiro para o nó anterior, que irá ser utilizado para a função breadh\_first\_search;

- ***representative***: é um ponteiro para o nó representante da componente conexa ao qual esse vértice pertence.

- ***number\_of\_vertices***: uma variável do tipo integer que indica o número de vértices da componente conexa à qual o vértice pertence.

- ***number\_of\_edges***: uma variável do tipo inteiro que indica o número de arestas da componente conexa à qual pertence o vértice.

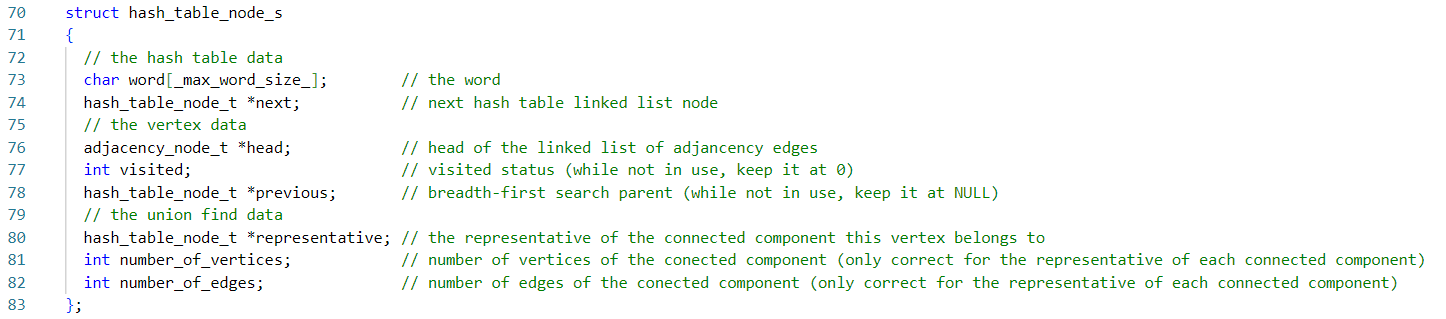


Fig.3 – Estrutura *hash\_table\_node\_s*

* ***hash\_table\_s* ou *hash\_table\_t*:** esta estrutura é utilizada para representar a *Hash Table*, e tem como atributos:

**- *hash\_table\_size*:** uma variável do tipo integer que indica o tamanho da *Hash Table*;

**- *number\_of\_entries*:** é uma variável do tipo integer que indica o número de entradas na *Hash Table*;

**- *heads*:** representa um ponteiro que aponta para um vetor do tipo *hash\_table\_node\_t*. Ao se inserir um elemento neste vetor ele irá ser inserido no início de uma lista ligada.

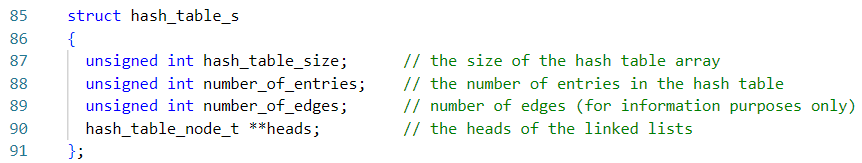
****

Fig.4 – Estrutura *hash\_table\_s*

## **Funções**

Neste ponto iremos explicar algumas funções que nos foram fornecidas e as funções que completámos.

### *allocate\_adjency\_node ()*

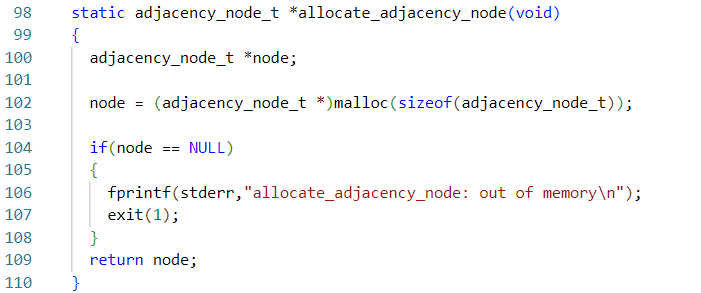


Fig.5 – Função *allocate\_adjency\_node ()*

Esta função tem como objetivo alocar de forma dinâmica um nó de uma lista de adjacência. Começa por declarar a variável *node* e, usando a função *malloc*, aloca um bloco de memória do tamanho especificado pelo tamanho da estrutura *adjency\_node\_t*. Caso a função *malloc* não funcione, ou seja, se a variável *node* for igual a NULL, isto significa que não existe memória suficiente para a alocação e a função imprime uma mensagem de erro. Caso a função malloc funcione, a função retorna um ponteiro para o bloco de memória alocado.

Esta função é simples e bastante útil pois simplifica o processo de alocação de nós da lista de adjacência que virá a ser necessário utilizar noutras funções.

### *free\_adjency\_node ()*

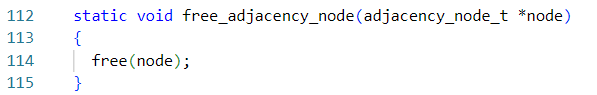


Fig.6 – Função *allocate\_adjency\_node ()*

A função *free\_adjency\_node* é uma função bastante simples pois apresenta apenas uma única instrução que consiste em utilizar a função *free ()* para libertar o espaço de memória alocado para o nó pretendido (o argumento de entrada da função). É importante libertar a memória alocada quando esta já não for necessário, para evitar vazamentos de memória.

### *allocate\_hash\_table\_node ()*

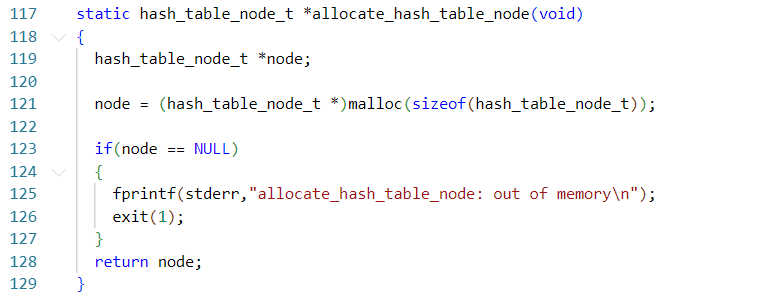


Fig.7 – Função *allocate\_hash\_table\_node ()*

A função *allocate\_hash\_table\_node* é muito semelhante *allocate\_adjency\_node ()*, pois ambas têm o mesmo objetivo que é alocar memória para um nó, só que no caso desta função o bloco de memória alocado irá ser atribuído a um nó da *Hash Table*.

### *free\_hash\_table\_node ()*

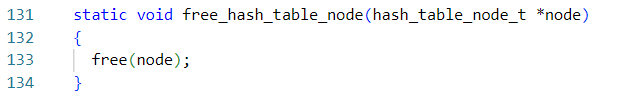


Fig.8 – Função *free\_hash\_table\_node ()*

Esta função é bastante semelhante à função *free\_adjency\_node ()*, pois ambas apresentam apenas uma instrução que tem o mesmo objetivo para libertar o espaço alocado, só que nesta função o espaço libertado será o alocado para um nó da *Hash Table*.

### *hash\_table\_create ()*

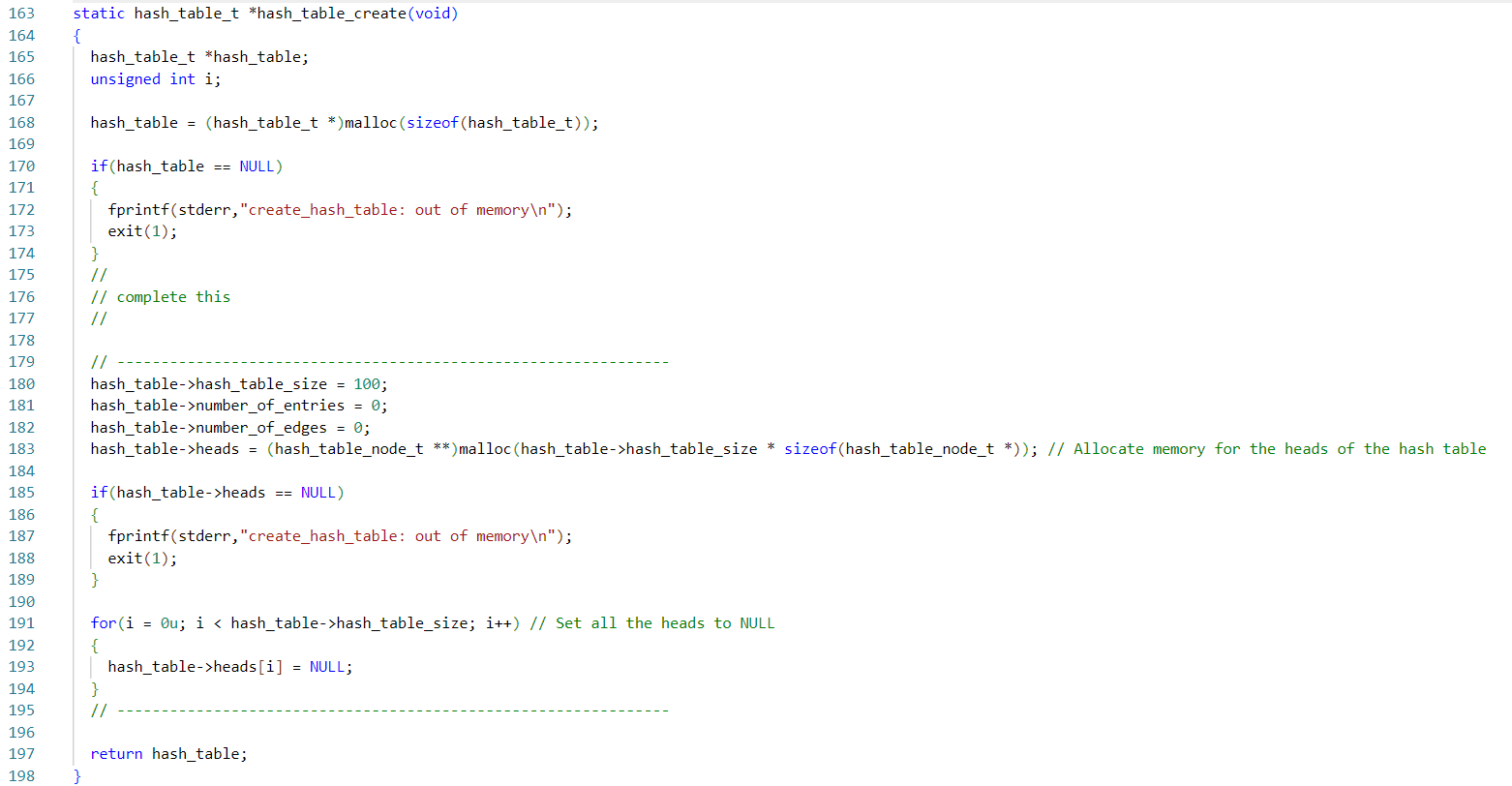
****

Fig.9 – Função *hash\_table\_create ()*

Esta função serve para inicializar uma *Hash Table*. Como primeiro passo é alocar um bloco de memória, através da função *malloc*, para a *Hash Table*, com tamanho igual ao da estrutura *hash\_table\_t*. Caso a alocação de memória não funcione é imprimido uma mensagem de erro e programa termina, caso contrário é atribuído como tamanho inicial da *Hash Table* cem, são inicializados o número de entradas e de arestas da *Hash Table* e é alocado espaço para o *array heads*, novamente com a função *malloc*, e verifica se a operação foi bem-sucedida. Como útlimo passo são inicializados todos os elementos que se encontram no array heads a NULL e termina a função com o retorno da *Hash Table* criada.

### *hash\_table\_free ()*

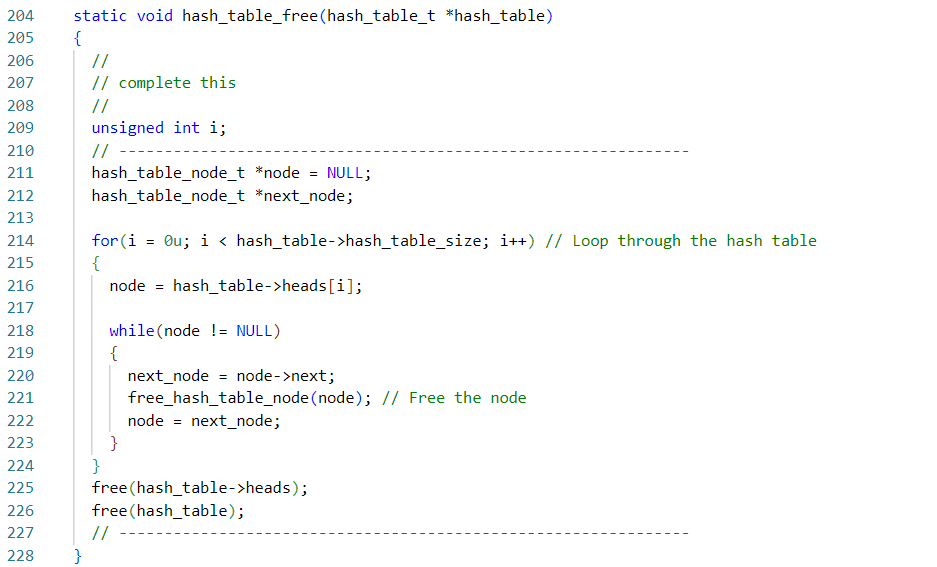
****

Fig.9 – Função *hash\_table\_free ()*

A função *hash\_table\_free ()* tem como objetivo libertar todo o espaço que foi alocado para a *Hash Table* e para os seus elementos.

O primeiro passo desta função é libertar os elementos do *array heads*, ou seja, o *array* que contém os nós da *Hash* *Table*, isto é feito através do ciclo for na linha 214, em que para cada elemento do *array*, é libertado o espaço que está no nó, avança para o próximo nó e termina quando a lista estiver vazia.

Depois de ser libertado o espaço de todos os nós, é também libertado o espaço de memória alocado para o *array heads* e para a estrutura *hash\_table\_t*.

Tal como as funções *free\_hash\_table\_node ()* e *free\_adjency\_node ()*, esta função previne a ocorrência de *memory leaks*.

### *hash\_table\_grow ()*

A função *hash\_table\_grow ()* tem como finalidade de duplicar o tamanho da *Hash Table*, quando chamada. Numa primeira parte, verifica se de facto pode aumentar a *Hash Table* com a condição para averiguar se esta existe, na linha 237. Depois de guardar o antigo tamanho da antiga *Hash Table*, duplica o seu tamanho e aloca um novo *array* de ponteiros para a estrutura *hash\_table\_node\_t*, que será usado para armazenar os cabeçalhos das *Linked Lists* em cada índice na nova e duplicada *Hash Table*.

**Uma imagem com texto

Descrição gerada automaticamente**

Fig.10a – Primeira parte da função *hash\_table\_grow ()*

Numa segunda parte da função, esta inicializa todos os elementos do novo *array* de ponteiros a NULL e itera sobre o *array* antigo de ponteiros e reinsere cada par chave-valor na nova hash table. Para isto, percorre a lista ligada em cada índice antigo do *array* e reinsere cada nó na nova hash table usando a função *crc32()* para determinar o código hash para cada nó. É necessário usar novamente a função *crc32 (*), pois a dimensão da *Hash Table* foi alterada para o dobro.

Por fim, a função liberta com um *free ()* o antigo *array* de ponteiros para as estruturas *hash\_table\_node\_t* e define *hash\_table->heads* como sendo igual ao novo *array*. Adicionalmente, imprime uma mensagem indicando que a *Hash* *Table* cresceu.

Uma imagem com texto

Descrição gerada automaticamente

Fig.10b – Segunda parte da função *hash\_table\_grow ()*

### *find\_word ()*

Nesta função, tal como diz o nome, o objetivo é encontrar uma palavra na *Hash Table*. Esta tabela é passada como argumento junto com a palavra a ser procurada e um valor inteiro designado por *insert\_if\_not\_found*.



Fig.13 – Função *find\_word ()*

Fig.11 – Função hash\_table\_node\_t \*find\_word ()

A função começa por calcular o índice da *Hash Table* onde a palavra deve ser armazenada usando a função *crc32 ()* e o tamanho atual da *Hash Table*. Em seguida, a função percorre cada índice calculado da *Linked List* e verifica se a palavra já está presente na *Hash Table*. Se estiver, a função retorna o nó da *Linked List* que contém a palavra.

Se a palavra não for encontrada e o valor da variável *insert\_if\_not\_found* for um, a função aloca um novo nó da lista ligada. Copia a palavra para o nó e insere-a na *Hash Table*. Se o número de entradas da tabela for maior do que setenta e cinco por cento do tamanho da *Hash Table*, é chamada a função *hash\_table\_grow ()* para duplicar o tamanho da *Hash* *Table*. Se a palavra não for encontrada e o valor da variável *insert\_if\_not\_found* não for um, a função simplesmente retorna NULL.

Em suma, esta função é usada para procurar uma palavra na *Hash Table* e, opcionalmente, inserir a palavra na mesma *Hash Table* se esta ainda não estiver presente nesta. É usada para manter uma *Hash Table* atualizada para garantir que o tamanho desta seja adequado para o número de palavras armazenadas.

### *print\_hash\_table ()*

Nota-se que para uma melhor visualização da *Hash Table* e as suas *Linked Lists*, criámos uma função capaz de imprimir cada nó e, se houver, a sua *Linked List*. Nas imagens abaixo é possível observar a função e um exemplo de um output produzido pela mesma:

Uma imagem com texto

Descrição gerada automaticamente

Fig.12 – Função *print\_hash\_table ()*

Uma imagem com texto

Descrição gerada automaticamente

Fig.12 – Output da função *print\_hash\_table ()* com o ficheiro *wordlist-four-letters.txt*

## **Resultados**

Para este ponto do relatório iremos analisar os resultados que obtivemos para a nossa implementação da *Hash Table*.

Como primeiro teste, quisemos verificar se a nossa função *hash\_table\_grow ()* funcionava, ou seja, quando o número de entradas da Hash Table exceder o tamanho máximo a função *hash\_table\_grow ()* é chamada para aumentar o seu tamanho para o dobro. Por isso corremos compilámos e corremos o nosso programa, passando como argumento o ficheiro *wordlist-four-letters.txt* e obtivemos o seguinte resultado.



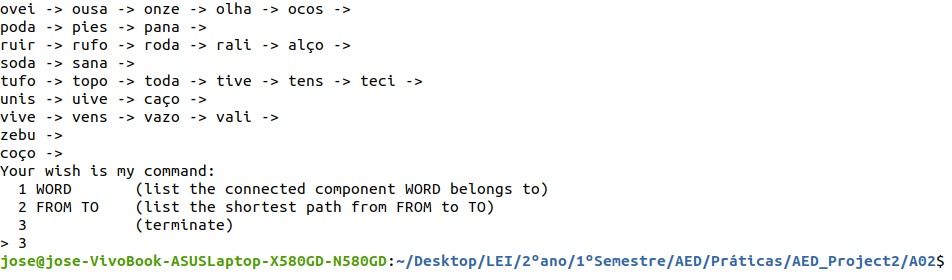


Fig.13 – Verificação da funcionalidade da função *hash\_table\_grow ()*

Ao observarmos o resultado e, sabendo que o ficheiro *wordlist-four-letters.txt* contém duas mil cento e quarenta e nove palavras podemos concluir que a nossa implementação para a função *hash\_table\_grow ()* funciona, pois esta terminou o programa com um tamanho máximo de três mil e duzentos e foi crescendo para o dobro sempre que o número de entradas ultrapasse o tamanho máximo da *Hash Table*.

De seguida decidimos testar se ao executarmos a nossa implementação existiam situações de *memory leaks*, ou seja, se a memória que nós alocámos para os diferentes componentes da *Hash Table* não foi libertada quando á não for preciso utilizá-la mais. Por isso compilámos e corremos a nossa implementação quatro vezes utilizando o *valgrind* e passando como argumentos os ficheiros de texto *wordlist-four-letters.txt*, *wordlist-five-letters.txt*, *wordlist-six-letters.txt* e *wordlist-big-latest.txt* e obtivemos os seguintes resultados:



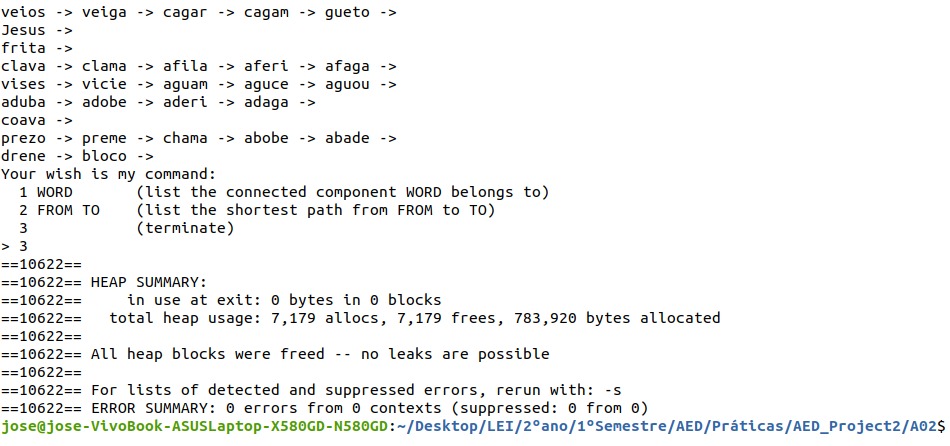
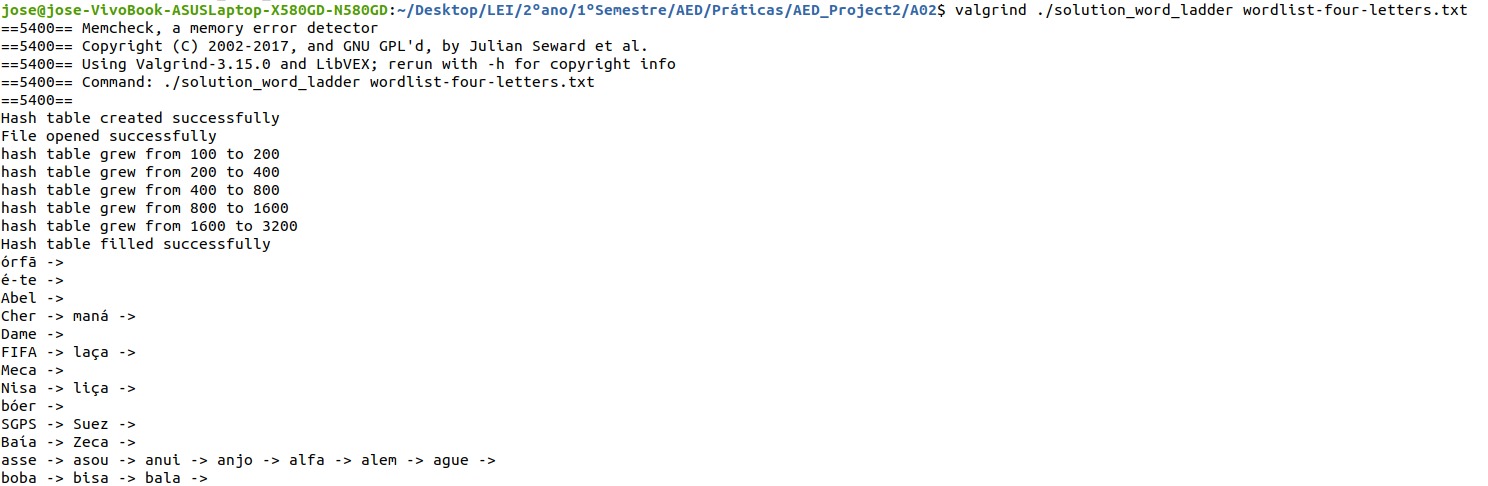


Fig.14 – Verificação da existência de *memory leaks* usando o ficheiro *wordlist-five-letters.txt*



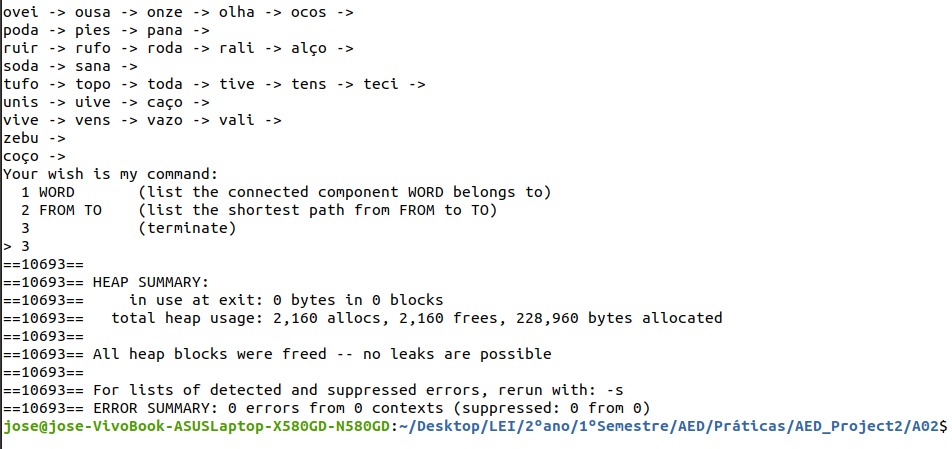


Fig.15 – Verificação da existência de *memory leaks* usando o ficheiro *wordlist-four-letters.txt*



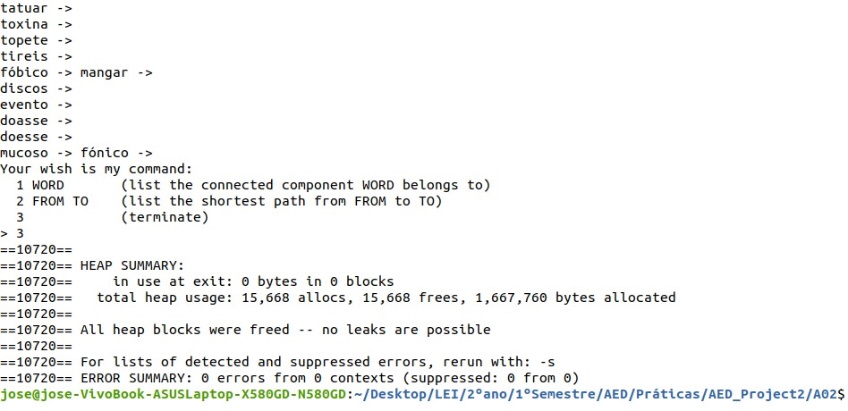
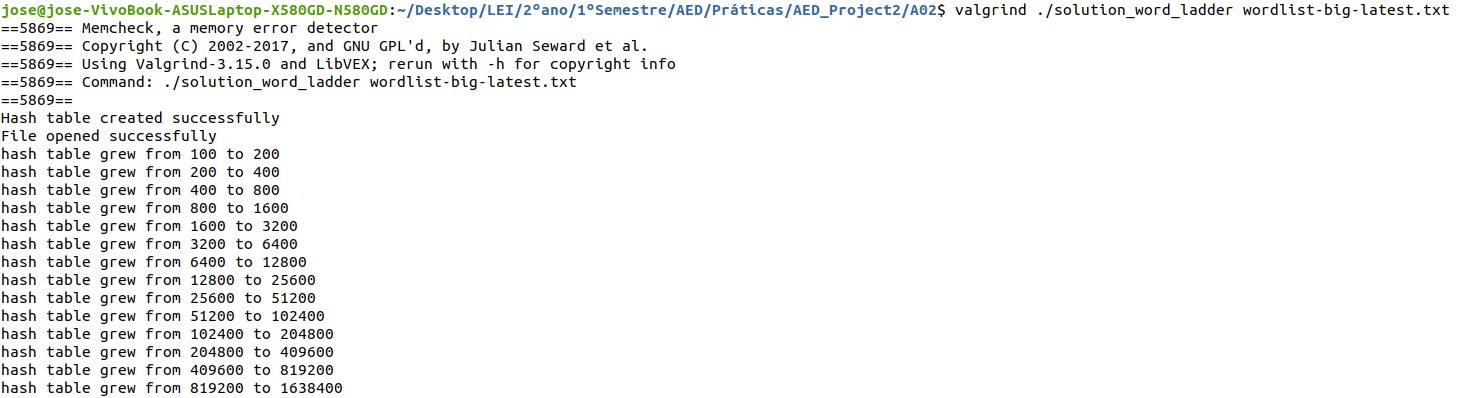


Fig.16 – Verificação da existência de *memory leaks* usando o ficheiro *wordlist-six-letters.txt*



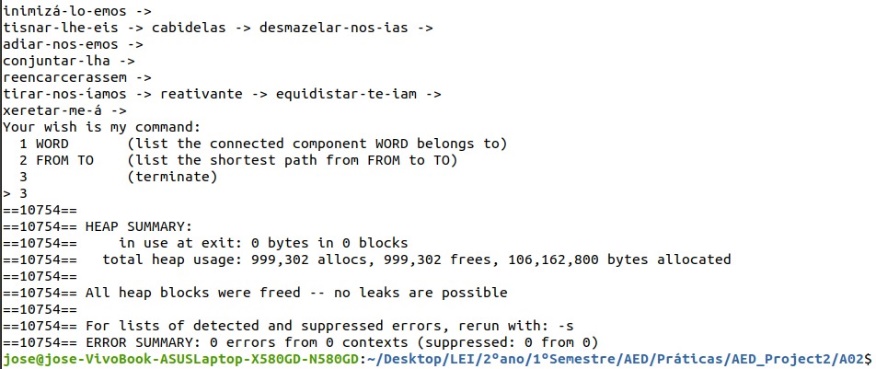


Fig.17 – Verificação da existência de *memory leaks* usando o ficheiro *wordlist-big-latest.txt*

Através destes resultados podemos concluir não existiram *memory leaks*, logo as nossas funções para a implementação de uma *Hash Table* encontram-se corretas.

Também decidimos criar um histograma que apresenta o número de colisões que existem ao corrermos o nosso programa e desenvolvemos um pequeno *script* em *Matlab* para podermos observar como varia o número de colisões que existem ao mudar-mos o número e o tamanho das palavras e obtivemos os seguintes histogramas:

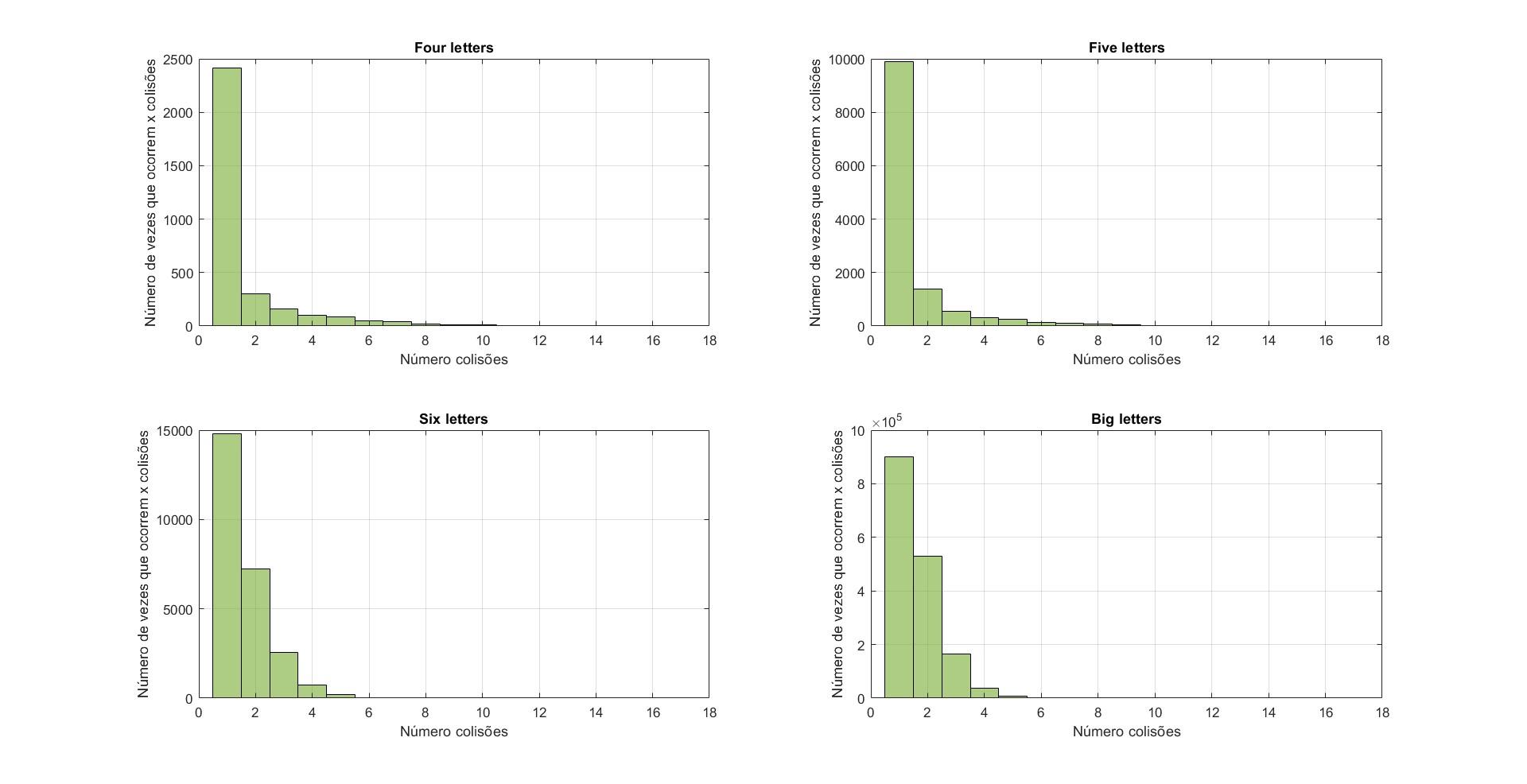


Fig.18 – Histogramas obtidos

Como podemos observar, quando o tamanho é igual a quatro e a quantidade das palavras é menor, não existem muitas colisões, mas ao aumentar-mos esses valores as colisões vão aumentando de forma exponencial.

# ***Código***

## **Código em C**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

//

// static configuration

//

#define \_max\_word\_size\_  32

//

// data structures (SUGGESTION --- you may do it in a different way)

//

typedef struct adjacency\_node\_s  adjacency\_node\_t;

typedef struct hash\_table\_node\_s hash\_table\_node\_t;

typedef struct hash\_table\_s      hash\_table\_t;

struct adjacency\_node\_s

{

  adjacency\_node\_t \*next;            // link to the next adjacency list node

  hash\_table\_node\_t \*vertex;         // the other vertex of the edge (the one that is adjacent to this vertex)

};

struct hash\_table\_node\_s

{

  // the hash table data

  char word[\_max\_word\_size\_];        // the word

  hash\_table\_node\_t \*next;           // next hash table linked list node

  // the vertex data

  adjacency\_node\_t \*head;            // head of the linked list of adjancency edges

  int visited;                       // visited status (while not in use, keep it at 0)

  hash\_table\_node\_t \*previous;       // breadth-first search parent (while not in use, keep it at NULL)

  // the union find data

  hash\_table\_node\_t \*representative; // the representative of the connected component this vertex belongs to

  int number\_of\_vertices;            // number of vertices of the conected component (only correct for the representative of each connected component)

  int number\_of\_edges;               // number of edges of the conected component (only correct for the representative of each connected component)

};

struct hash\_table\_s

{

  unsigned int hash\_table\_size;      // the size of the hash table array

  unsigned int number\_of\_entries;    // the number of entries in the hash table

  unsigned int number\_of\_edges;      // number of edges (for information purposes only)

  hash\_table\_node\_t \*\*heads;         // the heads of the linked lists

};

//

// allocation and deallocation of linked list nodes (done)

//

static adjacency\_node\_t \*allocate\_adjacency\_node(void)

{

  adjacency\_node\_t \*node;

  node = (adjacency\_node\_t \*)malloc(sizeof(adjacency\_node\_t));

  if(node == NULL)

  {

    fprintf(stderr,"allocate\_adjacency\_node: out of memory\n");

    exit(1);

  }

  return node;

}

static void free\_adjacency\_node(adjacency\_node\_t \*node)

{

  free(node);

}

static hash\_table\_node\_t \*allocate\_hash\_table\_node(void)

{

  hash\_table\_node\_t \*node;

  node = (hash\_table\_node\_t \*)malloc(sizeof(hash\_table\_node\_t));

  if(node == NULL)

  {

    fprintf(stderr,"allocate\_hash\_table\_node: out of memory\n");

    exit(1);

  }

  return node;

}

static void free\_hash\_table\_node(hash\_table\_node\_t \*node)

{

  free(node);

}

//

// hash table stuff (mostly to be done)

//

unsigned int crc32(const char \*str)  // CRC-32 (Cyclic Redundancy Check)

{

  static unsigned int table[256];

  unsigned int crc;

  if(table[1] == 0u) // do we need to initialize the table[] array?

  {

    unsigned int i,j;

    for(i = 0u;i < 256u;i++)

      for(table[i] = i,j = 0u;j < 8u;j++)

        if(table[i] & 1u)

          table[i] = (table[i] >> 1) ^ 0xAED00022u; // "magic" constant

        else

          table[i] >>= 1;

  }

  crc = 0xAED02022u; // initial value (chosen arbitrarily)

  while(\*str != '\0')

    crc = (crc >> 8) ^ table[crc & 0xFFu] ^ ((unsigned int)\*str++ << 24);

  return crc;

}

static hash\_table\_t \*hash\_table\_create(void)

{

  hash\_table\_t \*hash\_table;

  unsigned int i;

  hash\_table = (hash\_table\_t \*)malloc(sizeof(hash\_table\_t));

  if(hash\_table == NULL)

  {

    fprintf(stderr,"create\_hash\_table: out of memory\n");

    exit(1);

  }

  //

  // complete this

  //

  // ---------------------------------------------------------------

  hash\_table->hash\_table\_size = 100;

  hash\_table->number\_of\_entries = 0;

  hash\_table->number\_of\_edges = 0;

  hash\_table->heads = (hash\_table\_node\_t \*\*)malloc(hash\_table->hash\_table\_size \* sizeof(hash\_table\_node\_t \*)); // Allocate memory for the heads of the hash table

  if(hash\_table->heads == NULL)

  {

    fprintf(stderr,"create\_hash\_table: out of memory\n");

    exit(1);

  }

  for(i = 0u; i < hash\_table->hash\_table\_size; i++) // Set all the heads to NULL

  {

    hash\_table->heads[i] = NULL;

  }

  // ---------------------------------------------------------------

  return hash\_table;

}

//

// complete this

//

static void hash\_table\_free(hash\_table\_t \*hash\_table)

{

  //

  // complete this

  //

  unsigned int i;

  // ---------------------------------------------------------------

  hash\_table\_node\_t \*node = NULL;

  hash\_table\_node\_t \*next\_node;

  for(i = 0u; i < hash\_table->hash\_table\_size; i++) // Loop through the hash table

  {

    node = hash\_table->heads[i];

    while(node != NULL)

    {

      next\_node = node->next;

      free\_hash\_table\_node(node); // Free the node

      node = next\_node;

    }

  }

  free(hash\_table->heads);

  free(hash\_table);

  // ---------------------------------------------------------------

}

static void hash\_table\_grow(hash\_table\_t \*hash\_table)

{

  //

  // complete this

  //

  // ---------------------------------------------------------------

  if (hash\_table == NULL) { // Verify if the hash table is null

    fprintf(stderr, "Error: Cannot grow null hash table.\n");

    return;

  }

  hash\_table\_node\_t \*next\_node, \*first\_node, \*node;

  unsigned int i, old\_size;

  old\_size = hash\_table->hash\_table\_size;

  hash\_table->hash\_table\_size = hash\_table->hash\_table\_size \* 2u; // Double the size of the hash table

  hash\_table\_node\_t \*\*new\_heads = (hash\_table\_node\_t \*\*)malloc(hash\_table->hash\_table\_size \* sizeof(hash\_table\_node\_t \*));

  for(i = 0u; i < hash\_table->hash\_table\_size; i++) // Set all the heads to NULL

  {

    new\_heads[i] = NULL;

  }

  for(i = 0u ; i < old\_size ; i++)

  {

    node = hash\_table->heads[i];

    while (node != NULL)

    {

      next\_node = node->next;

      node->next = NULL; // Set the next node to NULL

      size\_t new\_index = crc32(node->word) % hash\_table->hash\_table\_size; // Get the new index

      if(new\_heads[new\_index] == NULL) // If the new index is empty

      {

        new\_heads[new\_index] = node;

      }

      else // If the new index is not empty

      {

        first\_node = new\_heads[new\_index];

        node->next = first\_node;

        new\_heads[new\_index] = node;

      }

      node = next\_node;

    }

  }

  free(hash\_table->heads); // Free the old heads

  hash\_table->heads = new\_heads;

  printf("hash table grew from %d to %d\n",old\_size,hash\_table->hash\_table\_size);

  // ---------------------------------------------------------------

}

static hash\_table\_node\_t \*find\_word(hash\_table\_t \*hash\_table,const char \*word,int insert\_if\_not\_found)

{

  hash\_table\_node\_t \*node, \*new\_node;

  unsigned int i;

  i = crc32(word) % hash\_table->hash\_table\_size;

  //

  // complete this

  //

  // ---------------------------------------------------------------

  node = hash\_table->heads[i];

  while(node != NULL)

  {

    if(strcmp(node->word, word) == 0) // If the word is found

    {

        return node;

    }

    node = node->next;

  }

  if(insert\_if\_not\_found == 1) // If the word is not found

  {

    // if the word size is greater than the maximum word size

    if(strlen(word) > \_max\_word\_size\_)

    {

      fprintf(stderr," the word size %s is greater than the maximum word size\n",word);

      exit(1);

    }

    node = allocate\_hash\_table\_node();  // Allocate a new node

    strncpy(node->word,word,\_max\_word\_size\_);  // Copy the word into the node

    node->next = NULL;

    if (hash\_table->heads[i] == NULL)

    {

      hash\_table->heads[i] = node;

      hash\_table->number\_of\_entries++;

    }

    else

    {

      node->next = hash\_table->heads[i]; // Set the next node to the new node

      hash\_table->heads[i] = node;

      hash\_table->number\_of\_entries++; // Increment the number of entries in the hash table

    }

    if(hash\_table->number\_of\_entries > 0.75 \* hash\_table->hash\_table\_size) // If the number of words is greater than the size of the hash table

    {

      hash\_table\_grow(hash\_table);

    }

    return node;

  }

  // ---------------------------------------------------------------

  return NULL;

}

//

// add edges to the word ladder graph (mostly do be done)

//

static hash\_table\_node\_t \*find\_representative(hash\_table\_node\_t \*node)

{

  hash\_table\_node\_t \*representative,\*next\_node;

  //

  // complete this

  //

}

static void add\_edge(hash\_table\_t \*hash\_table,hash\_table\_node\_t \*from,const char \*word) // Add an edge to the graph

{

  hash\_table\_node\_t \*to,\*from\_representative,\*to\_representative;

  adjacency\_node\_t \*link;

  //

  // complete this

  //

}

//

// generates a list of similar words and calls the function add\_edge for each one (done)

//

// man utf8 for details on the uft8 encoding

//

static void break\_utf8\_string(const char \*word,int \*individual\_characters)

{

  int byte0,byte1;

  while(\*word != '\0')

  {

    byte0 = (int)(\*(word++)) & 0xFF;

    if(byte0 < 0x80)

      \*(individual\_characters++) = byte0; // plain ASCII character

    else

    {

      byte1 = (int)(\*(word++)) & 0xFF;

      if((byte0 & 0b11100000) != 0b11000000 || (byte1 & 0b11000000) != 0b10000000)

      {

        fprintf(stderr,"break\_utf8\_string: unexpected UFT-8 character\n");

        exit(1);

      }

      \*(individual\_characters++) = ((byte0 & 0b00011111) << 6) | (byte1 & 0b00111111); // utf8 -> unicode

    }

  }

  \*individual\_characters = 0; // mark the end!

}

static void make\_utf8\_string(const int \*individual\_characters,char word[\_max\_word\_size\_])

{

  int code;

  while(\*individual\_characters != 0)

  {

    code = \*(individual\_characters++);

    if(code < 0x80)

      \*(word++) = (char)code;

    else if(code < (1 << 11))

    { // unicode -> utf8

      \*(word++) = 0b11000000 | (code >> 6);

      \*(word++) = 0b10000000 | (code & 0b00111111);

    }

    else

    {

      fprintf(stderr,"make\_utf8\_string: unexpected UFT-8 character\n");

      exit(1);

    }

  }

  \*word = '\0';  // mark the end

}

static void similar\_words(hash\_table\_t \*hash\_table,hash\_table\_node\_t \*from)

{

  static const int valid\_characters[] =

  { // unicode!

    0x2D,                                                                       // -

    0x41,0x42,0x43,0x44,0x45,0x46,0x47,0x48,0x49,0x4A,0x4B,0x4C,0x4D,           // A B C D E F G H I J K L M

    0x4E,0x4F,0x50,0x51,0x52,0x53,0x54,0x55,0x56,0x57,0x58,0x59,0x5A,           // N O P Q R S T U V W X Y Z

    0x61,0x62,0x63,0x64,0x65,0x66,0x67,0x68,0x69,0x6A,0x6B,0x6C,0x6D,           // a b c d e f g h i j k l m

    0x6E,0x6F,0x70,0x71,0x72,0x73,0x74,0x75,0x76,0x77,0x78,0x79,0x7A,           // n o p q r s t u v w x y z

    0xC1,0xC2,0xC9,0xCD,0xD3,0xDA,                                              // Á Â É Í Ó Ú

    0xE0,0xE1,0xE2,0xE3,0xE7,0xE8,0xE9,0xEA,0xED,0xEE,0xF3,0xF4,0xF5,0xFA,0xFC, // à á â ã ç è é ê í î ó ô õ ú ü

    0

  };

  int i,j,k,individual\_characters[\_max\_word\_size\_];

  char new\_word[2 \* \_max\_word\_size\_];

  break\_utf8\_string(from->word,individual\_characters);

  for(i = 0;individual\_characters[i] != 0;i++)

  {

    k = individual\_characters[i];

    for(j = 0;valid\_characters[j] != 0;j++)

    {

      individual\_characters[i] = valid\_characters[j];

      make\_utf8\_string(individual\_characters,new\_word);

      // avoid duplicate cases

      if(strcmp(new\_word,from->word) > 0)

        add\_edge(hash\_table,from,new\_word);

    }

    individual\_characters[i] = k;

  }

}

//

// breadth-first search (to be done)

//

// returns the number of vertices visited; if the last one is goal, following the previous links gives the shortest path between goal and origin

//

static int breadh\_first\_search(int maximum\_number\_of\_vertices,hash\_table\_node\_t \*\*list\_of\_vertices,hash\_table\_node\_t \*origin,hash\_table\_node\_t \*goal)

{

  //

  // complete this

  //

  return -1;

}

// list all vertices belonging to a connected component (complete this)

//

static void list\_connected\_component(hash\_table\_t \*hash\_table,const char \*word)

{

  //

  // complete this

  //

}

//

// compute the diameter of a connected component (optional)

//

static int largest\_diameter;

static hash\_table\_node\_t \*\*largest\_diameter\_example;

static int connected\_component\_diameter(hash\_table\_node\_t \*node)

{

  int diameter;

  //

  // complete this

  //

  return diameter;

}

//

// find the shortest path from a given word to another given word (to be done)

//

static void path\_finder(hash\_table\_t \*hash\_table,const char \*from\_word,const char \*to\_word)

{

  //

  // complete this

  //

}

//

// some graph information (optional)

//

static void graph\_info(hash\_table\_t \*hash\_table)

{

  //

  // complete this

  //

}

//print hash\_table with the enumeration of all linked list nodes

static void print\_hash\_table(hash\_table\_t \*hash\_table)

{

  hash\_table\_node\_t \*node = NULL;

  unsigned int i;

  for(i = 0u;i < hash\_table->hash\_table\_size;i++)

  {

    node = hash\_table->heads[i];

    while(node != NULL)

    {

      printf("%s -> ",node->word);

      node = node->next;

    }

    if (hash\_table->heads[i] != NULL)

    {

      printf("\n");

    }

  }

}

//

// main program

//

int main(int argc,char \*\*argv)

{

  char word[100],from[100],to[100];

  hash\_table\_t \*hash\_table;

  hash\_table\_node\_t \*node = NULL;

  unsigned int i = 0u;

  int command;

  FILE \*fp;

  // initialize hash table

  hash\_table = hash\_table\_create();

  printf("Hash table created successfully\n");

  // read words

  fp = fopen((argc < 2) ? "wordlist-big-latest.txt" : argv[1],"rb");

  if(fp == NULL)

  {

    fprintf(stderr,"main: unable to open the words file\n");

    exit(1);

  }

  printf("File opened successfully\n");

  while(fscanf(fp,"%99s",word) == 1)

    (void)find\_word(hash\_table,word,1);

  fclose(fp);

  printf("Hash table filled successfully\n");

  print\_hash\_table(hash\_table);

  // find all similar words

  for(i = 0u;i < hash\_table->hash\_table\_size;i++)

    for(node = hash\_table->heads[i];node != NULL;node = node->next)

      similar\_words(hash\_table,node);

  graph\_info(hash\_table);

  // ask what to do

  for(;;)

  {

    fprintf(stderr,"Your wish is my command:\n");

    fprintf(stderr,"  1 WORD       (list the connected component WORD belongs to)\n");

    fprintf(stderr,"  2 FROM TO    (list the shortest path from FROM to TO)\n");

    fprintf(stderr,"  3            (terminate)\n");

    fprintf(stderr,"> ");

    if(scanf("%99s",word) != 1)

      break;

    command = atoi(word);

    if(command == 1)

    {

      if(scanf("%99s",word) != 1)

        break;

      list\_connected\_component(hash\_table,word);

    }

    else if(command == 2)

    {

      if(scanf("%99s",from) != 1)

        break;

      if(scanf("%99s",to) != 1)

        break;

      path\_finder(hash\_table,from,to);

    }

    else if(command == 3)

      break;

  }

  // clean up

  hash\_table\_free(hash\_table);

  return 0;

}

## **Código em Matlab**

clear

clc

four = load("four\_letters.txt");

five = load("five\_letters.txt");

six = load("six\_letters.txt");

big = load("big\_letters.txt");

figure(1)

subplot(2,2,1)

histogram(four, 'FaceColor', [0.4660 0.6740 0.1880]);

xlim([0 18])

title("Four letters")

xlabel("Número colisões")

ylabel("Número de vezes que ocorrem x colisões")

grid on

subplot(2,2,2)

histogram(five, 'FaceColor', [0.4660 0.6740 0.1880]);

xlim([0 18])

title("Five letters")

xlabel("Número colisões")

ylabel("Número de vezes que ocorrem x colisões")

grid on

subplot(2,2,3)

histogram(six, 'FaceColor', [0.4660 0.6740 0.1880]);

xlim([0 18])

title("Six letters")

xlabel("Número colisões")

ylabel("Número de vezes que ocorrem x colisões")

grid on

subplot(2,2,4)

histogram(big, 'FaceColor', [0.4660 0.6740 0.1880]);

xlim([0 18])

title("Big letters")

xlabel("Número colisões")

ylabel("Número de vezes que ocorrem x colisões")

grid on

# ***Conclusão***

Com a realização deste trabalho concluímos que existem várias maneiras de abordar o problema proposto. Estas soluções utilizam métodos recursivos e iterativos. Constatamos que os métodos iterativos são mais rápidos e eficientes que os recursivos através da análise de gráficos de execução e dos ficheiros gerados pelo programa.

Observámos que, na criação de uma solução usando programação dinâmica, esta foi a melhor solução até à data visto que o gráfico do seu tempo de execução e o esforço obtido foram os menores.