universidade de aveiro deti departamento de electrónica, telecomunicações e informática

Trabalho Prático 2

Word Ladder

Licenciatura em Engenharia Informática

Relatório

Turma P8

Professores:

Joaquim Madeira (*jmadeira@ua.pt*)

Tomás Oliveira da Silva (*tos@ua.pt*)

Realizado por:

Diogo Falcão Nº108712 – **50%**

José Gameiro Nº108840 – **50%**

08/01/2023

Índice:

1. 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.

2. 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.

2.1. Estruturas

```
60    typedef struct adjacency_node_s  adjacency_node_t;
61    typedef struct hash_table_node_s hash_table_node_t;
62    typedef struct hash_table_s      hash_table_t;
```

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.***

```
64  struct adjacency_node_s
65  {
66      adjacency_node_t *next;          // link to the next adjacency list node
67      hash_table_node_t *vertex;       // the other vertex of the edge (the one that is adjacent to this vertex)
68  };
```

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.

```
70  struct hash_table_node_s
71  {
72    // the hash table data
73    char word[_max_word_size_];        // the word
74    hash_table_node_t *next;           // next hash table linked list node
75    // the vertex data
76    adjacency_node_t *head;            // head of the linked list of adjancency edges
77    int visited;                       // visited status (while not in use, keep it at 0)
78    hash_table_node_t *previous;       // breadth-first search parent (while not in use, keep it at NULL)
79    // the union find data
80    hash_table_node_t *representative; // the representative of the connected component this vertex belongs to
81    int number_of_vertices;            // number of vertices of the conected component (only correct for the representative of each connected component)
82    int number_of_edges;               // number of edges of the conected component (only correct for the representative of each connected component)
83  };
```

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.

```
85  struct hash_table_s
86  {
87    unsigned int hash_table_size;      // the size of the hash table array
88    unsigned int number_of_entries;    // the number of entries in the hash table
89    unsigned int number_of_edges;      // number of edges (for information purposes only)
90    hash_table_node_t **heads;         // the heads of the linked lists
91  };
```

Fig.4 – Estrutura *hash_table_s*

2.2. Funções

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

2.2.1. allocate_adjency_node ()

```
98   static adjacency_node_t *allocate_adjacency_node(void)
99   {
100    adjacency_node_t *node;
101
102    node = (adjacency_node_t *)malloc(sizeof(adjacency_node_t));
103
104    if(node == NULL)
105    {
106      fprintf(stderr,"allocate_adjacency_node: out of memory\n");
107      exit(1);
108    }
109    return node;
110  }
```

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.

2.2.2. free_adjency_node ()

```
112  static void free_adjacency_node(adjacency_node_t *node)
113  {
114    free(node);
115  }
```

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.

2.2.3. allocate_hash_table_node ()

```
117  static hash_table_node_t *allocate_hash_table_node(void)
118  {
119      hash_table_node_t *node;
120
121      node = (hash_table_node_t *)malloc(sizeof(hash_table_node_t));
122
123      if(node == NULL)
124      {
125          fprintf(stderr,"allocate_hash_table_node: out of memory\n");
126          exit(1);
127      }
128      return node;
129  }
```

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*.

2.2.4. free_hash_table_node ()

```
131  static void free_hash_table_node(hash_table_node_t *node)
132  {
133      free(node);
134  }
```

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*.

2.2.5. hash_table_create ()

```
163  static hash_table_t *hash_table_create(void)
164  {
165    hash_table_t *hash_table;
166    unsigned int i;
167
168    hash_table = (hash_table_t *)malloc(sizeof(hash_table_t));
169
170    if(hash_table == NULL)
171    {
172      fprintf(stderr,"create_hash_table: out of memory\n");
173      exit(1);
174    }
175    //
176    // complete this
177    //
178
179    // -----------------------------------------------------------
180    hash_table->hash_table_size = 100;
181    hash_table->number_of_entries = 0;
182    hash_table->number_of_edges = 0;
183    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
184
185    if(hash_table->heads == NULL)
186    {
187      fprintf(stderr,"create_hash_table: out of memory\n");
188      exit(1);
189    }
190
191    for(i = 0u; i < hash_table->hash_table_size; i++) // Set all the heads to NULL
192    {
193      hash_table->heads[i] = NULL;
194    }
195    // -----------------------------------------------------------
196
197    return hash_table;
198  }
```

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.

2.2.6. hash_table_free ()

```
204  static void hash_table_free(hash_table_t *hash_table)
205  {
206    //
207    // complete this
208    //
209    unsigned int i;
210    // -----------------------------------------------------------------
211    hash_table_node_t *node = NULL;
212    hash_table_node_t *next_node;
213
214    for(i = 0u; i < hash_table->hash_table_size; i++) // Loop through the hash table
215    {
216      node = hash_table->heads[i];
217
218      while(node != NULL)
219      {
220        next_node = node->next;
221        free_hash_table_node(node); // Free the node
222        node = next_node;
223      }
224    }
225    free(hash_table->heads);
226    free(hash_table);
227    // -----------------------------------------------------------------
228  }
```

Fig.10 – 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*.

2.2.7. 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*.

```
230  static void hash_table_grow(hash_table_t *hash_table)
231  {
232    //
233    // complete this
234    //
235
236    // -------------------------------------------------------------
237    if (hash_table == NULL) { // Verify if the hash table is null
238      fprintf(stderr, "Error: Cannot grow null hash table.\n");
239      return;
240    }
241
242    hash_table_node_t *next_node, *first_node, *node;
243    unsigned int i, old_size;
244
245    old_size = hash_table->hash_table_size;
246    hash_table->hash_table_size = hash_table->hash_table_size * 2u; // Double the size of the hash table
247
248
249    hash_table_node_t **new_heads = (hash_table_node_t **)malloc(hash_table->hash_table_size * sizeof(hash_table_node_t *));
250
251    for(i = 0u; i < hash_table->hash_table_size; i++) // Set all the heads to NULL
252    {
253      new_heads[i] = NULL;
254    }
255
```

Fig.11a – 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.

```
256     for(i = 0u ; i < old_size ; i++)
257     {
258       node = hash_table->heads[i];
259       while (node != NULL)
260       {
261         next_node = node->next;
262         node->next = NULL; // Set the next node to NULL
263         size_t new_index = crc32(node->word) % hash_table->hash_table_size; // Get the new index
264 
265         if(new_heads[new_index] == NULL) // If the new index is empty
266         {
267           new_heads[new_index] = node;
268         }
269         else // If the new index is not empty
270         {
271           first_node = new_heads[new_index];
272           node->next = first_node;
273           new_heads[new_index] = node;
274         }
275         node = next_node;
276       }
277     }
278     free(hash_table->heads); // Free the old heads
279     hash_table->heads = new_heads;
280     printf("hash table grew from %d to %d\n",old_size,hash_table->hash_table_size);
281     // -------------------------------------------------------------
282   }
```

Fig.11b – Segunda parte da função *hash_table_grow ()*

2.2.8. 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*.

```c
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);
    }
    printf("%d\n",hash_table->number_of_entries);
    return node;
  }
  // -------------------------------------------------------------
  return NULL;
}
```

Fig.12 – Função *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.

2.2.9. 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:

```
524   //print hash_table with the enumeration of all linked list nodes
525   static void print_hash_table(hash_table_t *hash_table)
526   {
527     hash_table_node_t *node = NULL;
528     unsigned int i;
529
530     for(i = 0u;i < hash_table->hash_table_size;i++)
531     {
532       node = hash_table->heads[i];
533       while(node != NULL)
534       {
535         printf("%s -> ",node->word);
536         node = node->next;
537       }
538       if (hash_table->heads[i] != NULL)
539       {
540         printf("\n");
541       }
542     }
543   }
```

Fig.13 – Função *print_hash_table ()*

```
cebo -> cios -> cozo ->
dona ->
enol -> ermo ->
fiem -> fios -> firo ->
gemi -> giro -> gozo ->
deão ->
leoa -> lona ->
muda -> mona -> miro -> miem ->
nona ->
ovas -> opia -> oiro -> obus ->
poli -> piro -> piem ->
ruís ->
rubi -> rios -> rifa -> riem -> remi -> alça ->
subi -> sebo ->
tona -> tiro -> tios -> temi -> taxa ->
uvas -> urbe -> unja ->
viro ->
zona ->
você ->
ação ->
Aida ->
Caio -> Cuba ->
Gaio -> Guam ->
Juno -> Java ->
Nuno ->
Paio -> Perl ->
Thor ->
Zola ->
móis -> asai -> amue -> adia -> aipo -> ajas -> galé ->
bola -> bati ->
cava -> caio -> cola -> come -> coxo -> cuba ->
dava -> dome ->
ecoe ->
fava -> fome ->
gema -> gene -> gola -> gome ->
huno ->
iene -> isca ->
juba ->
lati -> lava -> lema -> lida ->
muno -> mola -> miau -> maio ->
nome ->
puno -> pneu -> pipo -> pene -> paio ->
ruam -> roxo -> rola -> ripo -> rida -> rema -> raio ->
suba -> suam -> some -> sola -> sida -> saio ->
tuba -> tome -> tola -> tipo -> tida -> tema ->
usai -> unto -> unha ->
vida -> vaio ->
júri ->
terá ->
```

Fig.14 – Output da função *print_hash_table ()* com o ficheiro *wordlist-four-letters.txt*

2.3. 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.

```
jose@jose-VivoBook-ASUSLaptop-X580GD-N580GD:~/Desktop/LEI/2ºano/1ºSemestre/AED/Práticas/AED_Project2/A02$ ./solution_word_ladder wordlist-four-letters.txt
Hash table created successfully
File opened successfully
hash table grew from 100 to 200
hash table grew from 200 to 400
hash table grew from 400 to 800
hash table grew from 800 to 1600
hash table grew from 1600 to 3200
Hash table filled successfully
órfã ->
é-te ->
Abel ->
Cher -> maná ->
Dame ->
FIFA -> laça ->
Meca ->
Nisa -> liça ->
bóer ->
SGPS -> Suez ->
Baia -> Zeca ->
asse -> asou -> anui -> anjo -> alfa -> alem -> ague ->
boba -> bisa -> bala ->
cede -> ceai -> cala -> caiu -> cisa -> cite -> coam -> coce -> cure ->
dite -> doam -> doba -> doce -> dono -> duas -> dure -> fará -> lê-o ->
elos -> esse ->
divã -> fala -> fede -> fite -> fure ->
pivô -> gala -> game -> geai ->
ovei -> ousa -> onze -> olha -> ocos ->
poda -> pies -> pana ->
ruir -> rufo -> roda -> rali -> alço ->
soda -> sana ->
tufo -> topo -> toda -> tive -> tens -> teci ->
unis -> uive -> caço ->
vive -> vens -> vazo -> vali ->
zebu ->
coço ->
Your wish is my command:
  1 WORD       (list the connected component WORD belongs to)
  2 FROM TO    (list the shortest path from FROM to TO)
  3            (terminate)
> 3
jose@jose-VivoBook-ASUSLaptop-X580GD-N580GD:~/Desktop/LEI/2ºano/1ºSemestre/AED/Práticas/AED_Project2/A02$
```

Fig.15 – 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:

```
jose@jose-VivoBook-ASUSLaptop-X580GD-N580GD:~/Desktop/LEI/2ºano/1ºSemestre/AED/Práticas/AED_Project2/A02$ valgrind ./solution_word_ladder wordlist-five-letters.txt
==10622== Memcheck, a memory error detector
==10622== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==10622== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==10622== Command: ./solution_word_ladder wordlist-five-letters.txt
==10622==
Hash table created successfully
File opened successfully
hash table grew from 100 to 200
hash table grew from 200 to 400
hash table grew from 400 to 800
hash table grew from 800 to 1600
hash table grew from 1600 to 3200
hash table grew from 3200 to 6400
hash table grew from 6400 to 12800
Hash table filled successfully
ótimo ->
Óscar ->
égide ->
êxito -> chiça ->
choça ->
xexés ->
veios -> veiga -> cagar -> cagam -> gueto ->
Jesus ->
frita ->
clava -> clama -> afila -> aferi -> afaga ->
vises -> vicie -> aguam -> aguce -> aguou ->
aduba -> adobe -> aderi -> adaga ->
coava ->
prezo -> preme -> chama -> abobe -> abade ->
drene -> bloco ->
Your wish is my command:
  1 WORD       (list the connected component WORD belongs to)
  2 FROM TO    (list the shortest path from FROM to TO)
  3            (terminate)
> 3
==10622==
==10622== HEAP SUMMARY:
==10622==     in use at exit: 0 bytes in 0 blocks
==10622==   total heap usage: 7,179 allocs, 7,179 frees, 783,920 bytes allocated
==10622==
==10622== All heap blocks were freed -- no leaks are possible
==10622==
==10622== For lists of detected and suppressed errors, rerun with: -s
==10622== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
jose@jose-VivoBook-ASUSLaptop-X580GD-N580GD:~/Desktop/LEI/2ºano/1ºSemestre/AED/Práticas/AED_Project2/A02$
```

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

```
jose@jose-VivoBook-ASUSLaptop-X580GD-N580GD:~/Desktop/LEI/2ºano/1ºSemestre/AED/Práticas/AED_Project2/A02$ valgrind ./solution_word_ladder wordlist-four-letters.txt
==5400== Memcheck, a memory error detector
==5400== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==5400== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==5400== Command: ./solution_word_ladder wordlist-four-letters.txt
==5400==
Hash table created successfully
File opened successfully
hash table grew from 100 to 200
hash table grew from 200 to 400
hash table grew from 400 to 800
hash table grew from 800 to 1600
hash table grew from 1600 to 3200
Hash table filled successfully
órfã ->
é-te ->
Abel ->
Cher -> maná ->
Dame ->
FIFA -> laça ->
Meca ->
Nisa -> liça ->
bóer ->
SGPS -> Suez ->
Baia -> Zeca ->
asse -> asou -> anui -> anjo -> alfa -> alem -> ague ->
boba -> bisa -> bala ->
ovei -> ousa -> onze -> olha -> ocos ->
poda -> pies -> pana ->
ruir -> rufo -> roda -> rali -> alço ->
soda -> sana ->
tufo -> topo -> toda -> tive -> tens -> teci ->
unis -> uive -> caço ->
vive -> vens -> vazo -> vali ->
zebu ->
coço ->
Your wish is my command:
  1 WORD       (list the connected component WORD belongs to)
  2 FROM TO    (list the shortest path from FROM to TO)
  3            (terminate)
> 3
==10693==
==10693== HEAP SUMMARY:
==10693==     in use at exit: 0 bytes in 0 blocks
==10693==   total heap usage: 2,160 allocs, 2,160 frees, 228,960 bytes allocated
==10693==
==10693== All heap blocks were freed -- no leaks are possible
==10693==
==10693== For lists of detected and suppressed errors, rerun with: -s
==10693== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
jose@jose-VivoBook-ASUSLaptop-X580GD-N580GD:~/Desktop/LEI/2ºano/1ºSemestre/AED/Práticas/AED_Project2/A02$
```

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

```
jose@jose-VivoBook-ASUSLaptop-X580GD-N580GD:~/Desktop/LEI/2ºano/1ºSemestre/AED/Práticas/AED_Project2/A02$ valgrind ./solution_word_ladder wordlist-six-letters.txt
==5656== Memcheck, a memory error detector
==5656== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==5656== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==5656== Command: ./solution_word_ladder wordlist-six-letters.txt
==5656==
Hash table created successfully
File opened successfully
hash table grew from 100 to 200
hash table grew from 200 to 400
hash table grew from 400 to 800
hash table grew from 800 to 1600
hash table grew from 1600 to 3200
hash table grew from 3200 to 6400
hash table grew from 6400 to 12800
hash table grew from 12800 to 25600
Hash table filled successfully
borrai ->
segura ->
rilhes ->
secura ->
rolham -> imporá ->
rompem -> cancro ->
sumido ->
sanita ->
subido -> sacuda ->
blinde ->
álcool -> sismos ->
ralhas ->
zunido -> situam ->
colore ->
Balzac ->
berros -> berrem -> berrar -> ir-lhe ->
rachas ->
tatuar ->
toxina ->
topete ->
tireis ->
fóbico -> mangar ->
discos ->
evento ->
doasse ->
doesse ->
mucoso -> fónico ->
Your wish is my command:
  1 WORD       (list the connected component WORD belongs to)
  2 FROM TO    (list the shortest path from FROM to TO)
  3            (terminate)
> 3
==10720==
==10720== HEAP SUMMARY:
==10720==     in use at exit: 0 bytes in 0 blocks
==10720==   total heap usage: 15,668 allocs, 15,668 frees, 1,667,760 bytes allocated
==10720==
==10720== All heap blocks were freed -- no leaks are possible
==10720==
==10720== For lists of detected and suppressed errors, rerun with: -s
==10720== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
jose@jose-VivoBook-ASUSLaptop-X580GD-N580GD:~/Desktop/LEI/2ºano/1ºSemestre/AED/Práticas/AED_Project2/A02$
```

Fig.18 – Verificação da existência de *memory leaks* usando o ficheiro *wordlist-six-letters.txt*

```
jose@jose-VivoBook-ASUSLaptop-X580GD-N580GD:~/Desktop/LEI/2ºano/1ºSemestre/AED/Práticas/AED_Project2/A02$ valgrind ./solution_word_ladder wordlist-big-latest.txt
==5869== Memcheck, a memory error detector
==5869== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==5869== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==5869== Command: ./solution_word_ladder wordlist-big-latest.txt
==5869==
Hash table created successfully
File opened successfully
hash table grew from 100 to 200
hash table grew from 200 to 400
hash table grew from 400 to 800
hash table grew from 800 to 1600
hash table grew from 1600 to 3200
hash table grew from 3200 to 6400
hash table grew from 6400 to 12800
hash table grew from 12800 to 25600
hash table grew from 25600 to 51200
hash table grew from 51200 to 102400
hash table grew from 102400 to 204800
hash table grew from 204800 to 409600
hash table grew from 409600 to 819200
hash table grew from 819200 to 1638400
inimizá-lo-emos ->
tisnar-lhe-eis -> cabidelas -> desmazelar-nos-ias ->
adiar-nos-emos ->
conjuntar-lha ->
reencarcerassem ->
tirar-nos-iamos -> reativante -> equidistar-te-iam ->
xeretar-me-á ->
Your wish is my command:
  1 WORD       (list the connected component WORD belongs to)
  2 FROM TO    (list the shortest path from FROM to TO)
  3            (terminate)
> 3
==10754==
==10754== HEAP SUMMARY:
==10754==     in use at exit: 0 bytes in 0 blocks
==10754==   total heap usage: 999,302 allocs, 999,302 frees, 106,162,800 bytes allocated
==10754==
==10754== All heap blocks were freed -- no leaks are possible
==10754==
==10754== For lists of detected and suppressed errors, rerun with: -s
==10754== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
jose@jose-VivoBook-ASUSLaptop-X580GD-N580GD:~/Desktop/LEI/2ºano/1ºSemestre/AED/Práticas/AED_Project2/A02$
```

Fig.19 – 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 correrm 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:

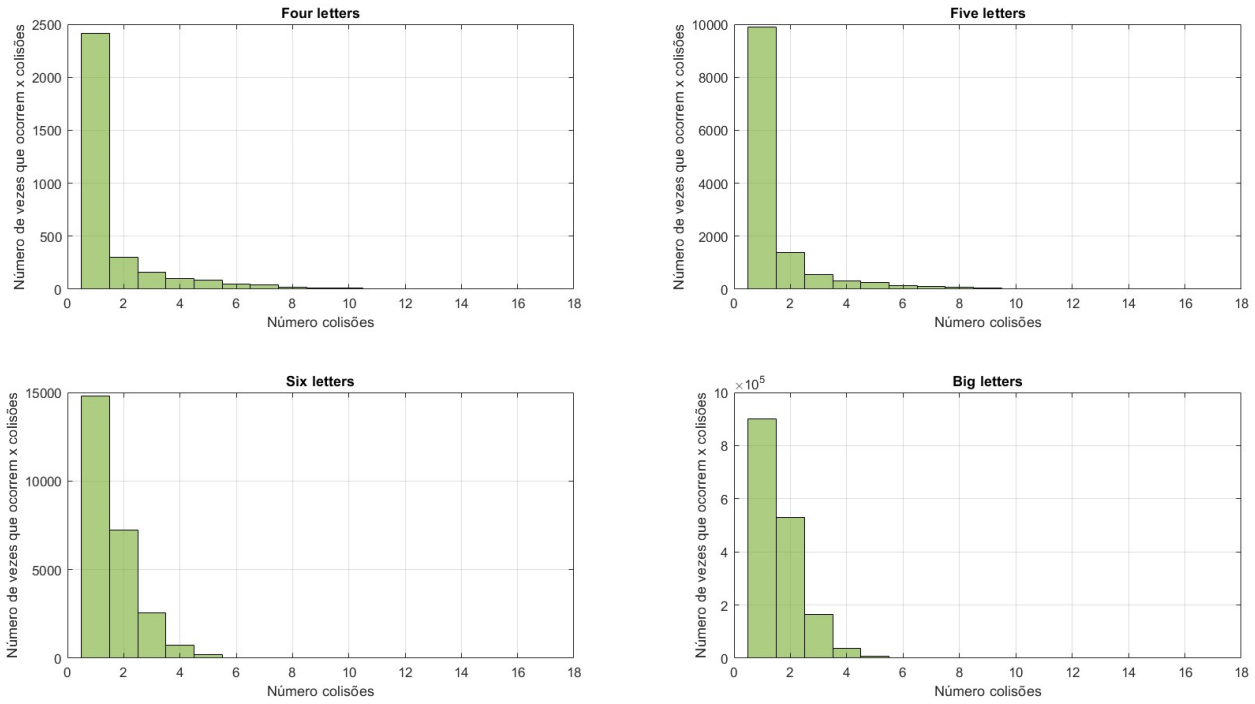

Fig.20 – Histogramas obtidos

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.

3. Código

3.1. 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;
}
```

```c
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;
}
```

3.2. 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
```

4. Conclusão

Concluindo, demonstrámos como criámos funções de modo a implementar uma *Word Ladder* para o trabalho proposto. Utilizámos os conhecimentos que adquirimos nas aulas práticas e teóricas como *Hash Table's* e *Linked List's*, no entanto, não realizámos todos os pontos pedidos (que estavam mais relacionados com a parte dos grafos).

Efetuamos testes a fim de implementar o programa que desenvolvemos e disponibilizámo-los juntamente com o output da função que permite visualizar as *Hash Table's* e as suas *Linked List's*. Constatámos também que, quanto maior a quantidade de palavras, maior será a probabilidade de ocorrerem colisões.

Por fim, usando a função *valgrind*, o programa não verificou nenhum caso de *memory leaks*, devolvendo zero erros. Isto é um aspeto importante visto que toda a memória alocada dinamicamente tem de ser libertada depois de já não ser necessária.