Projeto 2: Word Ladder

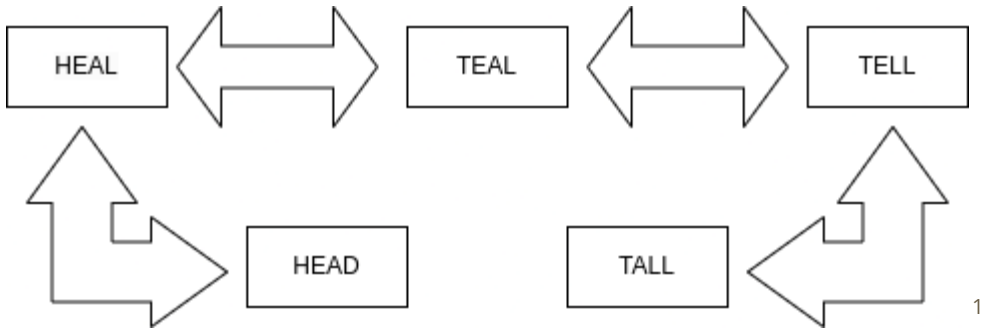
10/01/2023

Diogo Almeida (108902) / Gonçalo Ferreira (107853) / Tomás Matos (108624)
Professor Joaquim Madeira (P08)
Professor Tomás Oliveira e Silva (TP01)

Introdução

I. Explicação do Problema

O segundo projeto da disciplina de Algoritmos e Estruturas de Dados é baseado no jogo World Ladder, onde o jogador tem que encontrar uma cadeia de palavras que permitam ligar 2 palavras. Essas palavras intermédias devem apenas mudar uma letra de cada vez, por exemplo:



[1]

Com isto em mente, o objetivo é completar um programa inicialmente pelo professor ou fazer um novo programa que faça este processo através de ficheiros de texto contendo palavras de diferentes tamanhos.

O programa contém comentários explícitos onde podemos obter linhas-guia para realizar o projeto, que já contém algumas declarações de variáveis necessárias para completar partes do programa, principalmente a realização de uma estrutura hash table com linked lists, que permita construir grafos para aceder às diferentes palavras.

[1] Figura Ilustrativa do uma Word Ladder entre as palavras “Head” e “Tall”

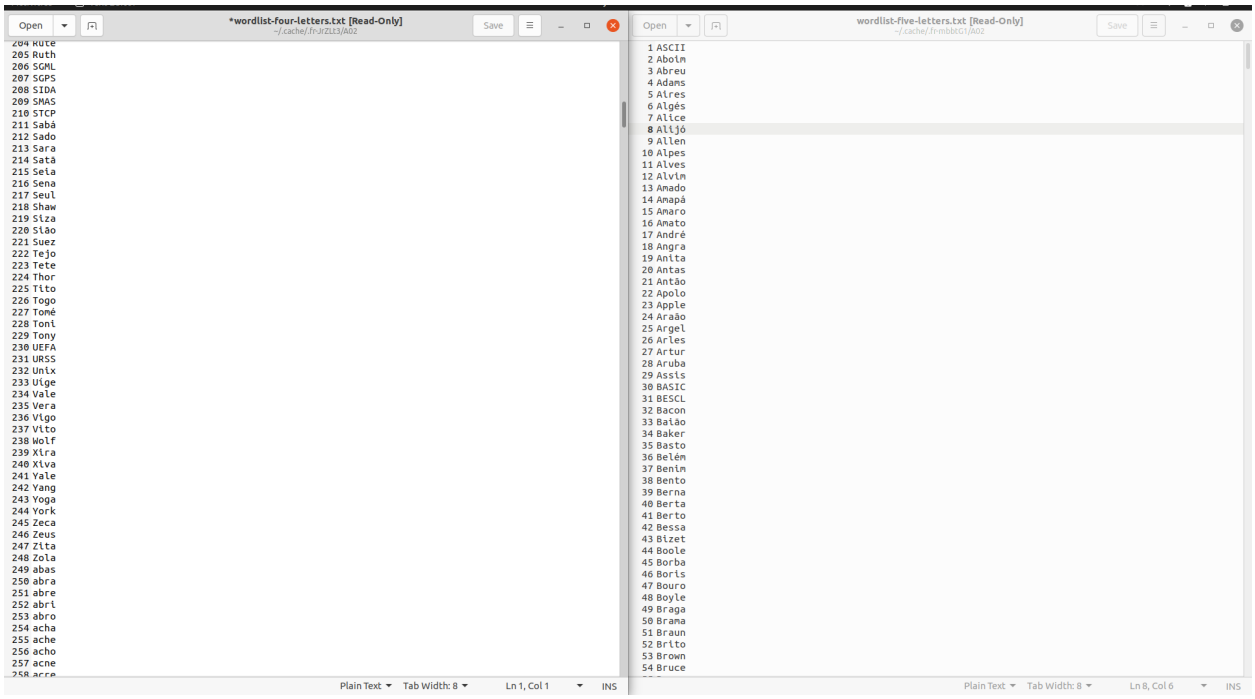
II. Análise do código dado

Assim, partimos para analisar o código que o professor deixou na plataforma *Elearning* para nos guiarmos no projeto em questão. O ficheiro é constituído por 2 pastas (A02 e P02) , onde a pasta P02 possui apenas o ficheiro header elapsed_time.h , que serve para podermos contar o tempo que o programa demora a correr, e assim podermos medir a eficiência do mesmo.

Já na pasta A02 existem os seguintes ficheiros:

- Ficheiros com listas de palavras

Nos ficheiros .txt da pasta fornecida pelo professor podemos encontrar ficheiros de texto que possuem listas de palavras por onde nos podemos guiar de maneira a criar grafos de palavras de quatro, cinco, seis ou mais letras, como podemos na seguinte imagem



2

[2] "Print" da composição dos ficheiros wordlist-four-letters.txt e wordlist-five-letters.txt

- Makefile

Este ficheiro é usado principalmente em projetos na linguagem C e tal como neste projeto, o objetivo é simplificar e automatizar a maneira como são criados os ficheiros, sendo assim apenas preciso digitar "make **comm**" onde **comm** é um dos seguintes comandos:

```
#
# Tomás Oliveira e Silva, AED, November 2022
#
# makefile to compile the A.02 assignment (word ladder)
#

clean:
	rm -rf a.out word_ladder solution_word_ladder


word_ladder:            word_ladder.c
	cc -Wall -Wextra -O2 word_ladder.c -o word_ladder -lm

solution_word_ladder:   solution_word_ladder.c
	cc -Wall -Wextra -O2 solution_word_ladder.c -o solution_word_ladder -lm
~
~
~
```

[3]

que executará uma das abaixo descritas no terminal.

[3] "Print" do código presente no ficheiro makefile

- word_ladder.c

```
//
// 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 th enext adjacency list node
  hash_table_node_t *vertex;         // the other 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
  // 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)
//
```

Aqui é a parte do programa onde se encontram as estruturas que vão ajudar a construir o grafo.

Podemos compreender que vamos ter que usar sistema de hash table para colocar cada *node*. A estrutura hash_table possui atributos como o seu tamanho , o número de nodes que entram na hash_table, e atributos como o número de que servem apenas para guardar informação sobre o grafo criado. Para além disso, possui uma estrutura que permite guardar a “cabeça” (primeiro node de uma linked list) de cada linked list, sendo este elemento um array de ponteiros.

A estrutura dos nodes tem como atributo principal e mais destacado a palavra, que o identifica, sendo os restantes atributos para detalhar a linked list e alguns outros para guardar informações necessárias à construção do grafo.

Os adjacency nodes representam nós ligados presentes numa lista ligada de adjacency nodes , onde cada um deles representa uma aresta do grafo, com atributos como o próximo adjacency node e o vértice.

```
//
// 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);
}
```

Aqui conseguimos visualizar parte de código que trata de alocar e liberar espaço na memória para os diferentes tipos de nodes. Trata também de verificar problemas e/ou erros nessa alocação de memória.

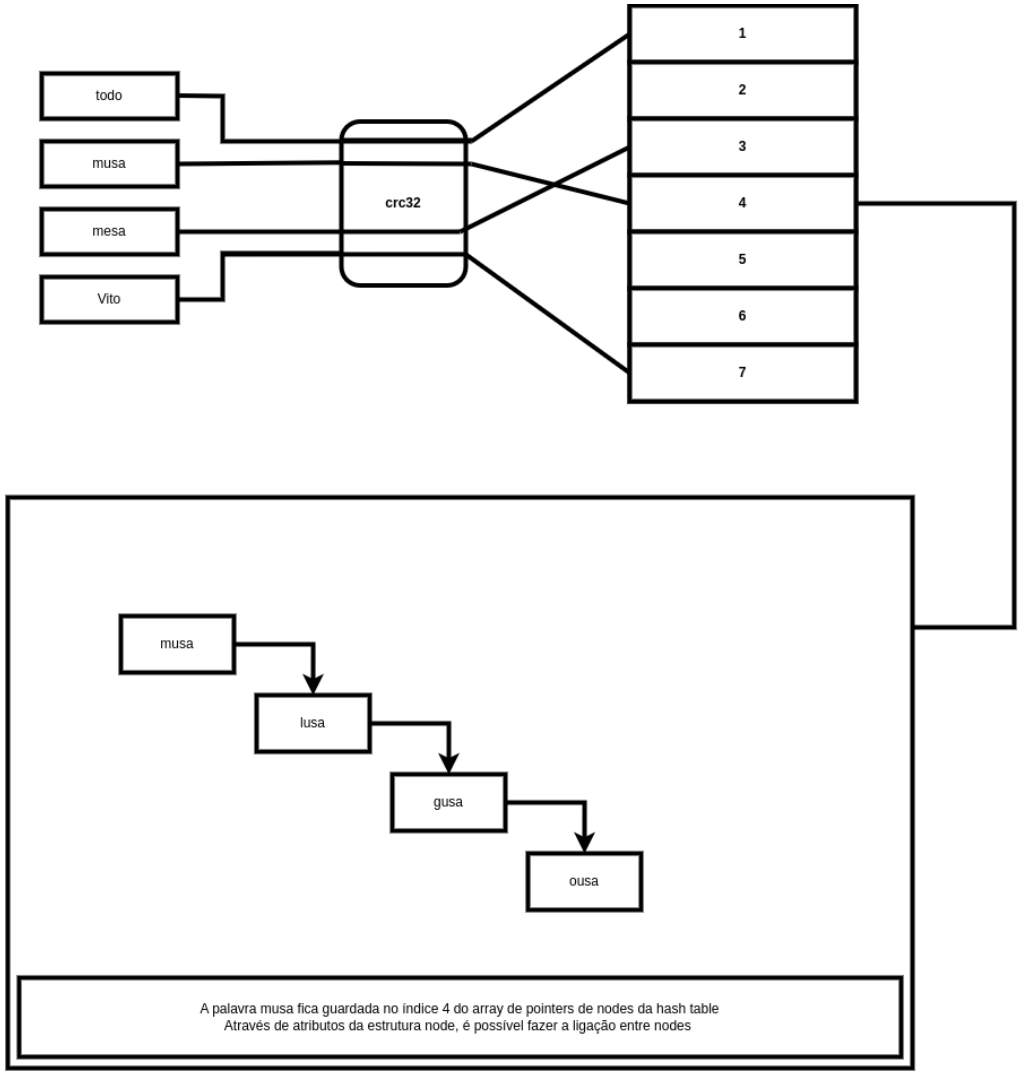
```
unsigned int crc32(const char *str)
{
  static unsigned int table[256];
  unsigned int crc;

  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;
}
```

Esta função é a hash function da nossa hash table. Esta função recebe (neste caso) uma palavra e transforma cada uma dessas palavras num hash code único, que depois ao dividir pelo tamanho da hash table, vamos obter o índice onde devemos inserir o node head da linked list.

Nota: As restantes funções usadas no código do professor não foram usadas no nosso código modificado, logo não explicaremos a sua importância.

III. Primeiras abordagens

Após analisar o código fornecido pelo professor consegue-se entender melhor o propósito do exercício e começar a montar uma solução para o problema. A nossa primeira ideia sempre foi montar um esquema tal como o que se encontra na imagem abaixo, conseguir colocar a estrutura da hash table completamente funcional e após isso trabalhar nos gráficos e verificar a validade dos resultados.



Explicação do código/funções desenvolvido(as)

- hash_table_create()

```
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 = 250;
  hash_table->heads = (hash_table_node_t **)malloc(sizeof(hash_table_node_t *) * hash_table->hash_table_size);
  hash_table->number_of_entries = 0;
  hash_table->number_of_edges = 0;
  for(i = 0;i < hash_table->hash_table_size;i++)
    hash_table->heads[i] = NULL;

  return hash_table;
}
```

Nesta função conseguimos visualizar a inicialização das estruturas de hash_table, onde é decidimos dar um tamanho inicial de 250, com intenções claras de aumentar o tamanho da mesma depois de preenchermos uma parte significativa destes espaços.

Encontramos aqui também a alocação de memória da linked list que permite aceder às outras linked lists todas, tal como o inicializar as variáveis que contam números de entradas e número de edges, ambas a zero.

Para terminar o programa, é necessário colocar todos os nodes alocados com o valor NULL.

- hash_table_grow()

```
178  static void hash_table_grow(hash_table_t *hash_table)
179  {
180    hash_table_node_t **old_heads, **new_heads, *node, *next;
181    unsigned int old_size, i;
182
183    // save a pointer to the old array of linked list heads and its size
184    old_heads = hash_table->heads;
185    old_size = hash_table->hash_table_size;
186
187    // create a new hash table with a larger size
188    hash_table->hash_table_size *= 2;
189    new_heads = (hash_table_node_t **)malloc(hash_table->hash_table_size * sizeof(hash_table_node_t *));
190    // check for out of memory
191    for (i = 0u; i < hash_table->hash_table_size; i++)
192      new_heads[i] = NULL;
193
194    if (new_heads == NULL)
195    {
196      fprintf(stderr, "hash_table_grow: out of memory");
197      exit(1);
198    }
```

Na função hash_table_grow() , o pressuposto é aumentar o tamanho da hash table. Para isso, a função começa por guardar ponteiros com as antigas variáveis como a linked list das linked lists na variável **old_heads e criar novos ponteiros como **new_heads, ponteiro esse que guardará a próxima heads.

Depois trata de alocar memória para uma nova hash_table, com o dobro da memória para uma nova linked list com o dobro da memória.

No final , tal como quando criamos a hash_table, devemos colocar os valores recentemente alocados todos em NULL.

- hash_table_free()

```
222  static void hash_table_free(hash_table_t *hash_table)
223  {
224    hash_table_node_t *node;
225    hash_table_node_t *temp;
226    adjacency_node_t *adj_node;
227    adjacency_node_t *temp_adj;
228    unsigned int i;
229
230    for (i = 0; i < hash_table->hash_table_size; i++)
231    {
232      node = hash_table->heads[i];
233      while (node != NULL)
234      {
235        temp = node;
236        adj_node = node->head;
237        while (adj_node != NULL)
238        {
239          temp_adj = adj_node;
240          adj_node = adj_node->next;
241          free(temp_adj);
242        }
243        node = node->next;
244        free(temp);
245      }
246    }
247
248    free(hash_table->heads);
249
250    free(hash_table);
251  }
```

Nesta função sucede-se a libertação da memória alocada para as hash_tables criadas. è suposto percorrermos todos os valores da linked list principal e entrar de em cada uma das linked lists e libertar a memória de cada um dos nodes, tal como os adjacency nodes ligados. Após isso libertamos a linked list principal e por final a hash_table.

- find_word()
-

```
253  static hash_table_node_t *find_word(hash_table_t *hash_table,const char *word,int insert_if_not_found)
254  {
255    hash_table_node_t *node;
256    unsigned int i;
257
258    // printf("word: %s\n", word);
259    i = crc32(word) % hash_table->hash_table_size;
260    node = hash_table->heads[i];
261    while (node != NULL)
262    {
263      if (strcmp(node->word, word) == 0)
264        return node;
265      node = node->next;
266    }
267
268    if (insert_if_not_found && strlen(word) < _max_word_size_)
269    {
270      node = allocate_hash_table_node();
271      strncpy(node->word, word, _max_word_size_);
272      node->representative = node;
273      node->next = hash_table->heads[i];
274      node->previous = NULL;
275      node->number_of_edges = 0;
276      node->number_of_vertices = 1;
277      node->visited = 0;
278      node->head = NULL;
279      hash_table->heads[i] = node;
280      hash_table->number_of_entries++;
281      if (hash_table->number_of_entries > hash_table->hash_table_size)
282        hash_table_grow(hash_table);
283
284      return node;
285    }
286
287    return NULL;
288  }
```

A find_word() é uma função que serve para verificar e/ou inserir palavras como nodes na hash_table.

Primeiro , calcula-se o índice onde para onde vamos inserir o node na linked list específica. Após isso é percorrido cada um dos nós da hash_table de maneira a verificar se a palavra que foi passada como argumento de entrada da função já existe ou não na hash_table. Caso exista, retornamos o node onde essa palavra se encontra, caso não, verificamos o parâmetro de entrada **insert_if_not_found**.

Caso este valor se encontre a 1 e a palavra não ultrapasse o limite de caracteres que é suposto, tratamos de alocar, colocar o novo node com a nova palavra na linked list e interligar os nodes, de maneira a que não se perca a conectividade da lista. Passamos pelo final a aumentar o número de entradas, após isso verificamos se existe necessidade de aumentar o tamanho comparando as entradas na hash_table com o tamanho da mesma, e no final de tudo retorna o próprio node.

Não encontrada a palavra e se o **insert_ not_found** estiver a 0 ou a palavra tiver mais letras do que as pressuposto , o valor retornado é NULL.

- find_representative()

Função simples:

```
static hash_table_node_t *find_representative(hash_table_node_t *node)
{
  hash_table_node_t *representative,*next_node,*node_atual; //Inicialização de variáveis

  //Loop que acha o representativo de node
  for (representative = node; representative != representative->representative; representative = representative->representative)
  ;

  return representative;

}
```

A função *find_representative()* tem como objetivo achar o nó representativo do grafo onde o nó dado como argumento pertence. Os nós da hash table que possuem uma possível ligação entre eles irão estar contidos num grafo. A cada grafo será atribuído um nó que irá ser o nó representativo. O nó representativo é o nó do grafo em que o ponteiro representative irá apontar para si próprio. Cada nó desse grafo irá possuir um ponteiro chamado representative que irá apontar para esse nó representativo.

Primeiro iremos passar como argumento o nó que pretendemos achar o seu nó representativo (dá entrada um ponteiro para esse nó): `hash_table_node_t *node`

De seguida fazemos a inicialização de variáveis que nos irão ser úteis no desenvolver da função:

```
hash_table_node_t              *representative,*next_node,*node_atual;
//Inicialização de variáveis
```

Por fim é criado um loop que começa por igualar a variável `representative` ao nó inicial dado como argumento (`node`). Depois vai igualando o valor do seu atributo representative a si mesmo (`representative = representative->representative`) até que o seu atributo representative seja igual a si mesmo e então pára o loop pois esse irá ser o nó representativo e então dá return nesse nó: `return representative;`

Para melhorar a eficiência do programa podemos adicionar mais um loop:

```
static hash_table_node_t *find_representative(hash_table_node_t *node)
{
  hash_table_node_t *representative,*next_node,*node_atual; //Inicialização de variáveis

  //Loop que acha o representativo de node
  for (representative = node; representative != representative->representative; representative = representative->representative)
  ;

  //Loop que atualiza o representative de todos os nós next ao node
  for (node_atual = node; node_atual != representative; node_atual = next_node)
  {
    next_node  = node_atual->representative;
    node_atual->representative = representative;
  }

  return representative;

}
```

O segundo loop irá começar por igualar a variável *node_atual* ao nó dado como argumento. Enquanto o *node_atual* não for igual ao representative que foi achado anteriormente o *node_atual* vai recebendo o valor de que está guardado no atributo representative de si próprio.

A variável *next_node* vai guardar o que está guardado no atributo representative do node_atual para de seguida atualizar o valor do atributo representative do node_atual para o representative que foi encontrado no loop anteriormente explicado. Assim vamos colocando todos os nós a "apontar" para o nó representativo, melhorando assim a eficiência da função em chamadas futuras.

- add_edge()

```c
static void add_edge(hash_table_t *hash_table, hash_table_node_t *from, const char *word)
{
  hash_table_node_t *to, *from_representative, *to_representative;
  from_representative = find_representative(from); //encontra o representante do vertice de origem
  to = find_word(hash_table, word, 0); //verifica se o vertice de destino existe na hash table

  if (to == from) //verifica se o vertice de origem é igual ao de destino
    return;
  to_representative = find_representative(to); //encontra o representante do vertice de destino

  if (from_representative == to_representative) // verifica se os representantes são iguais
  {
    from_representative->number_of_vertices++;//incrementa o numero de vertices do representante
  }
  else{
    if (from_representative->number_of_vertices < to_representative->number_of_vertices) //verifica qual dos representantes tem mais vertices
    {
      from_representative->representative = to_representative; //faz o representante do vertice de origem apontar para o representante do vertice de destino
      to_representative->number_of_vertices += from_representative->number_of_vertices; //incrementa o numero de vertices do representante do vertice de destino
      to_representative->number_of_edges += from_representative->number_of_edges; //incrementa o numero de arestas do representante do vertice de destino
    }
    else
    {
      to_representative->representative = from_representative; //faz o representante do vertice de destino apontar para o representante do vertice de origem
      from_representative->number_of_vertices += to_representative->number_of_vertices; //incrementa o numero de vertices do representante do vertice de origem
      from_representative->number_of_edges += to_representative->number_of_edges; //incrementa o numero de arestas do representante do vertice de origem
    }
  }
  adjacency_node_t *linkfrom = allocate_adjacency_node(); //aloca memoria para o vertice de origem
  adjacency_node_t *linkto = allocate_adjacency_node(); //aloca memoria para o vertice de destino

  linkfrom->vertex = to; //faz o vertice de origem apontar para o vertice de destino
  linkfrom->next = from->head; //faz o proximo vertice apontar para o vertice de origem
  from->head = linkfrom; //faz o vertice de origem apontar para o vertice de destino

  linkto->vertex = from; //faz o vertice de destino apontar para o vertice de origem
  linkto->next = to->head;//faz o proximo vertice apontar para o vertice de destino
  to->head = linkto;//faz o vertice de destino apontar para o vertice de origem

  from_representative->number_of_edges++; //incrementa o numero de arestas do representante do vertice de origem
  to_representative->number_of_edges++; //incrementa o numero de arestas do representante do vertice de destino
  hash_table->number_of_edges++; //incrementa o numero de arestas da hash table
  return;
}
```

Esta função tem como objetivo criar uma aresta que liga dois vértices (dois nós) similares de forma a criar um grafo com todas as palavras que possuem uma ligação entre si.

A função recebe de entrada a hash table, um nó inicial e uma word similar à word desse nó.

```c
from_representative = find_representative(from); //encontra o representante do vertice de origem
to = find_word(hash_table, word, 0); //verifica se o vertice de destino existe na hash table
```

Depois é chamada a função find_representative que vai retornar o nó representativo do nó inicial e é chamada também a função find_word que vai retornar o nó da hash table que possui a word que foi dada como argumento. É também achado o nó representativo desse nó e guardado em to_representative.

É feita uma comparação entre o nó inicial e o nó da palavra similar caso os nós sejam iguais é porque é a mesma palavra e então não é necessário adicionar aresta nenhuma e por isso é feito return.

```
if (from_representative == to_representative) // verifica se os representantes são iguais
{
  from_representative->number_of_vertices++;//incrementa o numero de vertices do representante
}
```

Depois de se concluir que os nós são diferentes são comparados os representativos de cada nó e caso sejam iguais é porque o nó já pertence ao grafo e por isso incrementa se o número de vértices associado ao representativo do grafo pois existe mais um nó no grafo.

```
}else{
  from_representative->representative = to_representative; //mete o representante do from igual ao do to
  to_representative->number_of_vertices += from_representative->number_of_vertices; //soma o numero de vertices do from ao do to
  to_representative->number_of_edges += from_representative->number_of_edges; //soma o numero de arestas do from ao do to
}
```

Caso os representativos sejam diferentes os representativos dos nós são igualados tanto como o número de vértices e de arestas.

```
adjacency_node_t *linkfrom = allocate_adjacency_node(); //aloca memoria para o linkfrom
adjacency_node_t *linkto = allocate_adjacency_node(); //aloca memoria para o linkto

linkfrom->vertex = to; //mete o vertice do linkfrom igual ao to
linkfrom->next = from->head;//mete o next do linkfrom igual ao head do from
from->head = linkfrom; //mete o head do from igual ao linkfrom

linkto->vertex = from; //mete o vertice do linkto igual ao from
linkto->next = to->head; //mete o next do linkto igual ao head do to
to->head = linkto; //mete o head do to igual ao linkto

from_representative->number_of_edges++; //incrementa o numero de arestas do from
to_representative->number_of_edges++; //incrementa o numero de arestas do to
hash_table->number_of_edges++; //incrementa o numero de arestas da hash table
```

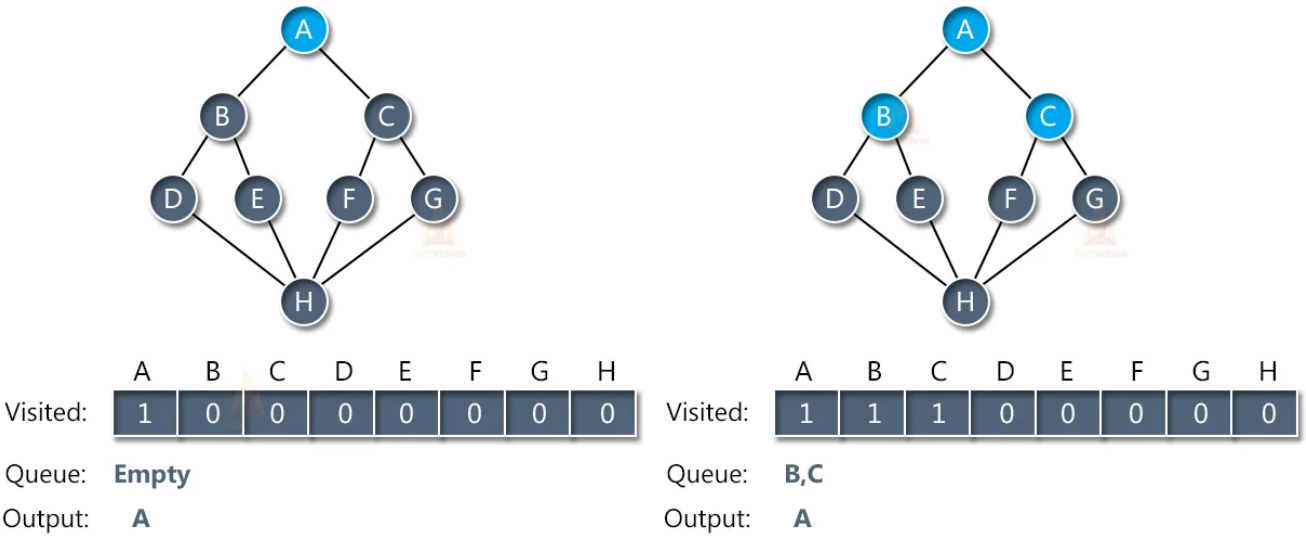
É feita uma alocagem de memória de dois ponteiros da classe adjancecy_node_t chamados linkfrom e linkto. O vértice do linkfrom vai ser o nó da palavra similar, o next da linkfrom vai ser o primeiro nó da lista do nó from e faz o inverso para o linkto. Depois incrementa o número de arestas do representativo do from do to e da hash table.

- breadh_first_search()

```c
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)
{
  int front = 0, back = 0; // front e back da queue
  list_of_vertices[back++] = origin; // adiciona o origin na queue
  origin->visited = 1; // marca o origin como visitado
  origin->previous = NULL; // marca o origin como o primeiro da queue
  int finish = 0;// variavel para saber se chegou ao goal

  while (front < back) // enquanto a queue nao estiver vazia
  {
    hash_table_node_t *vertex = list_of_vertices[front++]; // retira o primeiro da queue
    if (finish == 1) // se ja chegou ao goal
    {
      break;
    }

    adjacency_node_t *adjacent = vertex->head; // aponta para o primeiro adjacent do vertex
    while (adjacent != NULL)
    {
      if (adjacent->vertex->visited == 0) // se o adjacent nao tiver sido visitado
        {
          list_of_vertices[back++] = adjacent->vertex;// adiciona o adjacent na queue
          adjacent->vertex->visited = 1;// marca o adjacent como visitado
          adjacent->vertex->previous = vertex; // marca o adjacent como o anterior do vertex
          if (adjacent->vertex == goal) // se o adjacent for o goal
          {
            finish = 1;
            break;
          }
        }
      adjacent = adjacent->next; // aponta para o proximo adjacent
    }
  }
  for (int i = 0; i < back; i++) // percorre a queue
  {
    list_of_vertices[i]->visited = 0;// marca todos os vertices como nao visitados
  }
  return back; // retorna o numero de vertices visitados
}
```

O objetivo desta função é fazer uma busca em largura (Breadth first traversal) começando num vértice origin até outro vértice goal que são dados como argumento. Também são dados com argumento uma lista de vértices que vai guardar todos os vértices que foram percorridos desde o origin ao goal. Temos também um maximum_number_of_vertices para estabelecermos um número máximo de vértices que a função pode percorrer.

Para a função vamos usar uma queue em que vamos adicionando os vértices que vão ser analisados para saber se temos de passar por eles.



Primeiramente começamos por inicializar um front e um back que serão dois ponteiros que irão apontar para o primeiro e último vértice da queue. Depois colocamos o vértice origin como inicial na queue e colocamos o origin com visitado.

É feito um loop, enquanto a queue não estiver vazia (front<back). Em cada iteração é retirado o primeiro vértice da queue através do ponteiro front e verifica-se se já foi visitado. Caso não tenha sido visitado é colocado como visitado, adicionam-se os adjacentes desse vértice à queue e faz-se a verificação se ele é o goal. Se for o goal o loop é parado mas se não for é retirado da queue o próximo vértice e faz tudo novamente.

Por fim são colocados todos os vértices a não visitados.

- ## list_connected_component()

Esta função está a ser usada para imprimir as palavras que pertencem à mesma componente conexa na nossa hash table.

```
static void list_connected_component(hash_table_t *hash_table,const char *word)
{

  hash_table_node_t *node;

  // encontramos o nó que correspode à variavel dada no argumento
  node = find_word(hash_table, word, 0);

  if (node == NULL)
  {
    return;
  }

  // encontramos o resprensentativo dessa palavra
  hash_table_node_t *representative = find_representative(node);

  int Nr_max = representative->number_of_vertices;
  hash_table_node_t **list_of_vertices = (hash_table_node_t **)malloc(Nr_max * sizeof(hash_table_node_t *));

  int nrPalavras = breadh_first_search(Nr_max, list_of_vertices, node, NULL);

  int count = 1;
  for(int i = 0;i < nrPalavras;i++){
    printf("[%d] %s\n",count,list_of_vertices[i]->word);
    count++;
  }
  free(list_of_vertices);
}
```

Inicialmente, encontramos o nó (node) que corresponde à palavra que nos é dada como argumento através da função referenciada anteriormente *find_word()*, como argumentos desta colocamos a nossa hash table, a palavra e 0 uma vez que não pretendemos adicionar esta palavra se esta não existir. Se esta não existir, o nosso nó vai ser *NULL* e saímos da função.

Depois de verificarmos a existência da palavra, encontramos o representativo do nó a partir da função *find_representative()*.

Seguidamente, inicializamos duas variáveis *Nr_max* e *list_of_vertices* que vão ser utilizadas quando chamar-mos a *breadh_first_search*. O quarto argumento que utilizamos quando chamamos esta função, que corresponde ao *goal* (palavra do destino), é dado como *NULL*. Quando isto acontece a *breadh_first_search* vai percorrer todos os vértices possíveis até os ter visitado todos.

Por último, imprimimos o array *list_of_vertices* que inicializamos anteriormente e a *breadth first search* alterou e libertamos-lhe a memória.

- Implementação com Depth First

```c
#define MAX_WORDS 100000
char printed_words[MAX_WORDS][_max_word_size_];  //array para guardar as palavras que já foram repetidas
int nr_printed_words = 1;                        // variável para contar o número de palavras que já foram escritas

static void list_connected_component(hash_table_t *hash_table,const char *word)
{

  hash_table_node_t *node1;
  node1 = find_word(hash_table, word, 0);

  if (node1 == NULL)
  {
    return;
  }

  hash_table_node_t *representative = find_representative(node1);
  adjacency_node_t *adj_node = node1->head;

  // ciclo para percorrer as palavras adjacentes ao nó passado como argumento
  while (adj_node != NULL)
  {
    int validar = 0;
    hash_table_node_t *node = adj_node->vertex;

    for (int i = 0; i < nr_printed_words; i++)
    {
      //vamos verificar se já ecrevemos a palavra passada como argumento no terminal
      if (strcmp(printed_words[i], node->word) == 0)
      {
        validar = 1;
      }
    }
    // se ainda não tivermos escrito essa palavra:
    if (validar == 0)
    {
      //1) escrevemos a palavra
      printf("%s\n", node->word);

      //2) adicionamos ao array printed_words
      strcpy(printed_words[nr_printed_words-1], node->word);

      //3) incrementamos o indice do array (nr de palavras escritas)
      nr_printed_words++;

      //4) chamamos a função recursiva para o nó que acabámos de escrever
      list_connected_component(hash_table, node->word);
    }else
    {
      // se já tivermos escrito essa palavra passamos para os seguites nós adjacentes
      adj_node = adj_node->next;
    }
  }
}
```

Esta é uma implementação da *list_connected_component()* usando método Depth First Search, que consiste na ideia de explorar o máximo possível em uma direção. Ele funciona percorrendo o ramo inteiro de um nó antes de passar para os nós adjacentes.

Primeiramente inicializamos duas variáveis que vão ser usadas para guardar as palavras que já foram escritas no terminal (printed_words) e um contador da quantidade de palavras escritas que vai ser usado como índice do array.

Na função primeiramente implementamos um ciclo *while* que vai ter o intuito de percorrer os nós adjacentes do nó passado como argumento, dentro deste ciclo adicionamos outro ciclo que vai validar se já escrevemos a palavra em questão no terminal. Se já tiver sido escrita passamos para o nó adjacente seguinte.

Caso contrário, escrevemos essa palavra e vamos armazená-la no array printed_words e chamamos a função novamente para conseguirmos a profundidade referida anteriormente. Isto é repetido até que todas as palavras adjacentes tenham sido verificadas.

```
if(command == 1)
{

  if(scanf("%99s",word) != 1)
    break;

  //novo
  memset(printed_words, 0, sizeof(printed_words));
  nr_printed_words = 1;
  list_connected_component(hash_table,word);
  printf("> Total de palavras: %d\n", nr_printed_words-1);

}
```

Na main temos que alterar alguns pormenores como inicializar o array printed_words novamente, o contador de palavras escritas a 1. Após isso podemos chamar a função*list_connected_component()* e imprimir também o número total de palavras conectadas. Apesar desta implementação, não utilizamos a Depth First Search uma vez que com este método é impossível calcular o caminho mais curto entre duas palavras.

- connected_component_diameter()

Esta função é utilizada para determinar a maior distância entre dois vértices aleatórios dentro de um componente conectado de um grafo. Começamos por inicializar estas três variáveis:

```
static int largest_diameter;  //variavel para guardar o maior diametro de entre todas as componentes
static hash_table_node_t **largest_diameter_example; // array que guarda os nós do maior caminho
static int Sum_Diameters, Num_Diameters; // variaveis para calcular a média dos caminhos
```

```
static int connected_component_diameter(hash_table_node_t *node)
{
  int Nr_max = node->number_of_vertices;

  hash_table_node_t **list_of_vertices = (hash_table_node_t **)malloc(Nr_max * sizeof(hash_table_node_t *));
  int nrPalavras = breadh_first_search(Nr_max, list_of_vertices, node, NULL);
  int diametro = 0;

  for(int i = 0;i < nrPalavras;i++){ //----- 1 -----//

    hash_table_node_t **list_of_vertices_Dentro = (hash_table_node_t **)malloc(Nr_max * sizeof(hash_table_node_t *));
    //----- 2 -----//
    int dist = breadh_first_search(Nr_max, list_of_vertices_Dentro, list_of_vertices[i], NULL);

    hash_table_node_t *path = list_of_vertices_Dentro[dist-1];
    int realDist = 0;
    //----- 3 -----//
    while (path != NULL)
    {
      realDist++;
      path = path->previous;
    }
    //----- 4 -----//
    if (realDist > diametro)
    {
      diametro = realDist;
    }
    //----- 5 -----//
    if (realDist > largest_diameter)
    {
      largest_diameter = realDist;
      hash_table_node_t *palav = list_of_vertices_Dentro[dist-1];
      free(largest_diameter_example);
      largest_diameter_example = (hash_table_node_t **)malloc(largest_diameter * sizeof(hash_table_node_t *));
      int i = 0;
      //----- 6 -----//
      while (palav != NULL)
      {
        largest_diameter_example[i] = palav;

        i++;
        palav = palav->previous;
      }
    }
    free(list_of_vertices_Dentro);
    free(path);
  }
  //----- 7 -----//
  Sum_Diameters = Sum_Diameters + diametro-1;
  Num_Diameters++;
  free(list_of_vertices);
  return largest_diameter;
}
```

Após inicializar estas variáveis utilizamos a função *breadh_first_search* com o argumento *goal* (objetivo) a *NULL* para encontrar o número de palavras da componente conexa relacionada à palavra dada como argumento. Seguidamente, entramos um ciclo *for* sinalizado com o número **1** que vai percorrer todas as palavras da rede conexa da palavra em questão.

Avançando, após inicializarmos o array 'list_of_vertices_Dentro', chamamos novamente a função *breadh_first_search* (**2**) que vai armazenar neste array o valor de todas as palavras conexas por ordem dos níveis de adjacência.

```
int dist = breadh_first_search(Nr_max, list_of_vertices_Dentro, list_of_vertices[i], NULL);
```

Seguidamente, criamos um nó (path) que corresponde ao último elemento de 'list_of_vertices_Dentro' , uma vez que este é o que vai ter maior distância possível. Tendo este valor, usamos um (**3**) *while* para contar o caminho mais curto entre estes dois vértices usando o *previous* que foi implementado na *breadh_first_search*.

```
while (path != NULL)
{
  realDist++;
  path = path->previous;
}
```

A condição que se encontra comentada com o número **4** , vai comparar uma variável interna com a distância entre a dois vértices, para repararmos qual é o maior diâmetro na componente conexa em questão.

A condição seguinte (**5**) é referente à ao maior diâmetro entre todas as componentes, quando encontrarmos um valor maior que o anterior da variável externa 'largest diameter' e guardamos a maior sequência em outra variável externa (**6**) (largest diameter example).

```
largest_diameter_example = (hash_table_node_t **)malloc(largest_diameter * sizeof(hash_table_node_t *));
int i = 0;
//----- 6 -----//
while (palav != NULL)
{
  largest_diameter_example[i] = palav;

  i++;
  palav = palav->previous;
}
```

Após libertarmos a memória necessária, em duas variáveis também externas, somamos os valores dos diâmetros e contamos quantas vezes a função é chamada, que vão ter a utilidade de futuramente fazer o *graph_info*.

- graph_info()

```
static void graph_info(hash_table_t *hash_table)
{
  hash_table_node_t **representatives = malloc(sizeof(hash_table_node_t *) * hash_table->hash_table_size);
  int nrRepresentative = find_connected_component_representatives(hash_table, representatives);

  largest_diameter = 0;
  int distMaior;

  Sum_Diameters = 0;
  Num_Diameters = 0;

  for (int i = 0; i < nrRepresentative; i++)
  {
    distMaior = connected_component_diameter(representatives[i]);
  }

  free(representatives);
}
```

Nesta função implementamos uma outra função chamada *find_connected_component_representatives*:

```
static int find_connected_component_representatives(hash_table_t *hash_table, hash_table_node_t **representatives)
{
  int index = 0;

  for (int i = 0; i < hash_table->hash_table_size; i++)
  {
    for (hash_table_node_t *vertex = hash_table->heads[i]; vertex != NULL; vertex = vertex->next)
    {
      hash_table_node_t *representative = find_representative(vertex);

      if (!representative->visited)
      {
        representatives[index++] = representative->word;
        representative->visited = 1;
      }
    }
  }

  // tornamos o estado dos vertices que estavam visitados para 0
  for (int i = 0; i < hash_table->hash_table_size; i++)
  {
    for (hash_table_node_t *vertex = hash_table->heads[i]; vertex != NULL; vertex = vertex->next)
    {
      vertex->visited = 0;
    }
  }

  return index;
}
```

A função *find_connected_component_representatives* vai utilizar o array representatives e vai atualizar os seus valores adicionando lhe todos os representativos diferentes

Como forma de conseguir esse objetivo, nesse array vamos alterar nos representativos que estamos a iterar o estado visited para 1 e apenas adicionamos elementos ao *representatives* se estes tiverem o estado *visited* 0:

```
hash_table_node_t *representative = find_representative(vertex);

if (!representative->visited)
{
  representatives[index++] = representative->word;
  representative->visited = 1;
}
```

Por fim, colocamos o estado de *visited* dos representativos a 0 e retornamos o número de representativos diferentes:

```
for (int i = 0; i < hash_table->hash_table_size; i++)
{
  for (hash_table_node_t *vertex = hash_table->heads[i]; vertex != NULL; vertex = vertex->next)
  {
    vertex->visited = 0;
  }
}
```

A função *graph_info()* que nós estamos a implementar apenas percorre todos os representativos e para cada um chama a função *connected_component_diameter()* como forma de atualizar todas as nossas variáveis externas.

A informação referente ao número de vértices, número de arestas, o maior diâmetro, a maior sequência de palavras, ..., está a ser implementada na *main* no ponto 5.

- ## Main atualizada

Atualizou-se a função main do nosso programa para termos todas as funcionalidades pedidas:

```
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            (print words )
  4            (statistics hash )
  5            (graf info )
  6            (terminate)
```

Adicionou-se o caso 3 que vai imprimir todas as palavras que estão na hash_table e os correspondentes índices:

```
else if (command == 3)
{
  for(i = 0u;i < hash_table->hash_table_size;i++)
  {
    for(node = hash_table->heads[i];node != NULL;node = node->next)
    {
      printf("índice = %u -> %s\n",i,node->word);
    }
    printf("\n");
  }
}
```

O caso 4 vai imprimir a informação referente à hash_table:

```
static void hash_stats(hash_table_t *hash_table){
  printf("\n\nTamanho da Hash table: %u\n", hash_table->hash_table_size);
  printf("Número de entradas: %u\n", hash_table->number_of_entries);
  printf("Tamanho médio das Heads com Heads vazias: %f\n",(float)hash_table->number_of_entries / hash_table->hash_table_size);

  hash_table_node_t *node;
  unsigned int MaxHead = 0;
  unsigned int MinHead = _max_word_size_;
  unsigned int conta = 0;
  for(unsigned int i = 0u;i < hash_table->hash_table_size;i++){  // percorrer todos os indices
    unsigned int sizeHeads=0;                                    // contar o número de elementos de cada head
    for(node = hash_table->heads[i];node != NULL;node = node->next){  // percorrer todas as heads
      sizeHeads++;
    }
    if (sizeHeads != 0)
    {
      conta++;
    }
    if( sizeHeads > MaxHead){
      MaxHead = sizeHeads;
    }
    if( sizeHeads < MinHead && sizeHeads != 0){
      MinHead = sizeHeads;
    }
  }
  printf("Tamanho médio das Heads sem Heads vazias: %f\n",(float)hash_table->number_of_entries / conta);
  printf("Tamanho da maior Linked List: %u\n", MaxHead);
  printf("Tamanho da menor Linked List: %u\n\n", MinHead);
}
```

Imprimimos sobre a Hash table:

- Tamanho da Hash table;
- Número de entradas;
- Tamanho médio das Heads contando as que não contêm informação;
- Tamanho médio das Heads contando apenas aquelas com informação;
- E o maior e menor tamanho das Heads
-

Na main:

```
else if (command == 4)
{
  hash_stats(hash_table);
}
```

Por último, o caso 5 que vai conter a informação do grafo que criámos:

```
else if (command == 5)
{
  printf("Número de vértices: %d\n", hash_table->number_of_entries);
  printf("Número de arestas: %d\n\n", hash_table->number_of_edges);

  hash_table_node_t **representatives = malloc(sizeof(hash_table_node_t *) * hash_table->hash_table_size);
  int nrRepresentative = find_connected_component_representatives(hash_table, representatives);
  printf("Número de componentes conexas: %d\n\n", nrRepresentative);

  int size_small_comp = 10;
  int size_large_comp = 0;
  int sum_size_comp = 0;
  // ciclo que vai percorrer todos os representativos
  for (int i = 0; i < nrRepresentative; i++)
  {
    int Nr_max = representatives[i]->number_of_vertices; //número de vertices do componente cujo representativo é representatives[i]
    hash_table_node_t **list_of_vertices = (hash_table_node_t **)malloc(Nr_max * sizeof(hash_table_node_t *));

    int nrPalavras = breadh_first_search(Nr_max, list_of_vertices, representatives[i], NULL);

    sum_size_comp = sum_size_comp + nrPalavras;

    if (size_large_comp < nrPalavras)   // verificamos qual o maior número de componentes de uma componente
    {
      size_large_comp = nrPalavras;
    }

    if (size_small_comp > nrPalavras)  // verificamos qual o menor número de componentes de uma componente
    {
      size_small_comp = nrPalavras;
    }

    free(list_of_vertices);
  }
  printf("Tamanho mais pequeno de uma componente: %d\n", size_small_comp-1);
  printf("Tamanho maior de uma componente: %d\n", size_large_comp-1);
  printf("\nValor médio dos tamanhos das componentes: %f \n\n", (float)sum_size_comp/nrRepresentative);
  printf("\nlargest: %d\n\n", largest_diameter-1);
  printf("Largest Ladder:\n");
  for (int i = 0; i < largest_diameter; i++)
  {
    printf("[%d] %s\n", i+1, largest_diameter_example[i]->word);
  }

  printf("\nValor médio dos diametros: %f \n\n", (float)Sum_Diameters / Num_Diameters);

  free(representatives);
}
```

No comando 5 vamos imprimir:

- Número de vértices de vértices do grafo;
- Números de arestas do grafo;
- Tamanho mais pequeno e o maior entre o tamanho de todas as componentes;
- Tamanho médio do tamanho de todas as componentes;
- Maior diâmetro de todos;
- Escada com o maior tamanho (ou uma delas);
- Valor médio de todos os diâmetros;

Resultados

Ficheiro com 4 letras

Comando:

1 tudo	2 tudo nada	3
Nota: Últimos 37 elementos. [1894] anal [1895] anel [1896] unir [1897] unes [1898] uniu [1899] ungi [1900] está [1901] crês [1902] éter [1903] unem [1904] asou [1905] orou [1906] irou [1907] grou [1908] alou [1909] apôs [1910] após [1911] apor [1912] edil [1913] geou [1914] ovou [1915] três [1916] apus [1917] obus [1918] ecos [1919] uval [1920] aval [1921] Ovar [1922] egos [1923] unam [1924] usou [1925] içou [1926] grau [1927] opor [1928] ecoo [1929] ecoe [1930] ecoa [1931] odor	> 2 tudo nada [1] tudo [2] todo [3] nodo [4] nado [5] nada	índice = 3969 -> ater índice = 3969 -> revê índice = 3972 -> Néri índice = 3973 -> esta índice = 3974 -> fusa índice = 3977 -> icem índice = 3980 -> lhas índice = 3980 -> lusa índice = 3981 -> musa índice = 3983 -> Óder índice = 3983 -> ousa índice = 3983 -> ovei índice = 3988 -> Toni índice = 3988 -> Tito índice = 3990 -> Vito

Comando:

```
> 4


Tamanho da Hash table: 4000
Número de entradas: 2149
Tamanho médio das Heads com Heads vazias: 0.537250
Tamanho médio das Heads sem Heads vazias: 1.475962
Tamanho da maior Linked List: 6
Tamanho da menor Linked List: 1
```

```
> 5
Número de vértices: 2149
Número de arestas: 9267

Número de componentes conexas: 187

Tamanho mais pequeno de uma componente: 0
Tamanho maior de uma componente: 1930

Valor médio dos tamanhos das componentes: 11.491979


largest: 15

Largest Ladder:
[1] Marx
[2] Mark
[3] Park
[4] Pará
[5] fará
[6] faro
[7] firo
[8] fino
[9] fins
[10] fias
[11] aias
[12] asas
[13] ases
[14] aves
[15] avôs
[16] apôs

Valor médio dos diametros: 0.219251
```

Ficheiro com 5 letras

Comando:

1 veias	2 veias exume	3
Nota: Últimos 37 elementos. [6285] imuto [6286] imune [6287] antro [6288] entoo [6289] então [6290] entoe [6291] enjoa [6292] inste [6293] inale [6294] unira [6295] erigi [6296] eriço [6297] erice [6298] fá-lo [6299] fo-la [6300] imane [6301] astro [6302] enjoo [6303] Então [6304] estão [6305] enjoe [6306] unirá [6307] fo-lo [6308] imame [6309] emane [6310] estio [6311] emana [6312] emano [6313] espio [6314] Estio [6315] etano [6316] expio [6317] espia [6318] espie [6319] expia [6320] expie [6321] expõe	> 2 veias exume [1] veias [2] velas [3] nelas [4] nulas [5] aulas [6] atlas [7] atias [8] afias [9] afins [10] afina [11] afila [12] axila [13] exila [14] exile [15] exime [16] exume	índice = 7981 -> urgis índice = 7981 -> asila índice = 7981 -> aspas índice = 7982 -> soava índice = 7982 -> marés índice = 7982 -> apraz índice = 7982 -> laçai índice = 7984 -> anila índice = 7984 -> furem índice = 7985 -> ungis índice = 7986 -> greto índice = 7988 -> suava índice = 7989 -> vento índice = 7989 -> cabia índice = 7989 -> gueto índice = 7992 -> bingo índice = 7992 -> afila índice = 7994 -> aduba índice = 7995 -> tapir índice = 7995 -> pulse índice = 7995 -> coral índice = 7995 -> cofio índice = 7996 -> preme

Comando:

```
> 4


Tamanho da Hash table: 8000
Número de entradas: 7166
Tamanho médio das Heads com Heads vazias: 0.895750
Tamanho médio das Heads sem Heads vazias: 1.710671
Tamanho da maior Linked List: 8
Tamanho da menor Linked List: 1
```

```
> 5
Número de vértices: 7166
Número de arestas: 23446

Número de componentes conexas: 575

Tamanho mais pequeno de uma componente: 0
Tamanho maior de uma componente: 6320

Valor médio dos tamanhos das componentes: 12.462608


largest: 33

Largest Ladder:
[1] expõe
[2] expie
[3] expio
[4] espio
[5] estio
[6] estão
[7] então
[8] entro
[9] entra
[10] extra
[11] exara
[12] exala
[13] exila
[14] axila
[15] afila
[16] afira
[17] atira
[18] ativa
[19] atava
[20] alava
[21] flava
[22] fiava
[23] fiara
[24] fibra
[25] vibra
[26] viera
[27] viela
[28] vi-la
[29] vê-la
[30] dê-la
[31] dá-la
[32] fá-la
[33] fo-la
[34] fo-lo

Valor médio dos diametros: 0.365217
```

Ficheiro com 6 letras

1 ousado	2 ousado remela	3
Nota: Últimos 37 elementos. [11577] Viçosa [11578] piroga [11579] porosa [11580] enviou [11581] enfiou [11582] soroso [11583] moroso [11584] piloto [11585] picoto [11586] penoso [11587] venosa [11588] aprose [11589] aprovo [11590] apreso [11591] apresa [11592] aprova [11593] morosa [11594] seroso [11595] mofoso [11596] pilote [11597] pilota [11598] picote [11599] picota [11600] penosa [11601] aprove [11602] aprese [11603] apreço [11604] apreça [11605] seboso [11606] sedoso [11607] serosa [11608] pinote [11609] pelote [11610] pelota [11611] pacote [11612] aprece [11613] zelote	> 2 ousado remela [1] ousado [2] ousada [3] ougada [4] sugada [5] segada [6] regada [7] regala [8] regela [9] remela	índice = 15982 -> marrou índice = 15982 -> marido índice = 15982 -> apagam índice = 15983 -> posado índice = 15983 -> mureis índice = 15985 -> tireis índice = 15985 -> divide índice = 15986 -> manias índice = 15988 -> astuto índice = 15989 -> poisio índice = 15989 -> estude índice = 15990 -> pojado índice = 15990 -> listai índice = 15991 -> arreto índice = 15992 -> podado índice = 15992 -> doasse índice = 15992 -> ecluda índice = 15993 -> mudeis índice = 15996 -> doesse índice = 15999 -> macias índice = 15999 -> lodosa

Comando:

```
> 4


Tamanho da Hash table: 16000
Número de entradas: 15654
Tamanho médio das Heads com Heads vazias: 0.978375
Tamanho médio das Heads sem Heads vazias: 1.597836
Tamanho da maior Linked List: 7
Tamanho da menor Linked List: 1
```

```
> 5
Número de vértices: 15654
Número de arestas: 36204

Número de componentes conexas: 1929

Tamanho mais pequeno de uma componente: 0
Tamanho maior de uma componente: 11612

Valor médio dos tamanhos das componentes: 8.115086


largest: 57
```

```
Valor médio dos diametros: 0.666667
```

```
Largest Ladder:
[1] enruga
[2] enxuga
[3] enxuta
[4] enluta
[5] enlata
[6] engata
[7] engana
[8] encana
[9] encena
[10] enceta
[11] enceto
[12] exceto
[13] expeto
[14] espeto
[15] aspeto
[16] asseto
[17] assedo
[18] assado
[19] assada
[20] ossada
[21] ousada
[22] ourada
[23] curada
[24] corada
[25] cotada
[26] citada
[27] citava
[28] oitava
[29] optava
[30] opiava
[31] opiada
[32] opiado
[33] odiado
[34] adiado
[35] adindo
[36] alindo
[37] alando
[38] amando
[39] amanso
[40] amanse
[41] amasse
[42] alasse
[43] classe
[44] coasse
[45] doasse
[46] doa-se
[47] roa-se
[48] ria-se
[49] rir-se
[50] vir-se
[51] virose
[52] viroso
[53] piroso
[54] piloso
[55] piloto
[56] pilote
[57] pelote
[58] zelote
```

Ficheiro maior

1 Portugal	2 Pina olha	3

`> 1 Portugal`
`[1] Portugal`
	`> 2 Pina olha`
	`[1] Pina`
	`[2] tina`
	`[3] tino`
	`[4] pino`
	`[5] pito`
	`[6] puto`
	`[7] auto`
	`[8] alto`
	`[9] alho`
	`[10] olho`
	`[11] olha`
		`índice = 1023982 -> obtinha-me`
		`índice = 1023982 -> colcheteá-lo`
		`índice = 1023982 -> empantanar-vos`
		
		`índice = 1023983 -> adoentá-lo-emos`
		
		`índice = 1023985 -> reaverias`
		
		`índice = 1023988 -> desenvencilhá-lo-emos`
		
		`índice = 1023989 -> desmazelar-nos-ias`
		
		`índice = 1023990 -> adiar-nos-emos`
		
		`índice = 1023992 -> reencarcerassem`
		
		`índice = 1023994 -> reativante`
		
		`índice = 1023997 -> arredariam`
		
		`índice = 1023999 -> sentenciadora`
		`índice = 1023999 -> mingue-lhe`

Comandos:

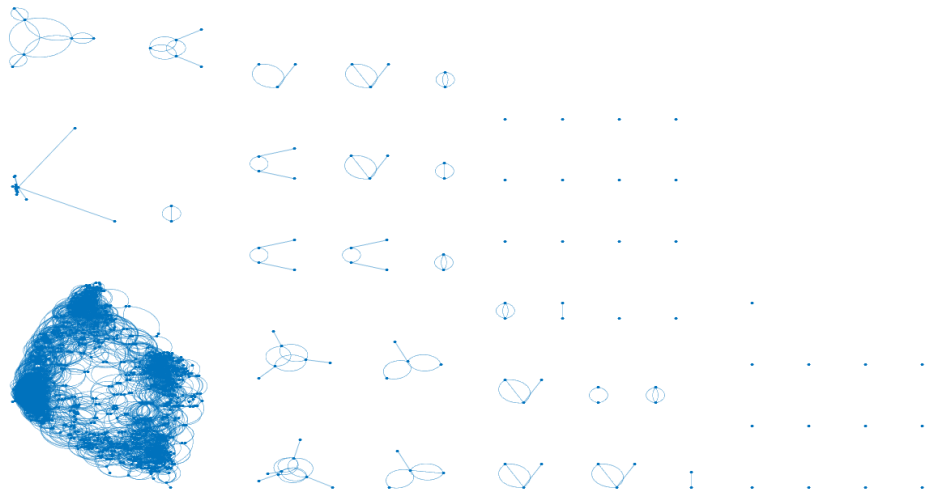
```
> 4


Tamanho da Hash table: 1024000
Número de entradas: 999282
Tamanho médio das Heads com Heads vazias: 0.975861
Tamanho médio das Heads sem Heads vazias: 1.567042
Tamanho da maior Linked List: 9
Tamanho da menor Linked List: 1
```

```
> 5
Número de vértices: 999282
Número de arestas: 1060534

Número de componentes conexas: 377234

Tamanho mais pequeno de uma componente: 0
Tamanho maior de uma componente: 16697

Valor médio dos tamanhos das componentes: 2.648971


largest: 92
```

```
Valor médio dos diametros: 0.798374
```

```
Largest Ladder
[1] empaleou
[2] empalhou
[3] espalhou
[4] espelhou
[5] espelhos
[6] espelhas
[7] espalhas
[8] espalhar
[9] espaldar
[10] escaldar
[11] escaldas
[12] escalpas
[13] escarpas
[14] escarnas
[15] encarnas
[16] encornas
[17] enfornas
[18] enformas
[19] enfermas
[20] enfermos
[21] enfeemos
[22] enteemos
[23] entremos
[24] extremos
[25] exaremos
[26] exaramos
[27] exarados
[28] exaradas
[29] exararas
[30] exarares
[31] exalares
[32] exilares
[33] axilares
[34] afilares
[35] afinares
[36] afanares
[37] afamares
[38] aramares
[39] tramares
[40] tremares
[41] tremeres
[42] premeres
[43] preferes
[44] preferis
[45] proferis
[46] proferia
[47] proveria
[48] provaria
[49] privaria
[50] crivaria
[51] cravaria
[52] chavaria
[53] chaparia
[54] chuparia
[55] chutaria
[56] coutaria
[57] contaria
[58] tontaria
[59] tostaria
[60] bostaria
[61] bastaria
[62] bastarda
[63] bastardo
[64] bastando
[65] pastando
[66] passando
[67] lassando
[68] lascando
[69] rascando
[70] riscando
[71] discando
[72] distando
[73] distendo
[74] distenso
[75] dispenso
[76] dispensa
[77] despensa
[78] despenda
[79] descenda
[80] descende
[81] descente
[82] desconte
[83] desponte
[84] desponto
[85] desporto
[86] desperto
[87] desperte
[88] despe-te
[89] deste-te
[90] veste-te
[91] viste-te
[92] visse-te
[93] visse-me
```

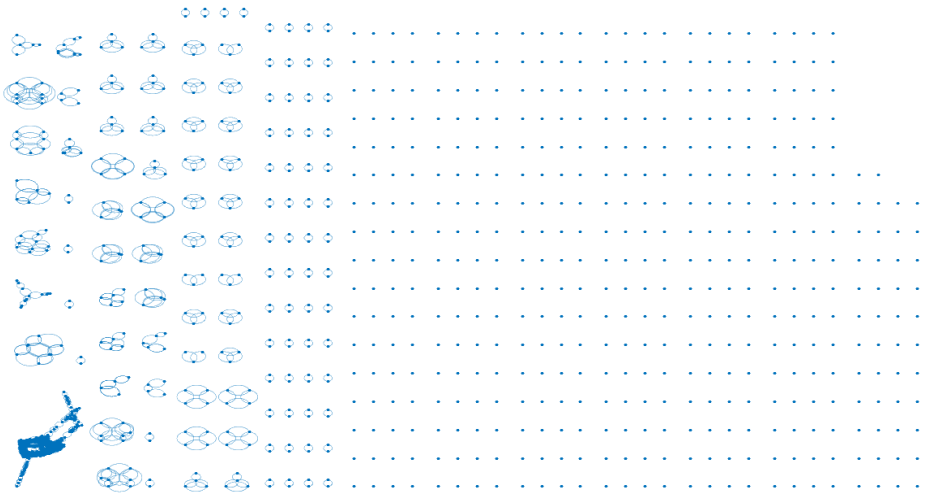
Grafos Ficheiro



Grafo exemplo do *wordlist-four-letters*

```
==18942==
==18942== HEAP SUMMARY:
==18942==     in use at exit: 0 bytes in 0 blocks
==18942==   total heap usage: 23,033 allocs, 23,033 frees, 143,231,936 bytes allocated
==18942==
==18942== All heap blocks were freed -- no leaks are possible
==18942==
==18942== For lists of detected and suppressed errors, rerun with: -s
==18942== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

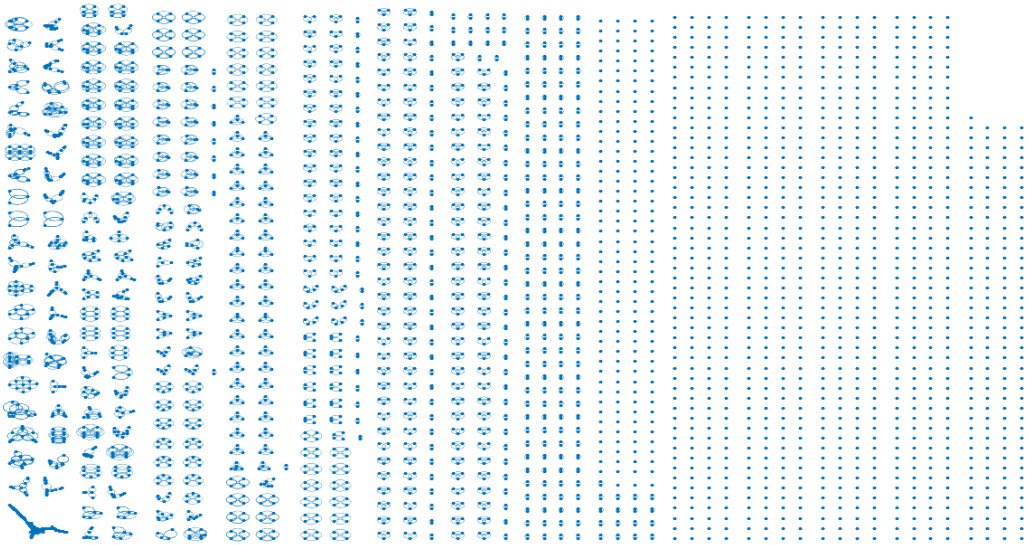
Não houve perda de memória para o ficheiro de quatro letras.



Grafo exemplo do *wordlist-five-letters*

```
==18972==
==18972== HEAP SUMMARY:
==18972==     in use at exit: 0 bytes in 0 blocks
==18972==   total heap usage: 61,820 allocs, 61,820 frees, 1,168,037,464 bytes allocated
==18972==
==18972== All heap blocks were freed -- no leaks are possible
==18972==
==18972== For lists of detected and suppressed errors, rerun with: -s
==18972== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Não houve perda de memória para o ficheiro de cinco letras.



Grafo exemplo do *wordlist-five-letters.*

```
==18997==
==18997== HEAP SUMMARY:
==18997==     in use at exit: 0 bytes in 0 blocks
==18997==   total heap usage: 105,671 allocs, 105,671 frees, 3,065,117,600 bytes allocated
==18997==
==18997== All heap blocks were freed -- no leaks are possible
==18997==
==18997== For lists of detected and suppressed errors, rerun with: -s
==18997== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Não houve perda de memória para o ficheiro de seis letras.

Apêndice

Código em MatLab

```
clear
clc
data = readtable('graphFour.txt','ReadVariableNames',false);
nodes = importdata("wordlist-four-letters.txt");
G = graph();
tamanho_data = height(data);
G = addnode(G,nodes);
for x = 1:tamanho_data
   first = data{x,1};
   second = data{x,2};
    G = addedge(G, first , second);
end
plot(G);
```

Código em C

```
#include <stdio.h>
```

```
#include <stdlib.h>
#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 th enext adjacency list node
 hash_table_node_t *vertex;         // the other 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
 // 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
};

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);
}
unsigned int crc32(const char *str)
{
 static unsigned 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);
 }

 hash_table->hash_table_size = 250;
 hash_table->heads = (hash_table_node_t **)malloc(sizeof(hash_table_node_t *) *
hash_table->hash_table_size);
 hash_table->number_of_entries = 0;
 hash_table->number_of_edges = 0;
 for(i = 0;i < hash_table->hash_table_size;i++)
   hash_table->heads[i] = NULL;
 return hash_table;
}


static void hash_table_grow(hash_table_t *hash_table)
{
 hash_table_node_t **old_heads, **new_heads, *node, *next;
 unsigned int old_size, i;

 // save a pointer to the old array of linked list heads and its size
 old_heads = hash_table->heads;
 old_size = hash_table->hash_table_size;

 // create a new hash table with a larger size
 hash_table->hash_table_size *= 2;
 new_heads = (hash_table_node_t **)malloc(hash_table->hash_table_size *
sizeof(hash_table_node_t *));
 // check for out of memory
 for (i = 0u; i < hash_table->hash_table_size; i++)
   new_heads[i] = NULL;

 if (new_heads == NULL)
```

```
 {
   fprintf(stderr, "hash_table_grow: out of memory");
   exit(1);
 }

 // run the hash function for old values with new size

 for (i = 0u; i < old_size; i++)
 {
   node = old_heads[i];
   while (node != NULL)
   {

     next = node->next;

     size_t index = crc32(node->word) % hash_table->hash_table_size;
     node->next = new_heads[index];
     new_heads[index] = node;

     node = next;
   }
 }
 free(old_heads);
 hash_table->heads = new_heads;

}

static void hash_table_free(hash_table_t *hash_table)
{
 hash_table_node_t *node;
 hash_table_node_t *temp;
 adjacency_node_t *adj_node;
 adjacency_node_t *temp_adj;
```

```
 unsigned int i;

 for (i = 0; i < hash_table->hash_table_size; i++)
 {
   node = hash_table->heads[i];
   while (node != NULL)
   {
     temp = node;
     adj_node = node->head;
     while (adj_node != NULL)
     {
       temp_adj = adj_node;
       adj_node = adj_node->next;
       free(temp_adj);
     }
     node = node->next;
     free(temp);
   }
 }

 free(hash_table->heads);

 free(hash_table);
}


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;
 unsigned int i;

 // printf("word: %s\n", word);
```

```
 i = crc32(word) % hash_table->hash_table_size;
 node = hash_table->heads[i];
 while (node != NULL)
 {
   if (strcmp(node->word, word) == 0)
     return node;
   node = node->next;
 }

 if (insert_if_not_found && strlen(word) < _max_word_size_)
 {
   node = allocate_hash_table_node();
   strncpy(node->word, word, _max_word_size_);
   node->representative = node;
   node->next = hash_table->heads[i];
   node->previous = NULL;
   node->number_of_edges = 0;
   node->number_of_vertices = 1;
   node->visited = 0;
   node->head = NULL;
   hash_table->heads[i] = node;
   hash_table->number_of_entries++;
   if (hash_table->number_of_entries > hash_table->hash_table_size)
     hash_table_grow(hash_table);

   return node;
 }
  return NULL;
}

static void hash_stats(hash_table_t *hash_table){
 printf("\n\nTamanho da Hash table: %u\n", hash_table->hash_table_size);
 printf("Número de entradas: %u\n", hash_table->number_of_entries);
```

```
 printf("Tamanho médio das Heads com Heads vazias: %f\n",(float)hash_table->number_of_entries
/ hash_table->hash_table_size);


 hash_table_node_t *node;
 unsigned int MaxHead = 0;
 unsigned int MinHead = _max_word_size_;
 unsigned int conta = 0;
 for(unsigned int i = 0u;i < hash_table->hash_table_size;i++){
   unsigned int sizeHeads=0;


   for(node = hash_table->heads[i];node != NULL;node = node->next){
     sizeHeads++;
   }


   if (sizeHeads != 0)
   {
     conta++;
   }



   if( sizeHeads > MaxHead){
     MaxHead = sizeHeads;
   }
   if( sizeHeads < MinHead && sizeHeads != 0){
     MinHead = sizeHeads;
   }


 }
 printf("Tamanho médio das Heads sem Heads vazias: %f\n",(float)hash_table->number_of_entries
/ conta);
 printf("Tamanho da maior Linked List: %u\n", MaxHead);
 printf("Tamanho da menor Linked List: %u\n\n", MinHead);
}
```

```
//
// 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;

 hash_table_node_t *node_atual;
 for (representative = node; representative != representative->representative; representative
= representative->representative)
 ;

 for (node_atual = node; node_atual != representative; node_atual = next_node)
 {
   next_node  = node_atual->representative;
   node_atual->representative = representative;
 }

 return representative;

}

static void add_edge(hash_table_t *hash_table, hash_table_node_t *from, const char *word)
{
 hash_table_node_t *to, *from_representative, *to_representative;

 from_representative = find_representative(from);
 to = find_word(hash_table, word, 0);

 if (to == from)
   return;
```

```
 to_representative = find_representative(to);


 if (from_representative == to_representative)
 {
   from_representative->number_of_vertices++;
 }else{
   from_representative->representative = to_representative; //mete o representante do from
igual ao do to
   to_representative->number_of_vertices += from_representative->number_of_vertices; //soma o
numero de vertices do from ao do to
   to_representative->number_of_edges += from_representative->number_of_edges; //soma o numero
de arestas do from ao do to
 }


 adjacency_node_t *linkfrom = allocate_adjacency_node();
 adjacency_node_t *linkto = allocate_adjacency_node();


 linkfrom->vertex = to;
 linkfrom->next = from->head;
 from->head = linkfrom;


 linkto->vertex = from;
 linkto->next = to->head;
 to->head = linkto;


 from_representative->number_of_edges++;
 to_representative->number_of_edges++;
 hash_table->number_of_edges++;


  return;
}


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){

       //+ rapido
       if (find_word(hash_table,new_word,0)!=NULL)
       {
         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)
{
```

```
// Initialize the queue
int front = 0, back = 0;
list_of_vertices[back++] = origin;

// Set the origin vertex as visited
origin->visited = 1;
origin->previous = NULL;
// Perform the search

int finish = 0;

while (front < back)
{
  // Dequeue the front vertex from the queue
  hash_table_node_t *vertex = list_of_vertices[front++];
  // Check if the vertex is the goal
  if (finish == 1)
  {
    break;
  }



  // Enqueue the adjacent vertices
  adjacency_node_t *adjacent = vertex->head;
  while (adjacent != NULL)
  {

    if (adjacent->vertex->visited == 0)
      {
        list_of_vertices[back++] = adjacent->vertex;
        adjacent->vertex->visited = 1;
        adjacent->vertex->previous = vertex;
```

```
        // short_path++;

        if (adjacent->vertex == goal)
        {
          finish = 1;
          break;
        }
       }
     adjacent = adjacent->next;
   }
 }

 for (int i = 0; i < back; i++)
 {
   list_of_vertices[i]->visited = 0;
 }

 return back;
}

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

static void list_connected_component(hash_table_t *hash_table,const char *word)
{

 hash_table_node_t *node;

 // encontramos o nó que correspode à variavel dada no argumento
 node = find_word(hash_table, word, 0);

 if (node == NULL)
```

```
 {
   return;
 }

 // encontramos o resprensentativo dessa palavra
 hash_table_node_t *representative = find_representative(node);

 int Nr_max = representative->number_of_vertices;
 hash_table_node_t **list_of_vertices = (hash_table_node_t **)malloc(Nr_max *
sizeof(hash_table_node_t *));

 int nrPalavras = breadh_first_search(Nr_max, list_of_vertices, node, NULL);

 int count = 1;
 for(int i = 0;i < nrPalavras;i++){
   printf("[%d] %s\n",count,list_of_vertices[i]->word);
   count++;
 }

 free(list_of_vertices);
}

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

static int largest_diameter;  //variavel para guardar o maior diametro de entre todas as
componentes
static hash_table_node_t **largest_diameter_example; // array que guarda os nós do maior
caminho
static int Sum_Diameters, Num_Diameters; // variaveis para calcular a média dos caminhos

static int connected_component_diameter(hash_table_node_t *node)
{
```

```
 int Nr_max = node->number_of_vertices;


 hash_table_node_t **list_of_vertices = (hash_table_node_t **)malloc(Nr_max *
sizeof(hash_table_node_t *));
 int nrPalavras = breadh_first_search(Nr_max, list_of_vertices, node, NULL);
 int diametro = 0;


 for(int i = 0;i < nrPalavras;i++){ //----- 1 -----//


   hash_table_node_t **list_of_vertices_Dentro = (hash_table_node_t **)malloc(Nr_max *
sizeof(hash_table_node_t *));
   //----- 2 -----//
   int dist = breadh_first_search(Nr_max, list_of_vertices_Dentro, list_of_vertices[i], NULL);


   hash_table_node_t *path = list_of_vertices_Dentro[dist-1];
   int realDist = 0;
   //----- 3 -----//
   while (path != NULL)
   {
     realDist++;
     path = path->previous;
   }
   //----- 4 -----//
   if (realDist > diametro)
   {
     diametro = realDist;
   }
   //----- 5 -----//
   if (realDist > largest_diameter)
   {
     largest_diameter = realDist;
     hash_table_node_t *palav = list_of_vertices_Dentro[dist-1];
     free(largest_diameter_example);
```

```
     largest_diameter_example = (hash_table_node_t **)malloc(largest_diameter *
sizeof(hash_table_node_t *));
     int i = 0;
     //----- 6 -----//
     while (palav != NULL)
     {
       largest_diameter_example[i] = palav;

       i++;
       palav = palav->previous;
     }
   }
   free(list_of_vertices_Dentro);
   free(path);
 }
 //----- 7 -----//
 Sum_Diameters = Sum_Diameters + diametro-1;
 Num_Diameters++;
 free(list_of_vertices);
 return largest_diameter;
}

static int find_connected_component_representatives(hash_table_t *hash_table,
hash_table_node_t **representatives)
{
 int index = 0;

 for (int i = 0; i < hash_table->hash_table_size; i++)
 {
   for (hash_table_node_t *vertex = hash_table->heads[i]; vertex != NULL; vertex =
vertex->next)
   {
     hash_table_node_t *representative = find_representative(vertex);
```

```
      if (!representative->visited)
      {
        representatives[index++] = representative->word;
        representative->visited = 1;
      }
    }
  }

  // tornamos o estado dos vertices que estavam visitados para 0
  for (int i = 0; i < hash_table->hash_table_size; i++)
  {
    for (hash_table_node_t *vertex = hash_table->heads[i]; vertex != NULL; vertex =
vertex->next)
    {
      vertex->visited = 0;
    }
  }

  return index;
}

//
// 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)
{
  hash_table_node_t *fromNode = find_word(hash_table, from_word, 0);
  hash_table_node_t *toNode = find_word(hash_table, to_word, 0);

  if(fromNode == NULL || toNode == NULL){
    return;
```

```
}

hash_table_node_t *fromRep = find_representative(fromNode);
hash_table_node_t *toRep = find_representative(toNode);

if(fromRep != toRep){
  return;
}

hash_table_node_t **list_of_vertices = malloc(sizeof(hash_table_node_t *) *
fromRep->number_of_vertices);


for (int i = 0; i < fromRep->number_of_vertices; i++)
{
  list_of_vertices[i]= NULL;
}

if (list_of_vertices == NULL)
{
  exit(1);
}

int goal_indice = breadh_first_search(0, list_of_vertices,fromNode, toNode);

char short_path[goal_indice-1][_max_word_size_];

hash_table_node_t *p = list_of_vertices[goal_indice-1];

int count = 0;
while (p != NULL)
{
  strcpy(short_path[count], p->word);
```

```
    count++;
    p = p->previous;
  }

  int indice2 = 1;
  for (int i = count-1; i >= 0; i--)
  {
    printf("[%d] %s\n", indice2, short_path[i]);
    indice2++;
  }

  free(list_of_vertices);
  // free(fromNode);
  // free(toNode);
  // free(toRep);
  // free(fromRep);
  // free(p);
}

//
// some graph information (optional)
//

static void graph_info(hash_table_t *hash_table)
{
  hash_table_node_t **representatives = malloc(sizeof(hash_table_node_t *) *
hash_table->hash_table_size);
  int nrRepresentative = find_connected_component_representatives(hash_table, representatives);
  largest_diameter = 0;
  int distMaior;

  Sum_Diameters = 0;
  Num_Diameters = 0;
```

```
  for (int i = 0; i < nrRepresentative; i++)
 {
   distMaior = connected_component_diameter(representatives[i]);
 }

 free(representatives);
}



int main(int argc,char **argv)
{
 char word[100],from[100],to[100];
 hash_table_t *hash_table;
 hash_table_node_t *node;
 unsigned int i;
 int command;
 FILE *fp;

 // initialize hash table
 hash_table = hash_table_create();
 // 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);
 }
 while(fscanf(fp,"%99s",word) == 1)
   (void)find_word(hash_table,word,1);
 fclose(fp);
 // 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            (print words )\n");
  fprintf(stderr,"  4            (statistics hash )\n");
  fprintf(stderr,"  5            (graf info )\n");
  fprintf(stderr,"  6            (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 == 6)
      break;

    else if (command == 3)
    {

      for(i = 0u;i < hash_table->hash_table_size;i++)
      {
        for(node = hash_table->heads[i];node != NULL;node = node->next)
        {
          printf("índice = %u -> %s\n",i,node->word);
        }
        printf("\n");
      }
    }
    else if (command == 4)
    {
      hash_stats(hash_table);
    }
    else if (command == 5)
    {

      printf("Número de vértices: %d\n", hash_table->number_of_entries);
      printf("Número de arestas: %d\n\n", hash_table->number_of_edges);

      hash_table_node_t **representatives = malloc(sizeof(hash_table_node_t *) *
hash_table->hash_table_size);

      int nrRepresentative = find_connected_component_representatives(hash_table,
representatives);
      printf("Número de componentes conexas: %d\n\n", nrRepresentative);
```

```
    int size_small_comp = 10;
    int size_large_comp = 0;
    int sum_size_comp = 0;

    for (int i = 0; i < nrRepresentative; i++)
    {
      int Nr_max = representatives[i]->number_of_vertices;
      hash_table_node_t **list_of_vertices = (hash_table_node_t **)malloc(Nr_max *
sizeof(hash_table_node_t *));

      int nrPalavras = breadh_first_search(Nr_max, list_of_vertices, representatives[i],
NULL);

      sum_size_comp = sum_size_comp + nrPalavras;

      if (size_large_comp < nrPalavras)
      {
        size_large_comp = nrPalavras;
      }

      if (size_small_comp > nrPalavras)
      {
        size_small_comp = nrPalavras;
      }

      free(list_of_vertices);
    }
    printf("Tamanho mais pequeno de uma componente: %d\n", size_small_comp-1);
    printf("Tamanho maior de uma componente: %d\n", size_large_comp-1);
```

```
    printf("\nValor médio dos tamanhos das componentes: %f \n\n",
(float)sum_size_comp/nrRepresentative);


    printf("\nlargest: %d\n\n", largest_diameter-1);
    printf("Largest Ladder:\n");
    for (int i = 0; i < largest_diameter; i++)
    {
      printf("[%d] %s\n", i+1, largest_diameter_example[i]->word);
    }

    printf("\nValor médio dos diametros: %f \n\n", (float)Sum_Diameters / Num_Diameters);


    free(representatives);
  }
}

free(largest_diameter_example);
hash_table_free(hash_table);
return 0;
}
```

Conclusão

Em conclusão, ficamos a perceber melhor como realizar trabalhos e problemas que se enquadrem na temática de Hashing" e também mais sobre algoritmos que nos vão ser muito necessários mais tarde na nossa progressão académica. Assim, agradecemos aos professores que nos ajudaram a realizar este trabalho e também ficamos agradecidos

pelo facto de termos enriquecido mais com todo o trabalho realizado ao longo da disciplina.