WORD LADDER

Algoritmos e Estruturas de Dados

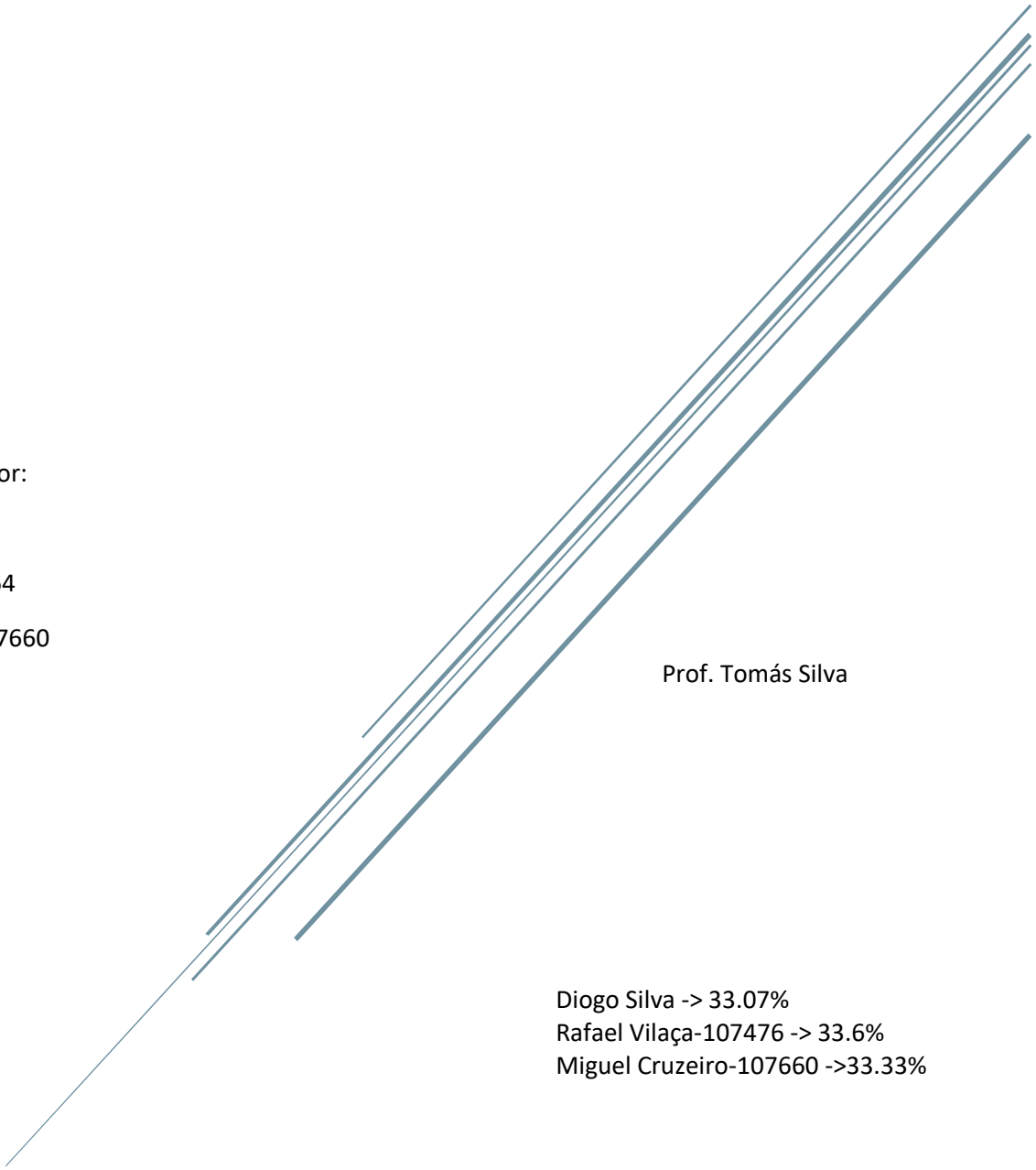
Trabalho realizado por:

Diogo Silva - 107647

Rafael Vilaça - 107764

Miguel Cruzeiro - 107660

Prof. Tomás Silva



Diogo Silva -> 33.07%
Rafael Vilaça-107476 -> 33.6%
Miguel Cruzeiro-107660 ->33.33%

Conteúdo

Introdução

Como objetivo deste segundo trabalho prático pretendia-se simular uma word ladder.

A word ladder é uma sequência de palavras em que duas palavras adjacentes diferem uma da outra por uma letra. Por exemplo, em português pode-se chegar de nada a tudo em 4 passos: tudo → todo → nodo → nado → nada.

Ainda são fornecidos ficheiros com palavras em português e um código fonte, word_ladder.c, que deve der completo seguindo as seguintes indicações:

- Completar código da hash table(Obrigatório); (feito)
- Construir um grafo (including union-find data); (feito)
- Implementar breadth-first search no grafo; (incompleto (não funcional))
- Listar todas as palavras de uma "connected component"; (feito)
- Encontrar o caminho mais curto entre duas palavras; (por fazer)
- Calcular o diâmetro de uma "connected component" e listar a cadeia de palavras mais longa; (por fazer)
- Mostrar algumas estatísticas sobre o grafo; (incompleto)
- Testar memory leaks. (feito)

Implementação do Problema

Porquê Hash Tables?

Hash Tables são uma estrutura de dados que associa chaves de pesquisa a valores. Estas chaves são um array de caracteres e, ao serem passadas por uma função de dispersão, é gerado o índice correspondente no array representativo da tabela, onde se vai guardar a informação pretendida. Assim, com o uso de Hash Tables, a pesquisa de informação torna-se mais rápida e eficiente.

Hash Table

O primeiro problema proposto foi completar o código que corresponde á implementação da Hash Table.

Função hash_table_create

Nesta função é criada a Hash Table e para isso é alocada memória para criar a estrutura de dados Hash Table. Esta, é criada inicialmente com tamanho inicial de 100 e com 0 entradas e 0 arestas. Ainda, são atribuídos aos 100 espaços na HashTable o valor NULL.

Por fim, são acrescentadas algumas mensagens de erro para o caso de não haver espaço na memória para criar a hash table e para o caso de não haver espaço para acrescentar mais alguma palavra.

```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);
  }
  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 *));
  if (hash_table->heads == NULL)
  {
    fprintf(stderr, "create_hash_table: out of memory\n");
    exit(1);
  }
  printf("hash table size = %u\n", hash_table->hash_table_size);
  return hash_table;
}
```

Função hash_table_grow

Esta função é utilizada para aumentar o tamanho da hash table ao atingir um certo numero de entradas. Para começar, as palavras (heads) que estavam na hash table antiga são armazenadas numa variável. Para aumentar o tamanho de uma hash table vai ser preciso calcular o novo tamanho da mesma que neste caso vai ser igual a old_size+old_size/2. De seguida é atribuído o novo espaço na memória e todas as heads da hash table são atualizadas para NULL.

Por fim as palavras que tinham sido guardadas previamente na variável old_heads são introduzidas novamente na hash table agora com o novo tamanho.

Estando esse processo concluído é então libertado o espaço na memoria onde a variável old_heads se situava.

O novo tamanho da hashtable vai ser old_size +old_size/2.

```c
static void hash_table_grow(hash_table_t *hash_table)
{

  hash_table_node_t **old_heads = hash_table->heads;
  hash_table_node_t *n2;
  hash_table_node_t *n1;
  int old_size = hash_table->hash_table_size;

  hash_table->hash_table_size = old_size + old_size / 2;
  hash_table->heads = (hash_table_node_t **)malloc(hash_table->hash_table_size * sizeof(hash_table_node_t *));
  for (int i = 0; i < hash_table->hash_table_size; i++)
  {
    hash_table->heads[i] = NULL;
  }
  for (int i = 0; i < old_size; i++)
  {
    for (n1 = old_heads[i]; n1 != NULL; n1 = n2)
    {
      n2 = n1->next; // importante nao atribuir em cima porque o valor é modificado abaixo
      int j = crc32(n1->word) % hash_table->hash_table_size;
      n1->next = hash_table->heads[j];
      hash_table->heads[j] = n1;
    }
  }
  free(old_heads);
}
```

Função hash_table_free

Esta função serve para libertar o espaço de memória utilizado pela hash table. Para isso são percorridas todas as heads e para cada head é encontrada a palavra seguinte (guardamos a posição na memória da palavra seguinte) até a palavra seguinte ter como valor NULL, depois é apagada a palavra atual e a lista das suas adjacências da memória, utilizando o mesmo método referido acima onde, primeiramente, é encontrada a head da lista de adjacencia e, guardado o endereço na memória da palavra seguinte até não haver mais nenhuma palavra, ou seja , até a palavra seguinte ter o valor NULL. Este processo é realizado para todas as palavras na hash table. Por fim, é libertado o espaço de memória reservado previamente para as heads e a hash table.

```c
static void hash_table_free(hash_table_t *hash_table)
{
  hash_table_node_t *n1;
  hash_table_node_t *n2;
  adjacency_node_t *an1;
  adjacency_node_t *an2;

  for (int i = 0; i < hash_table->hash_table_size; i++)
  {
    n1 = hash_table->heads[i];
    for (n1 = hash_table->heads[i]; n1 != NULL; n1 = n2)
    {
      n2 = n1->next;
      for (an1 = n1->head; an1 != NULL; an1 = an2)
      {
        an2 = an1->next;
        free_adjacency_node(an1);
      }
      free_hash_table_node(n1);
    }
  }
  free(hash_table->heads);
  free(hash_table);
}
```

Função find_word

```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;
  hash_table_node_t *node2;
  unsigned int i;
  // testar
  i = crc32(word) % hash_table->hash_table_size;
  for (node = hash_table->heads[i]; node != NULL; node = node2)
  {
    if (strcmp(node->word, word) == 0)
    {
      break;
    }
    node2 = node->next;
  }
  if (insert_if_not_found)
  {
    node = allocate_hash_table_node();
    hash_table->number_of_entries++;
    node->head = NULL;
    node->previous = NULL;
    node->number_of_edges = 0;
    node->number_of_vertices = 0;
    node->representative = node;
    strcpy(node->word, word);
    node->next = hash_table->heads[i];
    hash_table->heads[i] = node;
    if (hash_table->number_of_entries > 2 * (hash_table->hash_table_size))
    {
      hash_table_grow(hash_table);
      printf("hash table size = %u entries = %u\n", hash_table->hash_table_size, hash_table->number_of_entries);
    }
  }
  return node;
}
```

Esta função serve para encontrar uma determinada palavra na hash table e caso o mesmo não se verifique é introduzida na hash table apenas se o valor da variável insert_not_found for 1.

Para isso, é primeiramente verificado se a palavra já existe na hash table, percorrendo a mesma utilizando um for loop onde ao usar a função strcmp() para comparar duas palavras (a desejada e a atual na hash table) e é dada como terminada a função e retornado o node a que a palavra desejada pertence. Se a palavra não for encontrada, a mesma será acrescentada à hash table , para tal é alocado o node (para a palavra) incrementado o numero de entradas da hash table e atualizadas as características default do node. Por fim é inserida a palavra no node com o strcpy e atualizado o seguinte do mesmo para a head com o hashcode da palavra e essa mesma head é de seguida atualizada para o node.

Finalmente verificando-se um número de entradas duas vezes maior que o tamanho da hash table é chamada a função hash_table_grow para aumentar o tamanho da mesma.

Função find_representative

```c
static hash_table_node_t *find_representative(hash_table_node_t *node)
{
  hash_table_node_t *n1,*n2, *n3;

  for(n1 = node;n1->representative != n1; n1 = n1->representative ) //se o representante for ele proprio, entao ele é o representante
    ;
  for(n2 = node; n2 != n1; n2 = n3) //enquanto o n2 for diferente do n1, o n3 vai ser o n2
  {
    n3 = n2->representative;
    n2->representative = n1;
  }

  return n1;
}
```

Esta função é utilizada para encontrar o representativo. A função começa a sua execução definindo o representante da palavra como ele mesmo de seguida é então efetuado outro loop onde é procurado todas as palavras com esse mesmo representante.

Função add_edge

Esta função serve para adicionar uma aresta. Temos uma palavra que vai ser procurada na hash table com a função find_word, verificando-se que a palavra não existe na hash table a função termina.

De seguida verifica-se se se já existe um link no node from para a palavra pedida, ou seja, se já a mesma já existe na lista de adjacência do from,

Não existindo um link entre as palavras é então necessário fazer um, o link é feito duas vezes uma vez para cada uma das palavras em cada uma é atualizado os atributos seguintes: link->vertex para a palavra que a atual vai estar ligada, o link->next para a head da palavra atual (head de uma palavra é a sua lista de adjacencia) e a head da palavra atual para o link isto é feito para cada uma das duas palavras.

Cada vez que a função é chamada é também incrementado o numero de edges (arestas) da hash table.

Finalmente é encontrado o representante de cada uma das palavras se forem iguais é incrementado o numero de arestas do representativo e se forem diferentes o representativo menor, ou seja o representativo com menos palavras na sua componente conexa é ligado ao maior e todas as suas características são transferidas para o maior representativo sendo apenas o número de arestas a soma dos dois somado com um devido a essa nova ligação para alem disso é também decrementado o numero de componentes conexas devido á menor se tornar parte da maior (os seus valores estatisticos também são atualizados para 0)

```c
static void add_edge(hash_table_t *hash_table, hash_table_node_t *from, const char *word)
{
  // from é a palavra que ja esta no hash table
  // word é a palavra que esta a ser lida

  hash_table_node_t *to, *from_representative, *to_representative;
  adjacency_node_t *link;

  to = find_word(hash_table, word, 0);

  if (to == NULL)
    return;

  for (link = from->head; link != NULL && link->vertex != to; link = link->next)
    ;

  if (link != NULL)
    return;

  link = allocate_adjacency_node();
  link->vertex = to->word;
  link->next = from->head;
  from->head = link;
  from->number_of_edges++;

  link = allocate_adjacency_node();
  link->vertex = from->word;
  link->next = to->head;
  to->head = link;
  to->number_of_edges++;

  hash_table->number_of_edges++;

  from_representative = find_representative(from);
  to_representative = find_representative(to);

  if (from_representative != to_representative)
  {
    if (from_representative->number_of_vertices < to_representative->number_of_vertices)
    {
      to_representative->number_of_vertices += from_representative->number_of_vertices;
      to_representative->number_of_edges += from_representative->number_of_edges + 1 ;
      from_representative->number_of_vertices = 0;
      from_representative->number_of_edges = 0;
      hash_table->num_components--;
      from_representative->representative = to_representative;
    }
    else
    {
      from_representative->number_of_vertices += to_representative->number_of_vertices;
      from_representative->number_of_edges += to_representative->number_of_edges + 1 ;
      to_representative->number_of_vertices = 0;
      to_representative->number_of_edges = 0;
      hash_table->num_components--;
      to_representative->representative = from_representative;
    }
  }
}
```

Função enqueue, dequeue e free_queue

Nesta função é alocado o espaço necessário para a queue que é uma estrutura de dados que segue a regra primeiro a entrar é o primeiro a sair. O enqueue serve para adicionar um elemento no fim do queue e para isso, primeiramente é verificado se o queue está cheio e é dado o valor 0 ao index do primeiro elemento e o index do último elemento é aumentado em 1 e essa vai ser a posição do elemento a ser adicionado.

Por sua vez o dequeue serve para remover um elemento do queue e para isso é verificado se o queue está vazio e é retirado o elemento que está na frente do queue e o index da frente do queue é aumentado por 1. Para o último elemento os valores do index do elemento da frente e de trás são retornados para -1.

Por fim, a função free_queue servirá para limpar o espaço da memória ocupado pela queue começando por desocupar todos os nodes existentes na mesma e no final liberta o espaço da queue completamente.

```c
//enqueue
static void enqueue(queue_t *queue, hash_table_node_t *node)
  {
    queue_node_t *queue_node = (queue_node_t *)malloc(sizeof(queue_node_t));
    queue_node->node = node;
    queue_node->next = NULL;
    if (queue->head == NULL)
      queue->head = queue_node;
    else
      queue->tail->next = queue_node;
    queue->tail = queue_node;
  }

  //dequeue
  hash_table_node_t *dequeue(queue_t *queue)
  {
    hash_table_node_t *node;
    queue_node_t *queue_node;

    if (queue->head == NULL)
      return NULL;
    queue_node = queue->head;
    queue->head = queue_node->next;
    if (queue->head == NULL)
      queue->tail = NULL;
    node = queue_node->node;
    free(queue_node);
    return node;
  }

  //free_queue
  static void free_queue(queue_t *queue)
  {
    queue_node_t *queue_node;

    while (queue->head != NULL)
    {
      queue_node = queue->head;
      queue->head = queue_node->next;
      free(queue_node);
    }
    free(queue);
  }
```

Função breadth_first_search

Nesta função é implementado o algoritmo breadth_first_search. Este algoritmo é utilizado para percorrer uma árvore ou grafo (no nosso caso) começando por um vértice que preenche um certo critério. Começa a procurar na raiz e depois vai para os vértices vizinhos que ainda não tenham sido visitados e continua assim até encontrar o pretendido. Sempre que passa por um vértice marca o mesmo como visitado.

Inicialmente é retirado o primeiro valor no queue, que será o vértice inicial e verifica-se se é o objetivo, se for é retornado number_of_vertices_visited (número de vértices por onde passou). Se não for e caso o vértice ainda não tenha sido visitado, ele é marcado como visitado e todos os vértices vizinhos são colocados no queue e number_of_vertices_visited é incrementado em 1.

```
517 static int breadth_first_search(int maximum_number_of_vertices, hash_table_node_t **list_of_vertices, hash_table_node_t *origin, hash_table_node_t *goal)
518 {
519   int i, j, k, n, number_of_vertices_visited = 0;
520   hash_table_node_t *node, *node2;
521   adjacency_node_t *adjacency_node;
522
523   // initialize queue
524   queue_t *queue = (queue_t *)malloc(sizeof(queue_t));
525   queue->head = NULL;
526   queue->tail = NULL;
527
528   // enqueue origin
529   enqueue(queue, origin);
530   // while queue is not empty
531   while (queue->head != NULL)
532   {
533     // dequeue node if not null
534     if((node = dequeue(queue)) == NULL)
535       break;
536
537     // if node is goal
538     if (node == goal)
539     {
540       // return number of vertices visited
541       return number_of_vertices_visited;
542     }
543
544     // if node is not visited
545     if (node->visited == 0)
546     {
547       // mark node as visited
548       node->visited = 1;
549
550       // enqueue all adjacent nodes
551       adjacency_node = node->head;
552       while (adjacency_node != NULL)
553       {
554         if(adjacency_node->vertex->visited == 0){
555           adjacency_node->vertex->previous = node;
556           enqueue(queue, adjacency_node->vertex);
557           adjacency_node = adjacency_node->next;
558         }
559       }
560
561       // increment number of vertices visited
562       number_of_vertices_visited++;
563     }
564     free_queue(queue);
565   }
566   //free queue
567   return -1;
568 }
```

Função list_connected_component

Esta função lista todos os vértices pertencentes a um mesmo componente conectado. Primeiramente é verificado se a word está na hash table e, caso não seja encontrada, é impressa uma mensagem de erro. Sabendo que a word está na hash table, são encontradas todas as outras palavras conectadas com word e impressas no terminal.

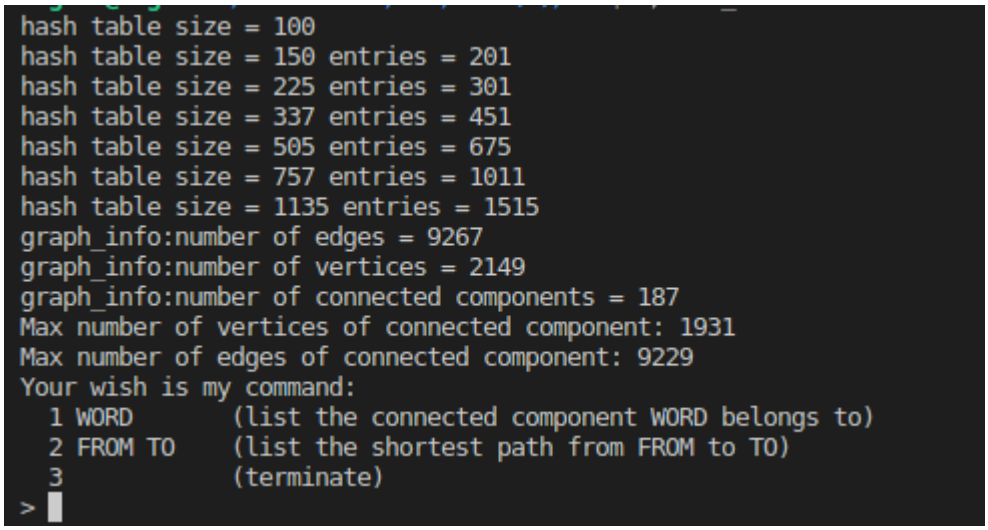
```c
static void list_connected_component(hash_table_t *hash_table, const char *word)
{
  hash_table_node_t *node;
  // find word in hash table
  node = find_word(hash_table, word, 0);
  if (node == NULL)
  {
    printf("list_connected_component: word not found\n");
    return;
  }

  adjacency_node_t *adjacency_node;
  adjacency_node = node->head;

  while (adjacency_node != NULL)
  {
  printf("%s \n", adjacency_node->vertex);
  //printf("%s \n", node->head->next->vertex);
  adjacency_node = adjacency_node->next;
  }


  /* while (node->next != NULL)
  {
    printf("%s \n", node->next);
    node = node->next;
  } */

}
```

Função graph_info

Nesta função são apresentados dados estatísticos tais são o número de arestas, vértices e componentes conexas do grafo e o número máximo vértices e arestas da componente conexa com o maior valor de cada um deles.

```
675  static void graph_info(hash_table_t *hash_table)
676  {
677    printf("graph_info:number of edges = %d \n", hash_table->number_of_edges); // resultado do stor 9265
678    printf("graph_info:number of vertices = %d \n", hash_table->number_of_entries); // resultado do stor 9917
679    printf("graph_info:number of connected components = %d \n", hash_table->num_components); // resultado do stor 187
680    hash_table_node_t *node;
681    hash_table_node_t *n1;
682    hash_table_node_t *n2;
683    hash_table_node_t *rep;
684    hash_table_node_t *rep_n;
685    int countedge = 0;
686    int count = 0;
687    for (int i = 0; i < hash_table->hash_table_size; i++)
688    {
689      for (n1 = hash_table->heads[i]; n1 != NULL; n1 = n2)
690      {
691        n2 = n1->next;
692        rep = find_representative(n1);
693        if(rep->number_of_vertices>count){
694          count = rep->number_of_vertices;
695        }
696        if(rep->number_of_edges>countedge){
697          countedge = rep->number_of_edges;
698        }
699      }
700    }
701    printf("Max number of vertices of connected component: %d \n", count);
702    printf("Max number of edges of connected component: %d \n", countedge);
703  }
```

Resultados Obtidos



Código

```c

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

typedef struct queue_s queue_t;
typedef struct queue_node_s queue_node_t;

struct queue_s
{
    queue_t *next; // link to the next queue node
    queue_t *prev; // link to the previous queue node
    queue_t *head; // head of the queue
    queue_t *tail; // tail of the queue
    hash_table_node_t *vertex; // the vertex
};

```

```c
struct queue_node_s
{
  queue_node_t *next; // link to the next queue node
  queue_node_t *prev; // link to the previous queue node
  queue_node_t *head; // head of the queue
  queue_node_t *tail; // tail of the queue
  queue_node_t *node; // the queue node
  hash_table_node_t *vertex; // the vertex
};

struct adjacency_node_s
{
  adjacency_node_t *next;    // link to the next 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)
  unsigned int num_components;   // number of connected components (for information purposes only)
  unsigned int max_component_size_v;   // number of connected components (for information purposes only)
  hash_table_node_t **heads;      // the heads of the linked lists
};

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

```
117
118   static adjacency_node_t *allocate_adjacency_node(void)
119   {
120     adjacency_node_t *node;
121
122     node = (adjacency_node_t *)malloc(sizeof(adjacency_node_t));
123     if (node == NULL)
124     {
125       fprintf(stderr, "allocate_adjacency_node: out of memory\n");
126       exit(1);
127     }
128     return node;
129   }
130
131   static void free_adjacency_node(adjacency_node_t *node)
132   {
133     free(node);
134   }
135
136   static hash_table_node_t *allocate_hash_table_node(void)
137   {
138     hash_table_node_t *node;
139
140     node = (hash_table_node_t *)malloc(sizeof(hash_table_node_t));
141     if (node == NULL)
142     {
143       fprintf(stderr, "allocate_hash_table_node: out of memory\n");
144       exit(1);
145     }
146     return node;
147   }
148
149   static void free_hash_table_node(hash_table_node_t *node)
150   {
151     free(node);
152   }
153
154   //
155   // hash table stuff (mostly to be done)
156   //
157
158   unsigned int crc32(const char *str)
159   {
160     static unsigned int table[256];
```

```
160     static unsigned int table[256];
161     unsigned int crc;
162
163     if (table[1] == 0u) // do we need to initialize the table[] array?
164     {
165       unsigned int i, j;
166
167       for (i = 0u; i < 256u; i++)
168         for (table[i] = i, j = 0u; j < 8u; j++)
169           if (table[i] & 1u)
170             table[i] = (table[i] >> 1) ^ 0xAED00022u; // "magic" constant
171           else
172             table[i] >>= 1;
173     }
174     crc = 0xAED02022u; // initial value (chosen arbitrarily)
175     while (*str != '\0')
176       crc = (crc >> 8) ^ table[crc & 0xFFu] ^ ((unsigned int)*str++ << 24);
177     return crc;
178   }
179
180   static hash_table_t *hash_table_create(void)
181   {
182     hash_table_t *hash_table;
183     unsigned int i;
184
185     hash_table = (hash_table_t *)malloc(sizeof(hash_table_t));
186
187     if (hash_table == NULL)
188     {
189       fprintf(stderr, "create_hash_table: out of memory\n");
190       exit(1);
191     }
192     hash_table->hash_table_size = 100;
193     hash_table->number_of_entries = 0;
194     hash_table->number_of_edges = 0;
195     hash_table->num_components = 0;
196     hash_table->heads = (hash_table_node_t **)malloc(hash_table->hash_table_size * sizeof(hash_table_node_t *));
197     for (i = 0; i < hash_table->hash_table_size; i++)
198       hash_table->heads[i] = NULL;
199     if (hash_table->heads == NULL)
200     {
201       fprintf(stderr, "create_hash_table: out of memory\n");
```

```
  {
    fprintf(stderr, "create_hash_table: out of memory\n");
    exit(1);
  }
  printf("hash table size = %u\n", hash_table->hash_table_size);
  return hash_table;
}

static void hash_table_grow(hash_table_t *hash_table)
{

  hash_table_node_t **old_heads = hash_table->heads;
  hash_table_node_t *n2;
  hash_table_node_t *n1;
  int old_size = hash_table->hash_table_size;

  hash_table->hash_table_size = old_size + old_size / 2;
  hash_table->heads = (hash_table_node_t **)malloc(hash_table->hash_table_size * sizeof(hash_table_node_t *));
  for (int i = 0; i < hash_table->hash_table_size; i++)
  {
    hash_table->heads[i] = NULL;
  }
  for (int i = 0; i < old_size; i++)
  {
    for (n1 = old_heads[i]; n1 != NULL; n1 = n2)
    {
      n2 = n1->next; // importante nao atribuir em cima porque o valor é modificado abaixo
      int j = crc32(n1->word) % hash_table->hash_table_size;
      n1->next = hash_table->heads[j];
      hash_table->heads[j] = n1;
    }
  }
  free(old_heads);
}

static void hash_table_free(hash_table_t *hash_table)
{
  hash_table_node_t *n1;
  hash_table_node_t *n2;
  adjacency_node_t *an1;
  adjacency_node_t *an2;

  for (int i = 0; i < hash_table->hash_table_size; i++)
  {
    n1 = hash_table->heads[i];
```

```
  for (int i = 0; i < hash_table->hash_table_size; i++)
  {
    n1 = hash_table->heads[i];
    for (n1 = hash_table->heads[i]; n1 != NULL; n1 = n2)
    {
      n2 = n1->next;
      for (an1 = n1->head; an1 != NULL; an1 = an2)
      {
        an2 = an1->next;
        free_adjacency_node(an1);
      }
      free_hash_table_node(n1);
    }
  }
  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;
  hash_table_node_t *node2;
  unsigned int i;
  // testar
  i = crc32(word) % hash_table->hash_table_size;
  for (node = hash_table->heads[i]; node != NULL; node = node2)
  {
    if (strcmp(node->word, word) == 0)
    {
      break;
    }
    node2 = node->next;
  }
  if (insert_if_not_found)
  {
    hash_table->number_of_entries++;
    hash_table->num_components++;

    node = allocate_hash_table_node();

    node->head = NULL;
    node->previous = NULL;
    node->number_of_edges = 0;
    node->number_of_vertices = 1;
    node->representative = node;
```

```
286       node->representative = node;
287       node->visited = 0;
288
289       node->next = hash_table->heads[i];
290       hash_table->heads[i] = node;
291       strcpy(node->word, word);
292
293
294       if (hash_table->number_of_entries > 2 * (hash_table->hash_table_size))
295       {
296         hash_table_grow(hash_table);
297         printf("hash table size = %u entries = %u\n", hash_table->hash_table_size, hash_table->number_of_entries);
298       }
299
300     }
301     return node;
302   }
303
304   //
305   // add edges to the word ladder graph (mostly do be done)
306   //
307
308   static hash_table_node_t *find_representative(hash_table_node_t *node)
309   {
310     hash_table_node_t *n1,*n2, *n3;
311
312     for(n1 = node;n1->representative != n1; n1 = n1->representative ) //se o representante for ele proprio, entao ele é o representante
313       ;
314     for(n2 = node; n2 != n1; n2 = n3) //enquanto o n2 for diferente do n1, o n3 vai ser o n2
315     {
316       n3 = n2->representative;
317       n2->representative = n1;
318     }
319
320     return n1;
321   }
322
323   static void add_edge(hash_table_t *hash_table, hash_table_node_t *from, const char *word)
324   {
325     // from é a palavra que ja esta no hash table
326     // word é a palavra que esta a ser lida
327
328     hash_table_node_t *to, *from_representative, *to_representative;
329     adjacency_node_t *link;
330
```

```
330
331     to = find_word(hash_table, word, 0);
332
333     if (to == NULL)
334       return;
335
336     for (link = from->head; link != NULL && link->vertex != to; link = link->next)
337       ;
338
339     if (link != NULL)
340       return;
341
342     link = allocate_adjacency_node();
343     link->vertex = to->word;
344     link->next = from->head;
345     from->head = link;
346     from->number_of_edges++;
347
348     link = allocate_adjacency_node();
349     link->vertex = from->word;
350     link->next = to->head;
351     to->head = link;
352     to->number_of_edges++;
353
354     hash_table->number_of_edges++;
355
356     from_representative = find_representative(from);
357     to_representative = find_representative(to);
358
359     if (from_representative != to_representative)
360     {
361       if (from_representative->number_of_vertices < to_representative->number_of_vertices)
362       {
363         to_representative->number_of_vertices += from_representative->number_of_vertices;
364         to_representative->number_of_edges += from_representative->number_of_edges + 1 ;
365         from_representative->number_of_vertices = 0;
366         from_representative->number_of_edges = 0;
367         hash_table->num_components--;
368         from_representative->representative = to_representative;
369       }
370       else
371       {
372         from_representative->number_of_vertices += to_representative->number_of_vertices;
373         from_representative->number_of_edges += to_representative->number_of_edges + 1 ;
374         to_representative->number_of_vertices = 0;
```

```
        to_representative->number_of_edges = 0;
        hash_table->num_components--;
        to_representative->representative = from_representative;
      }
    }else{
      from_representative->number_of_edges++;
    }
  }

  // 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;
```

```
      *(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};
  static int counter = 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
  //

  //enqueue
  static void enqueue(queue_t *queue, hash_table_node_t *node)
  {
    queue_node_t *queue_node = (queue_node_t *)malloc(sizeof(queue_node_t));
    queue_node->node = node;
    queue_node->next = NULL;
    if (queue->head == NULL)
      queue->head = queue_node;
    else
      queue->tail->next = queue_node;
    queue->tail = queue_node;
  }

  //dequeue
  hash_table_node_t *dequeue(queue_t *queue)
  {
    hash_table_node_t *node;
    queue_node_t *queue_node;

    if (queue->head == NULL)
      return NULL;
    queue_node = queue->head;
    queue->head = queue_node->next;
    if (queue->head == NULL)
      queue->tail = NULL;
    node = queue_node->node;
    free(queue_node);
    return node;
  }

  //free_queue
  static void free_queue(queue_t *queue)
  {
    queue_node_t *queue_node;

    while (queue->head != NULL)
    {
```

```
      queue_node = queue->head;
      queue->head = queue_node->next;
      free(queue_node);
    }
    free(queue);
  }


static int breadth_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 i, j, k, n, number_of_vertices_visited = 0;
  hash_table_node_t *node, *node2;
  adjacency_node_t *adjacency_node;

  // initialize queue
  queue_t *queue = (queue_t *)malloc(sizeof(queue_t));
  queue->head = NULL;
  queue->tail = NULL;

  // enqueue origin
  enqueue(queue, origin);
  // while queue is not empty
  while (queue->head != NULL)
  {
    // dequeue node if not null
    if((node = dequeue(queue)) == NULL)
      break;

    // if node is goal
    if (node == goal)
    {
      // return number of vertices visited
      return number_of_vertices_visited;
    }

    // if node is not visited
    if (node->visited == 0)
    {
      // mark node as visited
      node->visited = 1;

      // enqueue all adjacent nodes
      adjacency_node = node->head;
      while (adjacency_node != NULL)
```

```
        {
          if(adjacency_node->vertex->visited == 0){
            adjacency_node->vertex->previous = node;
            enqueue(queue, adjacency_node->vertex);
            adjacency_node = adjacency_node->next;
          }
        }

        // increment number of vertices visited
        number_of_vertices_visited++;
      }
      free_queue(queue);
    }
    //free queue
    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
  {
    hash_table_node_t *node;
    hash_table_node_t *n1;
    hash_table_node_t *n2;
    hash_table_node_t *rep;
    hash_table_node_t *rep_n;
    int count = 0;

    node = find_word(hash_table, word, 0);

    if (node == NULL)
    {
      printf("list_connected_component: word not found\n");
      return;
    }

    rep_n = find_representative(node);

    for (int i = 0; i < hash_table->hash_table_size; i++)
    {
      n1 = hash_table->heads[i];
      for (n1 = hash_table->heads[i]; n1 != NULL; n1 = n2)
      {
        n2 = n1->next;
        rep = find_representative(n1);
        if (rep == rep_n)
        {
          count++;
          printf("%s \n", n1->word);
        }
      }
    }
    printf("Number of vertices: %d \n", count);
  }

  //
  // 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
    //
    // find the shortest path from a given word to another given word
    // use breadth-first search
    hash_table_node_t *from = find_word(hash_table, from_word, 0);
    if (from == NULL)
```

```
641   if (from == NULL)
642   {
643     printf("path_finder: from_word not found\n");
644     return;
645   }
646
647   hash_table_node_t *to = find_word(hash_table, to_word, 0);
648   if (to == NULL)
649   {
650     printf("path_finder: to_word not found\n");
651     return;
652   }
653   // list of vertices of hash table
654   hash_table_node_t **list_of_vertices = hash_table->heads;
655
656   // breadth-first search
657   int number_of_vertices_visited = breadth_first_search(hash_table->number_of_entries, list_of_vertices, from, to);
658
659   // print result
660   printf("path_finder: number of vertices visited = %d  \n", number_of_vertices_visited);
661
662   // print path
663   hash_table_node_t *node = to;
664   while (node != NULL)
665   {
666     printf("path_finder: %s \n", node->word);
667     node = node->previous;
668   }
669 }
670
671 //
672 // some graph information (optional)
673 //
674
675 static void graph_info(hash_table_t *hash_table)
676 {
677   printf("graph_info:number of edges = %d \n", hash_table->number_of_edges); // resultado do stor 9265
678   printf("graph_info:number of vertices = %d \n", hash_table->number_of_entries); // resultado do stor 9917
679   printf("graph_info:number of connected components = %d \n", hash_table->num_components); // resultado do stor 187
680   hash_table_node_t *node;
681   hash_table_node_t *n1;
682   hash_table_node_t *n2;
683   hash_table_node_t *rep;
684   hash_table_node_t *rep_n;
685   int countedge = 0;
686   int count = 0;
687   int countmin = 4000000;
688   int countminedges = 4000000;
689   for (int i = 0; i < hash_table->hash_table_size; i++)
690   {
691     for (n1 = hash_table->heads[i]; n1 != NULL; n1 = n2)
692     {
693       n2 = n1->next;
694       rep = find_representative(n1);
695       if(rep->number_of_vertices>count){
696         count = rep->number_of_vertices;
697       }
698       if(rep->number_of_edges>countedge){
699         countedge = rep->number_of_edges;
700       }
701       if(rep->number_of_edges<countminedges){
702         countminedges = rep->number_of_edges;
703       }
704       if(rep->number_of_vertices<countmin){
705         countmin = rep->number_of_vertices;
706       }
707     }
708   }
709   printf("Max number of vertices of connected component: %d \n", count);
710   printf("Max number of edges of connected component: %d \n", countedge);
711   printf("Min number of vertices of connected component: %d \n", countmin);
712   printf("Min number of edges of connected component: %d \n", countminedges);
713 }
714
715 //
716 // main program
717 //
718
719 int main(int argc, char **argv)
720 {
721   char word[100], from[100], to[100];
722   hash_table_t *hash_table;
723   hash_table_node_t *node;
724   unsigned int i;
725   int command;
726   FILE *fp;
```

```
727
728   // initialize hash table
729   hash_table = hash_table_create();
730   // read words
731   fp = fopen((argc < 2) ? "wordlist-big-latest.txt" : argv[1], "rb");
732   if (fp == NULL)
733   {
734     fprintf(stderr, "main: unable to open the words file\n");
735     exit(1);
736   }
737   while (fscanf(fp, "%99s", word) == 1)
738     (void)find_word(hash_table, word, 1);
739   fclose(fp);
740   // find all similar words
741   for (i = 0u; i < hash_table->hash_table_size; i++)
742     for (node = hash_table->heads[i]; node != NULL; node = node->next)
743       similar_words(hash_table, node);
744   graph_info(hash_table);
745   // ask what to do
746   for (;;)
747   {
748     fprintf(stderr, "Your wish is my command:\n");
749     fprintf(stderr, "  1 WORD       (list the connected component WORD belongs to)\n");
750     fprintf(stderr, "  2 FROM TO    (list the shortest path from FROM to TO)\n");
751     fprintf(stderr, "  3            (terminate)\n");
752     fprintf(stderr, "> ");
753     if (scanf("%99s", word) != 1)
754       break;
755     command = atoi(word);
756     if (command == 1)
757     {
758
759       if (scanf("%99s", word) != 1)
760         break;
761       list_connected_component(hash_table, word);
762     }
763     else if (command == 2)
764     {
765       if (scanf("%99s", from) != 1)
766         break;
767       if (scanf("%99s", to) != 1)
768         break;
769       path_finder(hash_table, from, to);
770     }
771     else if (command == 3)
```

```
772         break;
773     }
774     // clean up
775     hash_table_free(hash_table);
776     return 0;
777   }
778
```

Link com o Código completo:
https://drive.google.com/file/d/1PxCS6fvXlrhc8y3RMkniK4cH0WD0Y7XH/view?usp=share_link

Conclusão

Este trabalho prático permitiu consolidar conhecimentos acerca de vários assuntos lecionados ao longo da disciplina de Algoritmos e Estruturas de Dados, nomeadamente *Hash Tables, Breadth First Search*. Além disso, este trabalho permitiu nos utilizar vários conceitos que estão por base na Linguagem de Programação usada, *C*, como alocação de memória.

Por fim, como base deste trabalho, é importante referir que de uma maneira eficiente e de fácil implementação, aprendemos uma nova forma de obter informação variada de um ficheiro de texto.