Word ladder

Algoritmos e Estruturas de Dados

Prof. Tomás Oliveira e Silva
Prof. João Manuel Rodrigues

Gonçalo F. Couto Sousa - Nº Mec: 108133 - 40%
Liliana P. Cruz Ribeiro - Nº Mec: 108713 - 60%

universidade de aveiro

Departamento de Eletrónica, Telecomunicações e Informática

Universidade de Aveiro

Índice

Introdução

Este relatório foi realizado no âmbito de expor os resultados do Segundo Trabalho Prático da cadeira Algoritmos e Estruturas de Dados (AED).

O projeto baseava-se no jogo Word Ladder, que consiste em partir de duas palavras com o mesmo número de letras e ir alterando uma delas letra por letra até chegarmos à segunda palavra, no menor número de alterações possíveis, sendo que as letras alteradas terão de fazer sempre uma palavra existente na linguagem que estamos a usar.

Para desenvolver este projeto partimos do código já desenvolvido pelos docentes da disciplina no ficheiro **word_ladder.c**, completando as várias funções apresentadas. As diferentes funções que tínhamos de completar eram de diferentes tipos (sendo que explicaremos cada uma delas mais à frente):

- Funções Hash:
  - hash_table_create
  - hash_table_grow
  - hash_table_free
  - find_word
- Funções de construção do grafo:
  - find_representative
  - add_edge
- Função breadh_first_search
- Função list_connected_component
- Função path_finder

Hash Tables

A parte obrigatória deste projeto consistia em completar diversas hash tables.

As hash tables são uma estrutura de dados que permite armazenar informação de forma eficiente e rápida. Elas funcionam através da utilização de uma hash function, que é responsável por transformar as chaves (ou seja, os identificadores) dos dados a serem armazenados num índice de um array, onde o valor correspondente será armazenado. Desta forma, ao querer recuperar um dado, basta utilizar a mesma hash function para calcular o índice onde esse dado se encontra e, assim, obter o valor desejado de forma rápida.

Hash_table_create

Esta função é usada para criar uma nova hash table. Ela aloca memória para a nova hash table e inicializa os seus atributos.

Primeiramente, é reservado espaço de memória para uma nova hash table usando a função malloc. Em seguida, é verificado se houve sucesso na alocação de memória, caso contrário, é exibida uma mensagem de erro e o programa é encerrado.

No trecho de código desenvolvido por nós é escolhido um tamanho para a hash table (Nós escolhemos 107), isto é importante para a escolha do tamanho da tabela, para evitar colisões e garantir a eficiência da tabela, é recomendado ser um número primo.

Depois são guardados os atributos da hash table que incluem

- hash_table_size: que é o tamanho escolhido para a tabela hash.
- number_of_entries: que é o número de entradas na tabela hash.
- number_of_edges: que é o número de arestas na tabela hash.

Por último é alocado espaço de memória para a tabela de cabeçalhos (heads) da hash table usando a função malloc e usa-se a

função memset para inicializar todos os elementos do array de cabeçalhos com zero.
Finalmente, a função retorna a nova hash table criada.

```
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);
   }

   /* students code */
   // choose size for the hash table (prime number)
   unsigned int size = 107;

   // save hash table atributes
   hash_table->hash_table_size = size;
   hash_table->number_of_entries = 0u;
   hash_table->number_of_edges = 0u;

   hash_table->heads = (hash_table_node_t **)malloc(size *
sizeof(hash_table->heads));
   memset(hash_table->heads, 0, hash_table->hash_table_size *
sizeof(hash_table->heads));

   /* end code */
   return hash_table;
}
```

Hash_table_grow

Esta função "hash_table_grow" é usada para aumentar o tamanho da hash table. Ela move todos os nós existentes na hash table antiga para uma nova hash table com o dobro do tamanho. Isso é feito para

aumentar a eficiência da hash table e evitar colisões, preservando os cabeçalhos antigos.

Primeiramente, a função declara e inicializa as variáveis usadas para salvar os valores antigos e novos da hash table:

- heads: é um ponteiro para a tabela de cabeçalhos antiga.
- new_heads: é um ponteiro para a nova tabela de cabeçalhos.
- node, next_node: são variáveis para percorrer os nós da hash table antiga.
- size, new_size: são variáveis para armazenar o tamanho antigo e o novo tamanho da hash table.

Depois, o código salva os valores antigos da hash table em variáveis temporárias: heads (a tabela de cabeçalhos) e size (o tamanho da tabela).

Em seguida, é alocado espaço de memória para a nova hash table com o tamanho duplicado (new_size) usando a função malloc e inicializado todos os elementos do array de cabeçalhos com zero.

Posteriormente, o código percorre a hash table antiga e re-calcula a função de hash para cada nó usando a nova hash table e novo tamanho. Isso é feito através de dois ciclos for. O primeiro ciclo percorre cada cabeçalho na tabela antiga e o segundo ciclo percorre cada nó em cada cabeçalho. Cada nó é adicionado à nova hash table usando a função de hash re-calculada.

Por fim, os valores antigos da tabela hash são libertados da memória e os valores novos (tamanho e tabela de cabeçalhos) são atribuídos à hash table.

```
static void hash_table_grow(hash_table_t *hash_table)
{
 /* students code */
 // variables to save old and new values
 hash_table_node_t **heads, **new_heads;
 hash_table_node_t *node, *next_node;
 unsigned int size, new_size;
 unsigned int i;

 // save old values
```

```
 heads = hash_table->heads;
 size = hash_table->hash_table_size;

 // new values (new size -> double the size)
 new_size = size * 2u;
 new_heads = (hash_table_node_t **)malloc(new_size *
sizeof(hash_table_node_t *));

 for (i = 0u; i < new_size; i++)
   new_heads[i] = NULL;

 if (new_heads == NULL)
 {
   fprintf(stderr, "hash_table_grow: out of memory\n");
   exit(1);
 }

 // do the hash function for the new size (and save on new_heads)
 // go throught all the heads and replace them for the new hash
function calculate
 for (i = 0u; i < size; i++)
   for (node = heads[i]; node != NULL; node = next_node)
   {
     next_node = node->next;
     node->next = new_heads[crc32(node->word) % new_size];
     new_heads[crc32(node->word) % new_size] = node;
   }
 // replace hash table old values (size e head) for new values
 free(heads);
 hash_table->hash_table_size = new_size;
 hash_table->heads = new_heads;
 /* end code */
}
```

Hash_table_free

Esta função é usada para libertar a memória alocada para uma hash table. Ela percorre todos os elementos da hash table, libertando a memória alocada para cada nó e cada lista de adjacência.

Primeiramente, no código desenvolvido por nós, declaramos algumas variáveis úteis. Depois, usamos um ciclo for para percorrer todos os elementos na hash table.

Dentro do ciclo for, usamos um ciclo while para percorrer a lista ligada de nós na hash table e libertar a memória alocada para cada nó.

Dentro desse ciclo while, usamos outro ciclo while para percorrer a lista de adjacência do nó atual e libertar a memória alocada para cada nó de adjacência.

Após o fim do ciclo for, o código liberta a memória alocada para o array de cabeçalhos da hash table e a estrutura da hash table em si.

```
static void hash_table_free(hash_table_t *hash_table)
{
 /* students code */
 unsigned int i;
 hash_table_node_t *node, *next_node;
 adjacency_node_t *adj_node, *next_adj_node;
 // Itera sobre todos os elementos na hash table
 for (i = 0u; i < hash_table->hash_table_size; i++)
 {
   node = hash_table->heads[i];
   while (node)
   {
     next_node = node->next;
     adj_node = node->head;
     while (adj_node)
     {
       next_adj_node = adj_node->next;
       free(adj_node);
       adj_node = next_adj_node;
     }
     free(node);
     node = next_node;
   }
   /* end code */
 }

 free(hash_table->heads);

 // liberta a estrutura da hash table
```

```
 free(hash_table);
}
```

find_word

Esta função é usada para encontrar ou inserir um nó específico na hash table dada uma palavra.

Primeiramente, o código usa a função crc32 para calcular o índice da hash table onde a palavra deve ser encontrada.

Em seguida, usamos um ciclo while para percorrer a lista ligada de nós no índice da hash table calculado e comparar cada nó com a palavra procurada. Se a palavra for encontrada, o ponteiro para o nó é retornado. Se a palavra não for encontrada e a flag "insert_if_not_found" estiver definida, o código aloca memória para um novo nó e inicializa os seus atributos com os valores da palavra e os valores iniciais. O novo nó é inserido na lista ligada no índice da tabela hash calculado. Também é verificado se é necessário aumentar o tamanho da tabela hash.
Se a flag "insert_if_not_found" não estiver definida, a função retorna NULL.

```
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;

 i = crc32(word) % hash_table->hash_table_size;

 /* studants code */
 node = hash_table->heads[i];
 while (node != NULL)
 {
   if (strcmp(node->word, word) == 0)
   {
     // retorna um apontador para o nó se a palavra for encontrada
     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->next = hash_table->heads[i];
   node->head = NULL;
   node->visited = 0;
   node->previous = NULL;
   node->representative = node;
   node->number_of_vertices = 1;
   node->number_of_edges = 0;
   // guarda os valores e incrementa as entradas
   hash_table->heads[i] = node;     // 8
   hash_table->number_of_entries++; // 9
   // cresce, caso necessário
   if (hash_table->number_of_entries > hash_table->hash_table_size)
   {
     hash_table_grow(hash_table);
   }
 }
 else
 {
   // caso a palavra não for encontrada e insert_if_not_found não
estiver definida
   return NULL;
 }
 // retorna um apontador para o nó se insert_if_not_found for definida
 return node;

 /* end studants code */
}
```

Testagem da Hash table

Após desenvolvermos as funções da hash table, desenvolvemos a função print_hash_table para testarmos se as funções da hash table estão a funcionar como nós pretendiamos, sendo que está dá print da hash table.

Para usar esta função, acrescentámos código na função main que deve ser descomentado para conseguirmos ver a hash table, como mostrado a seguir.

```c
// this function prints the hash table to test it just uncomment it on main
static void print_hash_table(hash_table_t *hash_table)
{
 for (unsigned int i = 0; i < hash_table->hash_table_size; i++)
 {
   printf("Space %d: ", i);
   hash_table_node_t *current = hash_table->heads[i];
   while (current != NULL)
   {
     printf("%s -> ", current->word);
     current = current->next;
   }
   printf("NULL\n");
 }
 return;
}
```

```
Space 3326: NULL
Space 3327: NULL
Space 3328: NULL
Space 3329: atum -> adam -> arfa -> NULL
Space 3330: bise -> NULL
Space 3331: cise -> caie -> NULL
Space 3332: deus -> NULL
Space 3333: NULL
Space 3334: fofo -> frui -> NULL
Space 3335: gene -> NULL
Space 3336: NULL
Space 3337: iene -> NULL
Space 3338: NULL
Space 3339: NULL
Space 3340: NULL
Space 3341: meus -> mofo -> NULL
Space 3342: NULL
Space 3343: NULL
Space 3344: Pera -> pene -> pise -> NULL
Space 3345: ruís -> NULL
Space 3346: raie -> NULL
Space 3347: sise -> seus -> NULL
Space 3348: triz -> teus -> NULL
Space 3349: caís -> NULL
Space 3350: vise -> vaie -> Vera -> NULL
Space 3351: NULL
Space 3352: NULL
Space 3353: NULL
Space 3354: NULL
Space 3355: NULL
```

*** até aqui conseguimos fazer tudo sem problemas :) ***

Funções de construção do grafo

find_representative

Esta função é usada para encontrar o representante de um componente conectado dado um nó específico.

Primeiramente, é definido um ponteiro "representative" para o nó especificado e um ponteiro "next_node" para o representante desse nó.

Em seguida, o código usa um ciclo while para percorrer a cadeia de representantes, comparando o próximo nó na cadeia com o representante atual. Se o próximo nó é diferente do representante atual, o representante é atualizado com o próximo nó e o ciclo continua. Caso contrário, o laço é interrompido e o ponteiro "representative" é retornado.

```
      static hash_table_node_t *find_representative(hash_table_node_t
*node)
      {
       hash_table_node_t *representative, *next_node;

       /* students code */
       // Follow the chain of representatives until the representative
of the connected component is found
       representative = node;
       next_node = node->representative;
       while (next_node != representative)
       {
         representative = next_node;
         next_node = representative->representative;
       }
       /* end code */
       return representative;
      }
```

add_edge

A função add_edge é utilizada para adicionar arestas ao grafo. Ela recebe como parâmetros um ponteiro para a hash table, um ponteiro

para um nó da hash table que representa a palavra de origem, e uma string que representa a palavra de destino.

Primeiro, a função usa a função find_word para encontrar um nó correspondente à palavra de destino na hash table. Se os nós de origem ou destino não existirem, a função retorna imediatamente.

Em seguida, a função usa a função find_representative para encontrar os representantes das componentes conectadas dos nós de origem e destino. Se os representantes são diferentes, a função atualiza as informações dos representantes de acordo com a quantidade de vértices presentes. Se os representantes são iguais, significa que as palavras já estão conectadas, então apenas incrementa o número de arestas.

Por fim, a função aloca dois novos nós de adjacência, e adiciona esses nós à lista de adjacência dos nós de origem e destino. Incrementa o número de arestas na hash table e retorna.

```
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;
 adjacency_node_t *link;

 to = find_word(hash_table, word, 0);
 /* students code */
 if (to == NULL)
   return;
 if (from == NULL)
   return;
 // atualizar os representantes das componentes conectadas
 from_representative = find_representative(from);
 to_representative = find_representative(to);
 if (from_representative != to_representative)
 {
   // se for menor
   if (from_representative->number_of_vertices <
to_representative->number_of_vertices)
   {
     to_representative->number_of_vertices +=
from_representative->number_of_vertices;
     from_representative->representative = to_representative;
```

```
      from_representative->number_of_edges++;
      from->representative = to_representative;
    }
    else
    { // se for maior ou igual
      from_representative->number_of_vertices +=
to_representative->number_of_vertices;
      to_representative->representative = from_representative;
      to->representative = from_representative;
    }
  }
  else
  {
    // se a representação das palavras for igual
    from_representative->number_of_edges++;
  }

  // adicionar link ao node e atualizar a lista de links
  // from
  link = allocate_adjacency_node();
  if (link == NULL)
  {
    fprintf(stderr, "add_edge: out of memory\n");
    exit(1);
  }
  link->vertex = to;
  link->next = from->head;
  from->head = link;

  // to
  link = allocate_adjacency_node();
  if (link == NULL)
  {
    fprintf(stderr, "add_edge: out of memory\n");
    exit(1);
  }
  link->vertex = from;
  link->next = to->head;
  to->head = link;

  // save que temos mais um edge
  hash_table->number_of_edges++;
  return;
```

```
 /* end code */
}
```

breadh_first_search

Esta função realiza uma busca em largura (breadth-first search) que é usada para encontrar o caminho mais curto entre dois vértices (nós) específicos no grafo (em caso de ser selecionada a opção 1 encontra o caminho entre a palavra dada e todas as outras). Ela recebe como parâmetros o número máximo de vértices, uma lista de vértices, o vértice de origem e o vértice de destino.
A função inicia com a adição do vértice de origem à lista de vértices e define o vértice anterior como nulo e o status "visitado" como 1. Ela também inicia uma variável "found" como 0 (falso) para indicar se o vértice de destino foi encontrado ou não.
A função entra num ciclo enquanto "found" for falso. Dentro do ciclo, a função verifica se há vértices na lista para serem lidos e, se houver, ele itera sobre os links de adjacência do vértice atual. Se um vértice não foi visitado anteriormente, ele é marcado como visitado, o vértice anterior é definido como o vértice atual e é adicionado à lista de vértices. Se o vértice atual for o vértice de destino, a variável "found" é definida como 1 (verdadeiro) e o ciclo é finalizado. Se não houver mais vértices na lista, o ciclo é finalizado.

Depois de sair do ciclo, a função dá reset do status "visitado" de todos os vértices visitados e retorna o número de vértices visitados. Se o último vértice visitado for o vértice de destino, seguindo os links anteriores iremos obter o caminho mais curto entre o vértice de destino e o vértice de origem.

```
      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 read_index = 0, write_index = 1;
       list_of_vertices[0] = origin;
```

```
    origin->previous = NULL;
    origin->visited = 1;
    unsigned int found = 0; // 0-> false | 1-> true

    // enquanto não encontrarmos percorremos tudo
    while (!found)
    {
      if (read_index != write_index)
      {
        adjacency_node_t *link = list_of_vertices[read_index]->head;
        //proximo vertice a ler
        read_index++;
        while (link != NULL)
        {
          if (link->vertex->visited == 0)
          {
            // visitamos um novo vertice
            link->vertex->visited = 1;
            link->vertex->previous = list_of_vertices[read_index -
1];
            list_of_vertices[write_index] = link->vertex;
            // proximo vertice a escrever
            write_index++;
            if (link->vertex == goal)
            {
              found = 1;
              break;
            }
          }
          link = link->next;
        }
      }
      else
      {
        break;
      }
    }
    // Reset the visited status of the vertices
    for (int i = 0; i < write_index; i++)
    {
      list_of_vertices[i]->visited = 0;
    }
    // return number of vertices visited
```

```
    return write_index;
}
```

list_connected_component

Esta função, list_connected_component, é usada para listar todas as palavras na mesma componente conectada a uma dada palavra. Ela começa por procurar a palavra especificada na hash table e verifica se ela existe. Se a palavra não existir, ela imprime uma mensagem de erro e sai da função. Se a palavra existir, ela encontra o representante dessa palavra, que é o nó no grafo que representa a componente conectada na qual a palavra está incluída. Ela posteriormente aloca espaço suficiente para armazenar todos os vértices na componente conectada, usando o número de vértices na componente conectada encontrado anteriormente. Em seguida, ela usa a função breadh_first_search para percorrer a componente conectada e armazenar todos os vértices visitados numa lista. Finalmente, ela imprime cada vértice na lista e liberta o espaço alocado para a lista.

```
static void list_connected_component(hash_table_t *hash_table, const
char *word)
{
 /* students code */
 hash_table_node_t *origin = find_word(hash_table, word, 0);
 // se a palavra não existir
 if (origin == NULL)
 {
   printf("list_connected_component: word not found\n");
   return;
 }
 // se a palavra existir
 unsigned int count = 0, numberV;
 unsigned int maximum_number_of_vertices =
find_representative(origin)->number_of_vertices;
 // get some space to creat our list
 hash_table_node_t **list_of_vertices = malloc(sizeof(hash_table_node_t
*) * maximum_number_of_vertices);
 // this function is for option 1 so there is no goal :)
```

```
 numberV = breadh_first_search(maximum_number_of_vertices,
list_of_vertices, origin, NULL);
 // print todos os vertices
 for (count; count < numberV; count++)
 {
   printf("%d. %s\n", count, list_of_vertices[count]->word);
 }
 free(list_of_vertices);
 /* end code */
}
```

path_finder

A função path_finder() tem como objetivo encontrar o caminho mais curto entre duas palavras fornecidas como entrada, "from_word" e "to_word". Ela começa por verificar se as palavras existem na hash table, atribuindo-as às variáveis "origin" e "goal". Se uma ou ambas as palavras não forem encontradas, a função imprime uma mensagem de erro e retorna.

Em seguida, a função encontra os representantes das palavras de origem e destino usando a função find_representative(). Se os representantes das palavras não forem os mesmos, significa que as palavras não estão conectadas no grafo, portanto, a função imprime uma mensagem de erro e retorna.

A função depois aloca espaço para uma lista de vértices com o número máximo de vértices do componente conectado da palavra de origem. Em seguida, usa a função breadh_first_search() para encontrar o caminho mais curto entre as palavras de origem e destino e depois retorna o número de vértices visitados.

Em seguida, a função imprime cada palavra no caminho mais curto, começando com a palavra de destino e seguindo as ligações "previous" até a palavra de origem. Por fim, a função liberta a memória alocada para a lista de vértices.

```
static void path_finder(hash_table_t *hash_table, const char
*from_word, const char *to_word)
```

```
{
 /* students code */
 // basicamente we do the same thing has list_connected_component but
with a goal
 hash_table_node_t *goal = find_word(hash_table, from_word, 0);
 hash_table_node_t *origin = find_word(hash_table, to_word, 0);
 // se a palavra não existir
 if (origin == NULL)
 {
   printf("list_connected_component: word not found\n");
   return;
 }
 // se a palavra existir
 unsigned int count = 0, numberV;
 unsigned int maximum_number_of_vertices =
find_representative(origin)->number_of_vertices;
 // get some space to creat our list
 hash_table_node_t **list_of_vertices = malloc(sizeof(hash_table_node_t
*) * maximum_number_of_vertices);
 // this function is for option 1 so there is no goal :)
 numberV = breadh_first_search(maximum_number_of_vertices,
list_of_vertices, origin, goal);
 // print todos os vertices
 unsigned int n = numberV - 1;
 hash_table_node_t *p = list_of_vertices[numberV - 1];
 while (p != NULL)
 {
   printf("%d: %s \n", count, p->word);
   count++;
   p = p->previous;
 }

 free(list_of_vertices);
 /* end code */
}
```

Word ladders que consideramos interessantes

- Considerámos interessante porque mostra-nos que a função funciona para um path grande;

```
Your wish is my command:
  1 WORD       (list the connected component WORD belongs to)
  2 FROM TO    (list the shortest path from FROM to TO)
  3            (terminate)
> 2 amor tabu
0: amor
1: amos
2: aios
3: fios
4: fins
5: fino
6: fano
7: fato
8: tato
9: tatu
10: tabu
```

- Mostra-nos que também funciona para um path pequeno;

```
Your wish is my command:
  1 WORD       (list the connected component WORD belongs to)
  2 FROM TO    (list the shortest path from FROM to TO)
  3            (terminate)
> 2 moça rosa
0: moça
1: roça
2: rosa
```

- Mostra-nos que o programa também funciona para palavras mais extensas;

```
Your wish is my command:
  1 WORD       (list the connected component WORD belongs to)
  2 FROM TO    (list the shortest path from FROM to TO)
  3            (terminate)
> 2 engole expele
0: engole
1: enfole
2: esfole
3: esfola
4: escola
5: escora
6: espora
7: espera
8: espeta
9: expeta
10: expete
11: expele
```

Apêndice do Código final

```
//
// AED, November 2022 (Tomás Oliveira e Silva)
//
// Second practical assignement (speed run)
//
// Place your student numbers and names here
//   N.Mec. 108713  Name: Liliana Ribeiro
//   N.Mec. 108133  Name: Gonçalo Sousa           .
//
// Do as much as you can
//   1) MANDATORY: complete the hash table code
//      *) hash_table_create
//      *) hash_table_grow
//      *) hash_table_free
//      *) find_word
//      +) add code to get some statistical data about the hash table
//   2) HIGHLY RECOMMENDED: build the graph (including union-find data)
-- use the similar_words function...
//      *) find_representative
//      *) add_edge
```

```
//   3) RECOMMENDED: implement breadth-first search in the graph
//      *) breadh_first_search
//   4) RECOMMENDED: list all words belonginh to a connected component
//      *) breadh_first_search
//      *) list_connected_component
//   5) RECOMMENDED: find the shortest path between to words
//      *) breadh_first_search
//      *) path_finder
//      *) test the smallest path from bem to mal
//         [ 0] bem
//         [ 1] tem
//         [ 2] teu
//         [ 3] meu
//         [ 4] mau
//         [ 5] mal
//      *) find other interesting word ladders
//   6) OPTIONAL: compute the diameter of a connected component and
list the longest word chain
//      *) breadh_first_search
//      *) connected_component_diameter
//   7) OPTIONAL: print some statistics about the graph
//      *) graph_info
//   8) OPTIONAL: test for memory leaks
//

#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;
```

```
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)
//

static adjacency_node_t *allocate_adjacency_node(void)
```

```
{
 adjacency_node_t *node;

 node = (adjacency_node_t *)malloc(sizeof(adjacency_node_t));
 if (node == NULL)
 {
   fprintf(stderr, "allocate_adjacency_node: out of memory\n");
   exit(1);
 }
 return node;
}

static void free_adjacency_node(adjacency_node_t *node)
{
 free(node);
}

static hash_table_node_t *allocate_hash_table_node(void)
{
 hash_table_node_t *node;

 node = (hash_table_node_t *)malloc(sizeof(hash_table_node_t));
 if (node == NULL)
 {
   fprintf(stderr, "allocate_hash_table_node: out of memory\n");
   exit(1);
 }
 return node;
}

static void free_hash_table_node(hash_table_node_t *node)
{
 free(node);
}

//
// hash table stuff (mostly to be done)
//

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

// hash_table_create is a function that returns a empty table with size
107
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);
 }

 /* students code */
 // choose size for the hash table (prime number)
 unsigned int size = 107;

 // save hash table atributes
 hash_table->hash_table_size = size;
 hash_table->number_of_entries = 0u;
 hash_table->number_of_edges = 0u;
```

```
 hash_table->heads = (hash_table_node_t **)malloc(size *
sizeof(hash_table->heads));
 memset(hash_table->heads, 0, hash_table->hash_table_size *
sizeof(hash_table->heads));

 /* end code */
 return hash_table;
}

// hash_table_grow is a function that double the size of an hash_table
preserving the old heads
static void hash_table_grow(hash_table_t *hash_table)
{
 /* students code */
 // variables to save old and new values
 hash_table_node_t **heads, **new_heads;
 hash_table_node_t *node, *next_node;
 unsigned int size, new_size;
 unsigned int i;

 // save old values
 heads = hash_table->heads;
 size = hash_table->hash_table_size;

 // new values (new size -> double the size)
 new_size = size * 2u;
 new_heads = (hash_table_node_t **)malloc(new_size *
sizeof(hash_table_node_t *));

 for (i = 0u; i < new_size; i++)
   new_heads[i] = NULL;

 if (new_heads == NULL)
 {
   fprintf(stderr, "hash_table_grow: out of memory\n");
   exit(1);
 }

 // do the hash function for the new size (and save on new_heads)
 // go throught all the heads and replace them for the new hash
function calculate
 for (i = 0u; i < size; i++)
   for (node = heads[i]; node != NULL; node = next_node)
```

```
    {
      next_node = node->next;
      node->next = new_heads[crc32(node->word) % new_size];
      new_heads[crc32(node->word) % new_size] = node;
    }
  // replace hash table old values (size e head) for new values
  free(heads);
  hash_table->hash_table_size = new_size;
  hash_table->heads = new_heads;
  /* end code */
}

// hash_table_free: This function should free all the memory used by
the hash table.
static void hash_table_free(hash_table_t *hash_table)
{
  /* students code */
  unsigned int i;
  hash_table_node_t *node, *next_node;
  adjacency_node_t *adj_node, *next_adj_node;
  // Itera sobre todos os elementos na hash table
  for (i = 0u; i < hash_table->hash_table_size; i++)
  {
    node = hash_table->heads[i];
    while (node)
    {
      next_node = node->next;
      adj_node = node->head;
      while (adj_node)
      {
        next_adj_node = adj_node->next;
        free(adj_node);
        adj_node = next_adj_node;
      }
      free(node);
      node = next_node;
    }
    /* end code */
  }

  free(hash_table->heads);

  // liberta a estrutura da hash table
```

```
 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;

 i = crc32(word) % hash_table->hash_table_size;

 /* studants code */
 node = hash_table->heads[i];
 while (node != NULL)
 {
   if (strcmp(node->word, word) == 0)
   {
     // retorna um apontador para o nó se a palavra for encontrada
     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->next = hash_table->heads[i];
   node->head = NULL;
   node->visited = 0;
   node->previous = NULL;
   node->representative = node;
   node->number_of_vertices = 1;
   node->number_of_edges = 0;
   // guarda os valores e incrementa as entradas
   hash_table->heads[i] = node;     // 8
   hash_table->number_of_entries++; // 9
   // cresce, caso necessário
   if (hash_table->number_of_entries > hash_table->hash_table_size)
   {
     hash_table_grow(hash_table);
   }
 }
```

```
 else
 {
   // caso a palavra não for encontrada e insert_if_not_found não
estiver definida
   return NULL;
 }
 // retorna um apontador para o nó se insert_if_not_found for definida
 return node;

 /* end studants code */
}

//
// 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;

 /* students code */
 representative = node;
 next_node = node->representative;
 while (next_node != representative)
 {
   representative = next_node;
   next_node = representative->representative;
 }
 /* end code */
 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;
 adjacency_node_t *link;

 to = find_word(hash_table, word, 0);
 /* students code */
 if (to == NULL)
   return;
 if (from == NULL)
```

```
    return;
  // atualizar os representantes das componentes conectadas
  from_representative = find_representative(from);
  to_representative = find_representative(to);
  if (from_representative != to_representative)
  {
    // se for menor
    if (from_representative->number_of_vertices <
to_representative->number_of_vertices)
    {
      to_representative->number_of_vertices +=
from_representative->number_of_vertices;
      from_representative->representative = to_representative;
      from_representative->number_of_edges++;
      from->representative = to_representative;
    }
    else
    { // se for maior ou igual
      from_representative->number_of_vertices +=
to_representative->number_of_vertices;
      to_representative->representative = from_representative;
      to->representative = from_representative;
    }
  }
  else
  {
    // se a representação das palavras for igual
    from_representative->number_of_edges++;
  }

  // adicionar link ao node e atualizar a lista de links
  // from
  link = allocate_adjacency_node();
  if (link == NULL)
  {
    fprintf(stderr, "add_edge: out of memory\n");
    exit(1);
  }
  link->vertex = to;
  link->next = from->head;
  from->head = link;

  // to
```

```
 link = allocate_adjacency_node();
 if (link == NULL)
 {
   fprintf(stderr, "add_edge: out of memory\n");
   exit(1);
 }
 link->vertex = from;
 link->next = to->head;
 to->head = link;

 // save que temos mais um edge
 hash_table->number_of_edges++;
 return;
 /* end code */
}

//
// generates a list of similar words and calls the function add_edge
for each one (done)
//
// man utf8 for details on the uft8 encoding
//

static void break_utf8_string(const char *word, int
*individual_characters)
{
 int byte0, byte1;

 while (*word != '\0')
 {
   byte0 = (int)(*(word++)) & 0xFF;
   if (byte0 < 0x80)
     *(individual_characters++) = byte0; // plain ASCII character
   else
   {
     byte1 = (int)(*(word++)) & 0xFF;
     if ((byte0 & 0b11100000) != 0b11000000 || (byte1 & 0b11000000) !=
0b10000000)
     {
       fprintf(stderr, "break_utf8_string: unexpected UFT-8
character\n");
       exit(1);
     }
```

```
      *(individual_characters++) = ((byte0 & 0b00011111) << 6) | (byte1
& 0b00111111); // utf8 -> unicode
    }
 }
 *individual_characters = 0; // mark the end!
}

static void make_utf8_string(const int *individual_characters, char
word[_max_word_size_])
{
 int code;

 while (*individual_characters != 0)
 {
   code = *(individual_characters++);
   if (code < 0x80)
     *(word++) = (char)code;
   else if (code < (1 << 11))
   { // unicode -> utf8
     *(word++) = 0b11000000 | (code >> 6);
     *(word++) = 0b10000000 | (code & 0b00111111);
   }
   else
   {
     fprintf(stderr, "make_utf8_string: unexpected UFT-8 character\n");
     exit(1);
   }
 }
 *word = '\0'; // mark the end
}

static void similar_words(hash_table_t *hash_table, hash_table_node_t
*from)
{
 static const int valid_characters[] =
      {
// unicode!
       0x2D,
// -
       0x41, 0x42, 0x43, 0x44, 0x45, 0x46, 0x47, 0x48, 0x49, 0x4A, 0x4B,
0x4C, 0x4D,              // A B C D E F G H I J K L M
       0x4E, 0x4F, 0x50, 0x51, 0x52, 0x53, 0x54, 0x55, 0x56, 0x57, 0x58,
0x59, 0x5A,              // N O P Q R S T U V W X Y Z
```

```
      0x61, 0x62, 0x63, 0x64, 0x65, 0x66, 0x67, 0x68, 0x69, 0x6A, 0x6B,
0x6C, 0x6D,             // a b c d e f g h i j k l m
      0x6E, 0x6F, 0x70, 0x71, 0x72, 0x73, 0x74, 0x75, 0x76, 0x77, 0x78,
0x79, 0x7A,             // n o p q r s t u v w x y z
      0xC1, 0xC2, 0xC9, 0xCD, 0xD3, 0xDA,
// Á Â É Í Ó Ú
      0xE0, 0xE1, 0xE2, 0xE3, 0xE7, 0xE8, 0xE9, 0xEA, 0xED, 0xEE, 0xF3,
0xF4, 0xF5, 0xFA, 0xFC, // à á â ã ç è é ê í î ó ô õ ú ü
      0};
 int i, j, k, individual_characters[_max_word_size_];
 char new_word[2 * _max_word_size_];

 break_utf8_string(from->word, individual_characters);
 for (i = 0; individual_characters[i] != 0; i++)
 {
   k = individual_characters[i];
   for (j = 0; valid_characters[j] != 0; j++)
   {
     individual_characters[i] = valid_characters[j];
     make_utf8_string(individual_characters, new_word);
     // avoid duplicate cases
     if (strcmp(new_word, from->word) > 0)
       add_edge(hash_table, from, new_word);
   }
   individual_characters[i] = k;
 }
}

//
// breadth-first search (to be done)
//
// returns the number of vertices visited; if the last one is goal,
following the previous links gives the shortest path between goal and
origin
//

static int breadh_first_search(int maximum_number_of_vertices,
hash_table_node_t **list_of_vertices, hash_table_node_t *origin,
hash_table_node_t *goal)
{
 int read_index = 0, write_index = 1;
 list_of_vertices[0] = origin;
 origin->previous = NULL;
```

```
 origin->visited = 1;
 unsigned int found = 0; // 0-> false | 1-> true

 // enquanto não encontrarmos percorremos tudo
 while (!found)
 {
   if (read_index != write_index)
   {
     adjacency_node_t *link = list_of_vertices[read_index]->head;
     // proximo vertice a ler
     read_index++;
     while (link != NULL)
     {
       if (link->vertex->visited == 0)
       {
         // visitamos um novo vertice
         link->vertex->visited = 1;
         link->vertex->previous = list_of_vertices[read_index - 1];
         list_of_vertices[write_index] = link->vertex;
         // proximo vertice a escrever
         write_index++;
         if (link->vertex == goal)
         {
           found = 1;
           break;
         }
       }
       link = link->next;
     }
   }
   else
   {
     break;
   }
 }
 // Reseta o estado visitado do vértice
 for (int i = 0; i < write_index; i++)
 {
   list_of_vertices[i]->visited = 0;
 }
 // retorna o número de vértices visitados
 return write_index;
}
```

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

// option 1
static void list_connected_component(hash_table_t *hash_table, const
char *word)
{
 /* students code */
 hash_table_node_t *origin = find_word(hash_table, word, 0);
 // se a palavra não existir
 if (origin == NULL)
 {
   printf("list_connected_component: word not found\n");
   return;
 }
 // se a palavra existir
 unsigned int count = 0, numberV;
 unsigned int maximum_number_of_vertices =
find_representative(origin)->number_of_vertices;
 // get some space to creat our list
 hash_table_node_t **list_of_vertices = malloc(sizeof(hash_table_node_t
*) * maximum_number_of_vertices);
 // this function is for option 1 so there is no goal :)
 numberV = breadh_first_search(maximum_number_of_vertices,
list_of_vertices, origin, NULL);
 // print todos os vertices
 for (count; count < numberV; count++)
 {
   printf("%d. %s\n", count, list_of_vertices[count]->word);
 }
 free(list_of_vertices);
 /* end code */
}

//
// 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)
//

// option 2
static void path_finder(hash_table_t *hash_table, const char
*from_word, const char *to_word)
{
 /* students code */
 // basicamente we do the same thing has list_connected_component but
with a goal
 hash_table_node_t *goal = find_word(hash_table, from_word, 0);
 hash_table_node_t *origin = find_word(hash_table, to_word, 0);
 // se a palavra não existir
 if (origin == NULL)
 {
   printf("list_connected_component: word not found\n");
   return;
 }
 // se a palavra existir
 unsigned int count = 0, numberV;
 unsigned int maximum_number_of_vertices =
find_representative(origin)->number_of_vertices;
 // get some space to creat our list
 hash_table_node_t **list_of_vertices = malloc(sizeof(hash_table_node_t
*) * maximum_number_of_vertices);
 // this function is for option 1 so there is no goal :)
 numberV = breadh_first_search(maximum_number_of_vertices,
list_of_vertices, origin, goal);
 // print todos os vertices
 unsigned int n = numberV - 1;
 hash_table_node_t *p = list_of_vertices[numberV - 1];
```

```
 while (p != NULL)
 {
   printf("%d: %s \n", count, p->word);
   count++;
   p = p->previous;
 }

 free(list_of_vertices);
 /* end code */
}

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

static void graph_info(hash_table_t *hash_table)
{
 //
 // complete this
 //
}

// this function prints the hash table to test it just uncomment it on
main
static void print_hash_table(hash_table_t *hash_table)
{
 for (unsigned int i = 0; i < hash_table->hash_table_size; i++)
 {
   printf("Space %d: ", i);
   hash_table_node_t *current = hash_table->heads[i];
   while (current != NULL)
   {
     printf("%s -> ", current->word);
     current = current->next;
   }
   printf("NULL\n");
 }
 return;
}

//
// main program
//
```

```c
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-four-letters.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);
 // print_hash_table(hash_table);
 //  ask what to do
 for (;;)
 {
   fprintf(stderr, "Your wish is my command:\n");
   fprintf(stderr, "  1 WORD       (list the connected component WORD
belongs to)\n");
   fprintf(stderr, "  2 FROM TO    (list the shortest path from FROM to
TO)\n");
   fprintf(stderr, "  3            (terminate)\n");
   fprintf(stderr, "> ");
   if (scanf("%99s", word) != 1)
     break;
   command = atoi(word);
   if (command == 1)
   {
```

```
      if (scanf("%99s", word) != 1)
        break;
      list_connected_component(hash_table, word);
    }
    else if (command == 2)
    {
      if (scanf("%99s", from) != 1)
        break;
      if (scanf("%99s", to) != 1)
        break;
      path_finder(hash_table, from, to);
    }
    else if (command == 3)
      break;
  }
  // clean up
  hash_table_free(hash_table);
  return 0;
}
```

Conclusão

Podemos concluir que conseguimos implementar corretamente as funções que estavam por completar no ficheiro **word_ladder.c**, consolidando os nossos conhecimentos acerca de implementação de hash tables, bem como de conceitos como memory leaks e construção de grafos. Conseguimos também aprimorar ainda mais os nossos conhecimentos da linguagem de programação C, pondo em prática os conteúdos e metodologias dadas aulas práticas e teóricas desta cadeira.

Referências

Para a realização deste trabalho recorremos aos slides disponibilizados na disciplina, que possuem guiões práticos e apontamentos teóricos. Face a dificuldades em algum assunto em específico recorremos também a sites web, mais notalvelmente o **stackoverflow.com.**