universidade de aveiro

Word Ladder

Relatório
15/01/2023

Tiago Sousa Fonseca (107266) – 37.5%
Tomás Sousa Fonseca (107245) – 37.5%
Beatriz Ferreira (107214) – 25%

Universidade de Aveiro
Algoritmos e Estruturas de Dados

universidade de aveiro

Índice

Introdução

Uma “word ladder” é uma sequência de palavras em que duas palavras adjacentes diferem por uma letra. Por exemplo:

- Inglês:

  head → heal → teal → tell → tal → tail

- Português:

  tudo → todo → nodo → nado → nada

Seguindo o princípio estabelecido para a diferenciação de duas palavras, podemos assim estabelecer um componente conectado, ao qual uma palavra pertence.

A partir de vários componentes conectados, é possível estabelecer um caminho entre duas palavras.

Funções Modificadas

hash_table_create:

Cria uma ***"Hash Table"*** através da alocação de memória (faz uso da função ***"malloc"***). Define o tamanho inicial da ***"Hash Table"*** para 50, e inicializa o número de entradas e de arestas para 0. Cria ainda um ***"array"*** de ponteiros para os nós da ***"Hash Table"***, e define-os com o valor ***"NULL"***. Se não houver memória suficiente para alocar a ***"Hash Table"*** ou o ***"array"*** de ponteiros, a função imprime uma mensagem de erro e sai do programa.
Assim, a função retorna um ponteiro para a ***"Hash Table"***.

hash_table_grow:

Aumenta o tamanho da ***"Hash Table"*** através da alocação de memória para um novo ***"array"*** de ponteiros para os nós da nova ***"Hash Table",*** e faz o ***"Rehashing"*** dos elementos da atual ***"Hash Table"*** para o novo ***"array"*** de ponteiros para os nós da ***"Hash Table"***. O novo tamanho da ***"Hash Table"*** é o dobro do tamanho da ***"Hash Table"*** atual.
A função começa por alocar memória para o ***"array"*** de ponteiros para os nós da ***"Hash Table"*** (faz uso da função ***"malloc"***), e define-os com o valor ***"NULL"***. Se não houver memória suficiente para alocar o novo ***"array"*** de ponteiros, a função imprime uma mensagem de erro e sai do programa.
De seguida, ocorre a iteração sobre os elementos da ***"Hash Table"*** atual, e é feito o ***"Rehashing"*** de cada elemento para o novo ***"array"*** (o ***"Rehashing"*** é feito através do resto da divisão entre função de ***"hash" ("crc_32")*** e o novo tamanho da ***"Hash Table"***), sendo este processo necessário para manter o ***"array"*** como sendo uma ***"linked list"***.
Após todos os elementos terem sido adicionados à nova ***"Hash Table"***, a função liberta a memória do antigo ***"array"*** de ponteiros, e atualiza os campos da ***"Hash Table"***.

hash_table_free:

Liberta a memória usada por uma ***"Hash Table"***.
Em primeiro lugar a função itera sobre o ***"array"*** de ponteiros para os nós da ***"Hash Table"***, e para cada ponteiro, liberta a memória usada pela ***"linked list"*** associada. Isto é conseguido através da travessia da ***"linked list"***, e do uso da função ***"free_hash_table_node"***, que liberta a memória alocada de cada nó.
Após a libertação dos nós, é feita a libertação da memória alocada para o ***"array"*** de ponteiros (uso da função ***"free"***). O objetivo de todas estas operações é reduzir a alocação de memória.

find_word:

A função aceita uma ***"Hash Table"***, uma ***"word"***, e uma ***"flag" "insert_if_not_found"*** que decide ou não a inserção da ***"word"*** na ***"Hash Table"***. No final a função retorna um ponteiro para um nó da ***"Hash Table"*** que contém a ***"word"*** dada.

Em primeiro lugar a função calcula o índice da ***"linked list"*** onde a palavra está localizada, depois a função itera sobre a ***"linked list"***, comparando a ***"word"*** de cada nó com a ***"word"*** fornecida como argumento (isto é conseguido através da função ***"strcmp"***). Se a função encontrar o nó com a ***"word"*** coincidente, esta retorna o ponteiro para esse nó.

No caso de a função não encontrar a ***"word"***, e a ***"flag" "insert_if_not_found"*** estiver a 1, a função aloca um novo nó (através da função ***"allocate_hash_table_node"***). Inicializa todos os campos associados a esse nó, e adiciona o nó no início da ***"linked list"***, acabando por incrementar o número de entradas da ***"Hash Table"***.

Se o número de entradas for maior do que a metade do tamanho da ***"Hash Table"***, é chamada a função ***"hash_table_grow"*** que duplica o tamanho da ***"Hash Table"***.

Se a função não encontrar a ***"word"*** coincidente, e a ***"flag" "insert_if_not_found"*** estiver a 0, a função retorna o valor ***"NULL"***.

find_representative:

A função usa uma técnica chamada ***"path_compression"*** para encontrar o representativo de um nó. Esta começa por inicializar um ponteiro ***"representative"*** com o nó passado como argumento. A seguir é iniciado um ***"While Loop"***, e enquanto o campo ***"representative"*** do nó atual não for igual a si mesmo, este atualiza esse mesmo campo de forma a apontar para o ***"representative"*** do nó atual, e continua a iterar até chegar ao nó representativo.

Após atingir o nó representativo, é iniciado outro ***"While Loop"***, e enquanto o campo ***"representative"*** do nó atual não estiver a apontar para o nó representativo, este atualiza o campo ***"representative"*** do nó atual de forma a apontar para o nó representativo. Atualiza ainda o ponteiro do nó atual para o próximo nó no caminho.

Desta maneira, a função comprime o caminho, desde o nó passado como argumento, até ao nó representativo, tornando as futuras pesquisas mais rápidas.

No final, a função retorna o ponteiro para o nó representativo.

add_edge:

A função aceita como argumentos uma ***"Hash Table"***, um ponteiro ***"from"*** para um nó da ***"Hash Table"***, e uma ***"word"***. Resumidamente a função adiciona uma aresta entre o nó ***"from"*** e o nó que contem a ***"word"*** na ***"Hash Table"***.

Em primeiro lugar é chamada a função ***"find_word"*** com objetivo de obter um ponteiro para o nó que contem a ***"word"***, e verificar a existência da ***"word"*** na ***"Hash Table"***. Se a ***"word"*** não estiver presente, a função retorna sem adicionar uma aresta.

Se a ***"word"*** estiver presente, a função incrementa o campo "***number_of_edges"*** da ***"Hash Table"***, ou seja, o número de arestas é incrementado.

De seguida a função cria dois nós adjacentes (através da função ***"allocate_adjacency_node"***), define o campo ***"vertex"*** de cada nó de forma a ligar-se ao outro nó adjacente recém criado, e o campo ***"next"*** é definido para a ***"head"*** atual da lista de adjacência do nó correspondente.

Finalmente, a função usa o algoritmo ***"union-find"*** para agrupar os nós em componentes conexos. A função ***"find_representative"*** é chamada para que seja possível obter um ponteiro para o nó representativo do ***"from"*** e outro ponteiro para o nó representativo do nó correspondente à ***"word"***.

Se os dois nós representativos não forem iguais, a função executa a operação ***"union"***, começando por apontar o nó representativo que contem o menor número de vértices ao nó representativo que contem o maior número de vértices, passando depois por atualizar o número de vértices do nó representativo com mais vértices (é feita a soma dos vértices de ambos os nós representativos, e de seguida esta soma é guardada no campo ***"number_of_vertices"*** do nó representativo com mais vértices).

Desta maneira, a função garante que todas as palavras que estão conectadas com arestas encontram-se de acordo com o componente conexo, e contêm o mesmo valor para o campo ***"number_of_vertices"***.

breadh_first_search:

Esta função é uma implementação do algoritmo ***"breadh_first_search"*** e aceita três argumentos:

1. ***"maximum_number_of_vertices"*** : um inteiro que representa ao número máximo de vértices que a ***"list_of_vertices"*** consegue armazenar;

2. ***"list_of_vertices"*** : uma lista the ponteiros que consegue armazenar os vértices visitados durante a execução do algoritmo;

3. ***"origin"*** : ponteiro que representa o nó original da procura;

4. ***"goal"*** : ponteiro que representa o nó final da procura.

A função começa por guardar o nó ***"origin"*** na ***"list_of_vertices"***, e marca-o como visitado. A seguir é iniciado um ***"While Loop"*** que, enquanto a ***"list_of_vertices"*** não estiver vazia, remove o nó que se encontra na frente da ***"list_of_vertices"***.
É verificado se o nó removido é o ***"goal"***. Se for, a função retorna o número de vértices visitados durante a procura, senão, é feita a iteração pelos nós vizinhos, e qualquer nó não visitado será adicionado à ***"list_of_vertices"*** e marcado como visitado, sendo que ao mesmo tempo o campo ***"previous"*** é atualizado para conter o ponteiro para o nó anterior.
Assim, quando a ***"list_of_vertices"*** estiver vazia, os nós marcados como visitados são desmarcados, e é retornado o número de vértices visitados durante a procura.

list_connected_component:

Esta função aceita dois argumentos, um ponteiro para uma ***"Hash Table"*** e uma ***"word"***.
A função começa por encontrar o nó da ***"Hash Table"*** que contem a ***"word"***, e se não o encontrar retorna.
Caso encontre, a função procura o representativo do componente conexo ao qual a ***"word"*** pertence (usa a função ***"find_representative"***). A seguir encontra o número de nós conectados ao componente através do campo ***"number_of_vertices"*** do nó representativo.
De seguida a função aloca memória para a ***"list_of_vertices"***, sendo o tamanho do ***"array"*** igual ao número de nós no componente conexo. A função ***"breadh_first_search"*** é então chamada, sendo passados como argumentos o número de nós, a ***"list_of_vertices"***, o nó representativo, e um nó ***"NULL"*** como objetivo.
Depois da execução do ***"breadh_first_search"***, a função ***"list_connected_component"*** imprime as ***"word"(s)*** que estão contidas nos nós guardados na ***"list_of_vertices"***, e liberta a memória alocada para a ***"list_of_vertices"***.

path_finder:

Esta função é usada para descobrir o caminho entre duas palavras contidas na ***"Hash_Table"***, sendo que são passados como argumentos as palavras ***"from_word"*** e ***"to_word"***. O primeiro passo é encontrar os nós da ***"Hash Table"*** que correspondem às palavras ***"from_word"*** e ***"to_word"***, e depois é feito o uso da função ***"breadh_first_search"*** para encontrar o caminho entre os dois nós correspondentes. Se um caminho for encontrado, a função imprime-o através da iteração sobre os nós (a iteração é feita através do campo ***"previous"*** que aponta para o nó que o antecede), que nos permite obter as ***"word"(s)*** contidas nos nós.
Se um caminho não for encontrado, a função imprime uma mensagem de erro.
Caso uma das palavras não for encontrada, a função imprime uma mensagem de erro.

connected_component_diameter:

Esta função aceita um ponteiro para um nó da ***"Hash Table"***, e retorna um inteiro que representa o diâmetro do componente conexo.
A função calcula o diâmetro do componente conexo, ou seja, o caminho mais longo entre qualquer de dois nós do componente conexo. Em primeiro lugar a função encontra o nó representativo do nó passado como argumento, depois aloca memória para uma ***"list_of_vertices"***.
Após ter feito isto, é chamada a função ***"breadh_first_search"*** para realizar a travessia do componente conexo e encontrar o caminho mais longo entre os dois nós no componente conexo. Depois de encontra o caminho mais longo, a função liberta a memória alocada para o ***"list_of_vertices"***.
No decorrer de todo este processo a função mantém-se a par do maior diâmetro calculado e do componente conexo associado a esse diâmetro.
Por fim, a função retorna o diâmetro calculado.

graph_info:

A função aceita um ponteiro para uma ***"Hash Table"***. O principal propósito desta função é juntar vários dados estatísticos sobre o grafo representado pela ***"Hash Table"***, incluindo o número de componentes conexos, o número de arestas, a grau médio dos nós, o diâmetro do maior componente conexo, e o maior componente conexo.
Esta começa por inicializar várias variáveis para manter-se a par das estatísticas e aloca memória para uma ***"list_of_vertices"***. Depois esta percorre a ***"Hash Table"*** e executa o algoritmo ***"breadh_first_search"*** para cada nó que contem uma ***"head"*** (ou seja, uma lista de nós vizinhos), de forma a encontrar o componente conexo ao qual o nó pertence.
Nesta fase a função mantem-se a par dos nós visitados, do número de componentes conexos, e do número de arestas no grafo. A função chama ainda a função ***"connected_component_diameter"*** para cada nó na iteração, de forma a encontrar o diâmetro do maior componente conexo, assim como o componente conexo correspondente.
Por fim, a função imprime todas as estatísticas de que se manteve a par, e liberta a memória alocada para a ***"list_of_vertices"***.

universidade de aveiro

Word Ladders

- **[volta] → [torta]:**

  volta → volva → voava → toava → torva → torta

- **[limpo] → [cinco]:**

  limpo → limbo → lombo → rombo → roubo → rouco → rosco → risco → cisco →cinco

- **[urina] → [torta]:**

  urina → crina → china → chita → coita → corta → torta

- **[recear] → [partir]:**

  recear → receai → recebi → receba → receia → recaia → recais → pecais → picais → pirais → parais → partis → partir

- **[aleijar] → [falhada]:**

  aleijar → aleitar → alentar → atentar → atendar → atender → atendes → aterdes → aturdes → aturdis → atureis → atareis → arareis → trareis → toareis → torreis → morreis → mordeis → moldeis → molheis → folheis → folheio → folhedo → folhado → falhado → falhada

- **[malhado] → [jeitoso]:**

  malhado → falhado → folhado → folhedo → folheio → folheis → molheis → moldeis → mordeis → morreis → torreis → toareis → trareis → trameis → tremeis → premeis → premeia → premera → premira → premida → presida → presada → fresada → frisada → fritada → fritara → fritura → feitura → leitura → leitora → leitosa → jeitosa → jeitoso

- **[partir] → [chegar]:**

  partir → partis → partas → cartas → cortas → coutas → chutas → chatas → chagas → chegas → chegar

- **[ingerir] → [falhado]:**

  ingerir → inferir → inferia → inferna → inverna → inverta → inventa → intenta → intenda → entenda → enteada → enteava → entrava → encrava → escrava → escoava → escoara → escorra → escorri → escorai → escoras → esporas → espiras → aspiras → assiras → assaras → assadas → ossadas → ousadas → ougadas → sugadas → segadas → regadas → regidas → retidas → retinas → retinis → retinia → retinha → retenha → retenho → retendo → remendo → gemendo → gemando → gerando → gerindo → ferindo → feriado → ferrado → berrado → borrado → bordado → cordado → cardado → caldado → calhado → falhado

universidade de aveiro

Apêndice

word_ladder.c

```
//
// AED, November 2022 (Tomás Oliveira e Silva)
//
// Second practical assignement (speed run)
//
// Place your student numbers and names here
//   N.Mec. 107266  Name: Tiago Fonseca
//   N.Mec. 107245  Name: Tomás Fonseca
//   N.Mec. 107214  Name: Beatriz Ferreira
//
// 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 belonging to a connected co mponent
//      *) 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 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)
  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;
}

static void hash_table_free(hash_table_t *hash_table)   // FEITO
{
  for(unsigned int i = 0u;i < hash_table->hash_table_size;i++)
  {
    hash_table_node_t *node;
    node = hash_table->heads[i];
    while(node != NULL)
    {
      hash_table_node_t *next_node;
      next_node = node->next;
      free_hash_table_node(node);
      node = next_node;
    }
    free(hash_table->heads[i]);
  }
}

static hash_table_t *hash_table_create(void)    // FEITO
{
  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 = 50u;
  hash_table->number_of_entries = 0u;
  hash_table->number_of_edges = 0u;
  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);
  }
  for(i = 0u;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)   // FEITO
{
  hash_table_node_t **new_hash_array;
  size_t new_size = hash_table->hash_table_size * 2;
  unsigned int index;
  hash_table_node_t *nextNode;
  int conta = 0;

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

  for(unsigned int i = 0; i< new_size;i++)
  {
    new_hash_array[i] = NULL;
  }
```

```
  hash_table->number_of_entries =0;
  for(unsigned int i = 0u; i< hash_table->hash_table_size;i++)
  {
    hash_table_node_t *node;
    node = hash_table->heads[i];
    while(node != NULL)
    {
      index = crc32(node->word) % new_size; // hash function
      nextNode = node->next;

      if(new_hash_array[index]  == NULL)
      {
        new_hash_array[index]  = node;
        node->previous = NULL;
        node->next = NULL;

      }
      else
      {
        new_hash_array[index] ->previous = node;
        node->next = new_hash_array[index] ;
        new_hash_array[index]  = node;
      }
      conta++;
      node = nextNode;
    }
  }

  free(hash_table->heads);
  hash_table->heads = new_hash_array;
  hash_table->hash_table_size = new_size;
  hash_table->number_of_entries = conta;
}

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; // hash function
```

```
  for(node = hash_table->heads[i]; node != NULL; node = node->next)
  {
    if(strcmp(node->word,word) == 0)
    {
      return node;
    }
  }
  if(insert_if_not_found == 1)
  {
    node = allocate_hash_table_node();
    strcpy(node->word,word);
    node->previous = NULL;
    node->head = NULL;
    node->visited = 0u;
    node->representative = node;  // initially, each word is its own representative
    node->number_of_edges = 0u;  // initially, each word has no edges
    node->number_of_vertices = 1u; // initially, each word is a vertex in its own graph
    node->next = NULL;
    if(hash_table->heads[i] == NULL)
    {
      hash_table->heads[i] = node;
    }
    else
    {
      hash_table->heads[i]->previous = node;
      node->next = hash_table->heads[i];
      hash_table->heads[i] = node;
    }
    hash_table->number_of_entries++;

    if(hash_table->number_of_entries > hash_table->hash_table_size / 2u)
    {
      hash_table_grow(hash_table);
    }
    return node;
  }
  return NULL;
}


//
```

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

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

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

  if (to == NULL)
  {
    return;
  }

  hash_table->number_of_edges++;

  //from -> to
  link = allocate_adjacency_node();
  link->next = from->head;
  link->vertex = to;
  from->head = link;
```

```
  //to -> from
  link = allocate_adjacency_node();
  link->next = to->head;
  link->vertex = from;
  to->head = link;

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

  if (from_representative != to_representative) // union
  {
    if (from_representative->number_of_vertices < to_representative->number_of_vertices)
    {
      from_representative->representative = to_representative;
      to_representative->number_of_vertices += from_representative->number_of_vertices;
    }
    else
    {
      to_representative->representative = from_representative;
      from_representative->number_of_vertices += to_representative->number_of_vertices;
    }
  }

  return;
}



//
// 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)
{
  // Create a queue for the BFS funtion
  int head = 0, tail = 0;


  // Place the origin node in the queue, and mark it as visited
  list_of_vertices[tail++] = origin;
  origin->previous = NULL;
  origin->visited = 1;
  // Iterate through the queue until it is empty
  while (head < tail) {
    // Remove the front node from the queue
    hash_table_node_t *node = list_of_vertices[head++];


    // Check if the dequeued node is the goal node
    if (node == goal) {
      for (int i = 0; i < tail; i++) {
        list_of_vertices[i]->visited = 0;
      }
      return tail;  // Found the goal node
    }


    // Iterate through the node's neighbors
   for (adjacency_node_t *neighbor = node->head; neighbor != NULL; neighbor = neighbor->next) {
      if (neighbor->vertex->visited == 0) {
        list_of_vertices[tail++] = neighbor->vertex;
        neighbor->vertex->visited = 1;
        neighbor->vertex->previous = node;
      }
    }
  }


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


  // The goal node was not found
  return tail;
```

```
}



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

static int largest_diameter = 0;
static hash_table_node_t **largest_diameter_example;

static int connected_component_diameter(hash_table_node_t *node)
{
  int diameter = 0;
  hash_table_node_t *representative;

  // get number of nodes connected to the component
  representative = find_representative(node);
  int num_nodes = representative->number_of_vertices;

  // allocate list of vertices
  hash_table_node_t **list_of_vertices = malloc(num_nodes * sizeof(hash_table_node_t *));
  if (list_of_vertices == NULL) { fprintf(stderr, "Out of memory"); }

  // do the bfs traversal
  int num_visited = breadh_first_search(num_nodes, list_of_vertices, representative, NULL);

  // find the longest path
  for (int i = 0; i < num_visited; i++) {
    int path_length = 0;
    hash_table_node_t *current = list_of_vertices[i];
    while (current != NULL) {
      path_length++;
      current = current->previous;
    }
    if (path_length > diameter) {
      diameter = path_length;
    }
    if(diameter > largest_diameter) {
      largest_diameter = diameter;
      largest_diameter_example = list_of_vertices;
    }
```

```
  }

  // free the list of vertices
  free(list_of_vertices);

  return diameter;

}

//
// 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, *representative;

  node = find_word(hash_table, word, 0);
  if (node == NULL)
    return;

  representative = find_representative(node);

  // get number of nodes connected to the component
  int num_nodes = representative->number_of_vertices;

  // allocate list of vertices
  hash_table_node_t **list_of_vertices = malloc(num_nodes * sizeof(hash_table_node_t *));
  if (list_of_vertices == NULL) { fprintf(stderr, "Out of memory"); }

  // do the bfs traversal
  int num_visited = breadh_first_search(num_nodes, list_of_vertices, representative, NULL);

  // print the nodes in the list of vertices
  for (int i = 0; i < num_visited; i++) {
    printf("  %s\n", list_of_vertices[i]->word);
  }

  // free the list of vertices
  free(list_of_vertices);
```

```
}



//
// 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 *from,*to, *goal;


  // find the nodes
  from = find_word(hash_table,from_word,0);
  to = find_word(hash_table,to_word,0);
  goal = to;
  if(from == NULL)
  {
    fprintf(stderr,"Path_finder: word \"%s\" not found\n",from_word);
    return;
  }
  if(to == NULL)
  {
    fprintf(stderr,"Path_finder: word \"%s\" not found\n",to_word);
    return;
  }
  // find a path
  hash_table_node_t **list = (hash_table_node_t **)malloc(hash_table->number_of_entries* sizeof(hash_table_node_t
*));
  if(!breadh_first_search(hash_table->number_of_entries,list,from,to))
  {
    fprintf(stderr,"path_finder: no path found\n");
    return;
  }
  // print the path
  for( ; to != NULL; to = to->previous)
  {
    if (to == goal) {
      printf("%s",to->word);
    } else {
      printf("->%s",to->word);
    }
```

```
  }
  printf("\n");
}



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

static void graph_info(hash_table_t *hash_table)
{
  // get number of nodes connected to the component
  int num_nodes = hash_table->number_of_entries;
  int num_visited = 0;
  hash_table_node_t *representative;

  // allocate list of vertices
  hash_table_node_t **list_of_vertices = malloc(num_nodes * sizeof(hash_table_node_t *));

  // print the number of connected components
  int num_connected_components = 0;
  int num_edges = 0;
  int present = 0;

  // do the bfs traversal
  for(unsigned int i = 0; i < hash_table->hash_table_size; i++)
  {
    if(hash_table->heads[i] != NULL)
    {
      hash_table_node_t *node = hash_table->heads[i];
      while (node != NULL)
      {
        num_visited ++;
        if(node != NULL && node->head != NULL)
        {
          representative = find_representative(node);

          for(int j = 0; j < num_visited; j++)
          {
            if(list_of_vertices[j] == representative)
            {
```

```
            present = 1;
          }
        }
        list_of_vertices[num_visited - 1] = representative;

        if(present == 0)
        {
          num_connected_components++;
        }
        present = 0;
        connected_component_diameter(node);

        num_edges += node->number_of_vertices;
      }
      node = node->next;
    }
  }
}

printf("\n                        STATISTICS                        \n");
printf("__________________________________________________________\n");

printf("Number of words stored: %d\n\n", num_visited);

// print the size of the hash table
printf("Size of the hash_table: %d\n\n", hash_table->hash_table_size);

// print the number of nodes in the graph
printf("Number of nodes in the graph: %d\n\n", num_visited);

printf("Number of connected components: %d\n\n", num_connected_components);

printf("Number of edges in the graph: %d\n\n", num_edges);

// print the average degree of the nodes in the graph
printf("Average degree of the nodes in the graph: %f\n\n", (float)num_edges/num_visited);

// print the diameter of the largest connected component
printf("Diameter of the largest connected component: %d\n\n", largest_diameter);

// print the example of the largest connected component
```

```
  printf("Example of the largest connected component: \n\n");
  for (int i = 0; i < largest_diameter; i++) {
    if (i == 0) {
      printf("%s", largest_diameter_example[i]->word);
    } else {
      printf("->%s", largest_diameter_example[i]->word);
    }
  }

  printf("\n_____\n");

  // free the list of vertices
  free(list_of_vertices);
}

//
// main program
//

int main(int argc,char **argv)
{
  char word[100],from[100],to[100];
  hash_table_t *hash_table;
  hash_table_node_t *node;
  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            (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,to,from);
    }
    else if(command == 3)
    {
      break;
    }
  }
  // clean up
  hash_table_free(hash_table);
  return 0;
}
```

Todo o código usado pode ser encontrado aqui.