

# Word Ladder

**Relatório 08/12/2022**

## Tiago Sousa Fonseca (107266) – 37.5%

**Tomás Sousa Fonseca (107245) – 37.5%**

## Beatriz Ferreira (107214) – 25%

**Universidade de Aveiro Algoritmos e Estruturas de Dados**

**Índice**

[Introdução 3](#_TOC_250005)

Funções Modificadas 4

hash\_table\_create.....................................................................4

hash\_table\_grow 4

hash\_table\_free 5

find\_word 5

find\_representative 6

add\_edge 6

breadh\_first\_search 7

list\_connected\_component 7

path\_finder 7

connected\_component\_diameter 8

graph\_info 8

[Comentários 8](#_TOC_250001)

[Apêndice 10](#_TOC_250000)

# 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 —> tall —> 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”* e/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 uma iteração sobre os elementos da ***“Hash Table”*** atual, e faz 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 encontra 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, e também atualiza 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 ***“breadth\_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”*** : um 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 este 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 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 ***“breadth\_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 ***“breadth\_first\_search”***, a função ***“list\_connected\_component”*** imprime as ***“word”(s)*** que estão contidas nos nós guardados na ***“list\_of\_vertices”***, e liberta o espaço alocado para a ***“list\_of\_vertices”***.

## path\_finder:

Esta função é usada para descobrir o caminho entre duas palavras contidas na ***“Hash\_Table”***, sendo que é passado como argumento as palavras ***“from\_word”*** e o ***“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 ***“breadth\_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 ***“breadth\_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 ***“breadth\_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 maio 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”***.

# 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;

}

//

// 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.