Algoritmos e Estruturas de Dados Ano Letivo 2022/2023

Trabalho 2
Word_ladder

Pedro Rei 107463, P1 - 33%
Cristiano Nicolau 108536, P4 - 33%
Tiago Fonseca 108615, P4 - 33%

Índice

1. Introdução

Este trabalho prático surgiu no contexto da cadeira de Algoritmos e Estruturas de Dados, onde nos foi fornecido um ficheiro com o programa principal (*world_ladder.c*), em que implementámos e completámos as funções já inicializadas pelo docente. Foram-nos também fornecidos ficheiros do tipo *.txt* que contêm uma infinidade de palavras para utilizarmos no nosso programa.

Este *script* será feito com base na matéria dada nas aulas práticas, onde foram abordados diversos conceitos associados à programação em *C*. No entanto esperamos ainda aplicar os conteúdos abordados nas aulas teóricas da cadeira de Algoritmos e Estruturas de Dados. De acordo com o *PowerPoint* onde temos as indicações para este trabalho, teremos de completar partes do código que irão permitir realizar o objetivo do mesmo, que é a transformação de uma palavra inicial para uma final mudando um carater de cada vez.

Com este trabalho, esperamos ficar mais familiarizados com a programação em linguagem *C*, e em todos os processos necessários para a realização deste trabalho prático.

2. Apresentação do problema

O problema consiste em transformar uma palavra com um certo número de letras em outra de igual tamanho no mínimo de passos possível, sendo que cada passo exige a alteração de uma única letra até formar a palavra final que é pretendida. É um tipo de problema de busca em grafos, onde cada palavra é um nó e cada transição entre palavras diferentes é uma aresta. O objetivo é encontrar o caminho de menor comprimento entre a palavra inicial e a palavra final.

3. Análise do código

Nesta parte do relatório iremos abordar as funções fornecidas inicialmente mas que careceram do completamento das mesmas para que o programa fosse funcional.

hash_table_create()

A função começa declarando um ponteiro para uma *Hash Table*, que é uma estrutura de dados que permite armazenar e recuperar elementos rapidamente. Ela funciona dividindo os elementos em "listas encadeadas" usando uma função de *hash*, que mapeia os elementos para os índices das listas.

Relativamente à função, temos também um inteiro sem sinal chamado "i".

Em primeiro lugar é usada a função *malloc* para atribuir memória para a *Hash Table* e atribui o ponteiro para a memória recém atribuída. Se a função *malloc* retornar NULL, indicando que a atribuição de memória falhou, a função imprime uma mensagem de erro e encerra o programa.

Caso contrário a função prossegue, e é definido o número de entradas da *Hash Table* como 0, o número de arestas como 0 e o tamanho como 100.

É usada então a função malloc para designar memória para um *array* de nós da *Hash Table* e atribui o ponteiro para o *array* para o campo "heads" da *Hash Table*. Se a função malloc retornar NULL, indicando que a designação de memória falhou, a função imprime uma mensagem de erro e encerra o programa.

A função usa então uma condição *for* em *loop* para percorrer o *array* de nós da *Hash Table* e define o valor de cada nó como NULL.

Finalmente, a função retorna o ponteiro para a *Hash Table* recém-criada.

```
static hash_table_t *hash_table_create(void)
{
  hash_table_t *hash_table;
  unsigned int i, size=100;

  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->number_of_edges =0;
  hash_table->number_of_entries = 0;
  hash_table->hash_table_size = size;
  hash_table->heads = (hash_table_node_t **) malloc( size * sizeof(hash_table_node_t));
  if ( hash_table->heads == NULL ) {
    fprintf(stderr,"create_hash_table: out of memory\n");
    exit(1);
  }
  for( i = 0; i < hash_table->hash_table_size; i++ ) {
    hash_table->heads[i] = NULL;
  }

  return hash_table;
}
```

Figura 1 – Função hash_table_create()

hash_table_grow()

Este código implementa uma função que amplia o tamanho de uma tabela *hash*.

Quando a tabela fica muito cheia, a função de hash começa a colocar muitos elementos na lista, o que diminui a eficiência da busca. Para resolver esse problema, a tabela pode ser ampliada, adicionando mais listas à tabela.

O código começa definindo duas variáveis locais: '*node*' e '*temp*'. Em seguida, ele coloca um novo conjunto de listas com o tamanho duplicado da tabela atual, armazenando o endereço da primeiro lista encadeada em *hash_table->heads*. Se a atribuição falhar, a função imprime uma mensagem de erro e encerra o programa.

Em seguida, o código efetua a iteração pelas listas novas e inicializa as mesmas como nulas. Em seguida, efetua a iteração pelas listas antigas e move cada elemento para uma lista nova usando a função de *hash*. Finalmente, a função atualiza o tamanho da tabela para o novo tamanho.

```c
static void hash_table_grow(hash_table_t *hash_table)
{
  hash_table_node_t *node, *temp;
  unsigned int i, new_size;


  new_size = (unsigned int)(hash_table->hash_table_size * 2 );

  hash_table->heads = (hash_table_node_t **)malloc(new_size * sizeof(hash_table_node_t *));
  if (hash_table->heads == NULL)
  {
    free(hash_table);
    fprintf(stderr,"hash_table_grow: out of memory\n");
    exit(1);
  }
  for (i = 0; i < new_size; i++)
  {
    hash_table->heads[i] = NULL;
  }
  for (i = 0; i < hash_table->hash_table_size; i++)
  {
    node = hash_table->heads[i];
    while (node != NULL)
    {
      temp = node;
      node = node->next;

      unsigned int new_index = crc32(temp->word) % new_size;
      temp->next = hash_table->heads[new_index];
      hash_table->heads[new_index] = temp;
    }
  }
  hash_table->hash_table_size = new_size;
}
```

Figura 2 – Função hash_table_grow()

hash_table_free()

Este código implementa uma função que dispensa a memória atribuída por uma tabela *hash*.

Para cada índice "i" do vetor *heads*, a função cria um ponteiro *node* que aponta para o primeiro nó da lista no índice i.

Enquanto o ponteiro não for nulo, a função cria um ponteiro temporário denominado *temp*, que aponta para o *node* atual e move-o para o próximo nó da lista.

A função dispensa a memória para temp e repete o processo até *node* ser nulo. Em seguida, a função dispensa a memória para as listas e para a tabela em si, usando a função *free*. Isso retira toda a memória atribuída à tabela hash.

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

Figura 3 – Função hash_table_free()

find_Word()

Em seguida temos esta função, cujo objetivo é procurar determinada palavra numa *Hash Table* e inserir a palavra na tabela se ela não for encontrada.

A função começa declarando uma variável "*node*" e atribuindo a ela o valor da cabeça da lista na posição "i" da matriz de cabeças. Em seguida, realiza uma condição *while* que permance em *loop* enquanto "*node*" for diferente de NULL. Dentro do *loop*, é feita uma comparação entre a palavra armazenada no nó atual ("*node->word*") e a palavra passada como argumento ("*word*"). Se as palavras forem iguais, a função retorna o nó atual. Caso contrário, a variável "*node*" é atualizada para o próximo nó na lista e o *loop* continua.

Se a palavra não for encontrada, a função cria um nó para a palavra e insere o nó na lista na posição "i". Além disso, a função atualiza o contador de colisões e o contador de entradas na tabela e chama a função "hash_table_grow" para verificar se é necessário aumentar o tamanho da tabela. Por fim, a função retorna o nó inserido ou NULL se a palavra não foi encontrada e não foi inserida.

```
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;
  node = hash_table->heads[i];
  while(node != NULL){
    if (strcmp(word,node->word) != 0)
      return node;
    node = node->next;
  }
  if (insert_if_not_found)
  {
    node = allocate_hash_table_node();
    strncpy(node->word,word,_max_word_size_);
    node->next = hash_table->heads[i];
    hash_table->heads[i] = node;

  }

  return node;
}
```

Figura 4 – Função find_Word()

find_representative()

Este código é uma função que encontra o representante de um nó numa estrutura de dados denominada de conjunto disjunto.

Um conjunto disjunto é uma estrutura de dados que mantém um conjunto de elementos divididos em vários conjuntos disjuntos. Cada elemento é representado por um nó com um ponteiro para o seu representante, que é outro nó no conjunto. Se um nó é o próprio representante, o ponteiro para o representante aponta para ele mesmo.

A função começa declarando duas variáveis: "*representative*" e "*next_node*". Em seguida, é realizada uma condição *for* que irá percorrer a lista de nós a partir do nó passado como argumento até chegar ao nó representante. A cada iteração, a variável "*representative*" é atualizada para o próximo nó na lista. Quando a variável "*representative*" aponta para ela mesma, dá-se por terminada a condição *for* e a função retorna *representative*.

A condição *if* que encontramos dentro do *for* é usada para atualizar os ponteiros dos diversos nós na lista e para apontar diretamente para o nó representante. Esse processo tem como objetivo tornar as operações de pesquisa de representante mais rápidas, pois diminui o tamanho da lista que precisa ser percorrida.

Terminada esta condição, a função retorna *representative*.

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


  representative = node;
  next_node = node->representative;

  for(next_node = node; next_node != representative; next_node = node)
  {
    if (next_node == NULL) break;
    representative = next_node;
    next_node = representative->representative;
  }
  return representative;
}
```

Figura 5 – Função find_representative()

add_edge()

A função *add_edge()* é uma função que tem como funcionalidade adicionar uma aresta entre dois nós numa *Hash Table*.

Esta função tem três parâmetros de entrada:

1. uma *Hash Table*, representando o grafo;
2. um nó de origem (*from*), a partir do qual a aresta será criada;
3. uma string (*word*) representando o nó de destino para a aresta.

A função começa chamando outra função "*find_word*", já apresentada e descrita anteriormente neste relatório, com o objetivo de encontrar o nó de destino (*to*) correspondente à palavra dada. Se o nó não existir, a função retorna imediatamente.
Em seguida, ela verifica se já existe uma aresta entre os dois nós. Se já existir, a função retorna imediatamente.

Se não existir uma aresta entre os nós, a função atribui memória para uma nova aresta e adiciona essa aresta à lista de adjacência do nó de origem.
A função origina um aumento do número total de arestas no grafo.

Em seguida, a função encontra os representantes dos conjuntos disjuntos dos nós de origem e destino. Se eles forem diferentes, a função combina os conjuntos, atualizando as informações de conjunto do representante do conjunto com maior número de vértices com as informações do conjunto com menor número de vértices.

```
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;}
  for (link = from->head; link; link = link->next)
  {
    if(link->vertex == to) {return;}
  }

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

  hash_table->number_of_edges++;
  from_representative = find_representative(from);
  to_representative = find_representative(to);
  if (from_representative != to_representative){
    if (from_representative->number_of_vertices < to_representative->number_of_vertices)
    {
      from_representative->representative = to_representative;
      to_representative->number_of_edges += from_representative->number_of_edges;
      to_representative->number_of_vertices += from_representative->number_of_vertices;
    }
    else
    {
      to_representative->representative = from_representative;
      from_representative->number_of_edges += to_representative->number_of_edges;
      from_representative->number_of_vertices += to_representative->number_of_vertices;
    }
  }
}
```

Figura 6 – Função add_edge()

breadh_first_search()

Neste caso verificamos que a função a completar, "*breadh_first_search*", é uma implementação de pesquisa que começa num nó específico (*origin*) e procura até outro nó específico (*goal*) ou até que todos os nós possíveis tenham sido visitados.

A função tem quatro parâmetros de entrada:

1. o número máximo de vértices que podem ser visitados (*maximum_number_of_vertices*);
2. uma lista para armazenar os vértices visitados (*list_of_vertices*);
3. o nó de origem (*origin*);
4. o nó de destino (*goal*).

A função utiliza duas filas: uma para armazenar os nós atuais que estão a ser visitados (*list_of_vertices*) e outra para armazenar os nós vizinhos a serem visitados (*vizinhanca*). Inicialmente o que a função faz é adicionar o nó de origem à fila de vértices visitados, marcando-o como visitado. Se o número máximo de vértices for menor ou igual a zero, a função retorna -1.

Enquanto a fila de vértices visitados não estiver vazia, a função remove o primeiro nó da fila e verifica se é o nó de destino. Caso seja, a função para a pesquisa.
Caso não seja, a função marca o nó atual como visitado e adiciona todos os seus vizinhos não visitados à fila de vértices vizinhos.

No final da pesquisa, a função percorre a lista de vértices visitados, marca-os como não visitados e retorna o tamanho da lista se o nó de destino foi encontrado. Caso isso não aconteça ele irá retornar -1.

```c
static int breadh_first_search(int maximum_number_of_vertices,hash_table_node_t **list_of_vertices,hash_table_node_t *origin,hash_table_node_t *goal)
{
  hash_table_node_t *node;
  adjacency_node_t *vizinhanca;
  int head_list_vertices = 0, tail_list_vertices = 0;
  list_of_vertices[0] = origin;
  origin->visited = 1;

  if (maximum_number_of_vertices <= 0 )
  {
    return -1;
  }

  while (head_list_vertices <= tail_list_vertices)
  {
    node = list_of_vertices[head_list_vertices++];

    if (node == goal){
      break;
    }

    node->visited = 1;

    for (vizinhanca = node->head; vizinhanca; vizinhanca = vizinhanca->next)
    {
      if (!vizinhanca->vertex->visited)
      {
        list_of_vertices[++tail_list_vertices] = vizinhanca->vertex;
        vizinhanca->vertex->visited = 1;
        vizinhanca->vertex->previous = node;
      }
    }
  }

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

  if (node == goal)
  {
    return tail_list_vertices + 1;
  } else {
    return -1;
  }
}
```

Figura 7 – função breadh_first_search()

path_finder()

Nesta situação apresentamos uma função que encontra o caminho mais curto entre dois vértices especificados por uma palavra num grafo representado por uma *Hash Table*.

A função tem três parâmetros de entrada:

1. uma *Hash Table*, que representa o grafo;
2. uma string (*from_word*) representando o nó de origem;
3. uma string (*to_word*) representando o nó de destino.

Inicialmente é chamada a função "*find_word*", já descrita anteriormente, para encontrar os nós correspondentes às palavras de origem e destino. Se algum dos nós não existir, a função exibe a seguinte mensagem de erro ("*Word not found*") e retorna.

Em seguida, a função atribui memória para uma lista de vértices com o tamanho do conjunto disjunto a que o nó de destino pertence e chama a função "*breadh_first_search*", que também já foi apresentada, para encontrar o caminho mais curto entre os dois nós, passando essa lista como parâmetro.

Se a função "*breadh_first_search*" retornar 0, a função exibe uma mensagem informando que nenhum caminho foi encontrado. Se a função retornar um valor diferente de 0, a função exibe o caminho mais curto encontrado, imprimindo cada palavra do caminho na ordem inversa.

Por fim, a função dispensa a memória atribuída para a lista de vértices.

```c
static void path_finder(hash_table_t *hash_table,const char *from_word,const char *to_word)
{
  hash_table_node_t *from, *to;
  from = find_word(hash_table, from_word, 0);
  to = find_word(hash_table, to_word, 0);

  if (from == NULL || to == NULL)
  {
    fprintf(stderr, "\nWord not found:\n");
    return ;
  }

  int vertices = find_representative(to)->number_of_vertices;
  hash_table_node_t **list = ((hash_table_node_t* )malloc(vertices * sizeof(hash_table_node_t)));
  int lenght = breadh_first_search(vertices, list, to, from);

  if (lenght == 0)
  {
    printf("No path was found from %s to %s\n", from_word, to_word);
  }
  else
  {
   printf("Shortest path from %s to %s:\n", from_word, to_word);
    for (int i = lenght - 1; i >= 0; i--)
    {
      printf("  %s\n", list[i]->word);
    }
  }
}
```

Figura 8 – Função path_finder()

4. Compilação e resultados

Compilação

Para compilar o nosso trabalho temos em mão na realidade um processo muito simples. É nos fornecido um *make file* com o necessário para compilar o programa e apenas precisamos de escrever no terminal "*make word_ladder*". Importante salientar que, para a compilação funcionar, é necessário recorrer a um sistema operativo *Linux* (sendo que nós usamos *Linux-Ubuntu*) ou então usando uma *virtual machine*.
Na imagem em baixo temos um exemplo do processo de compilação.

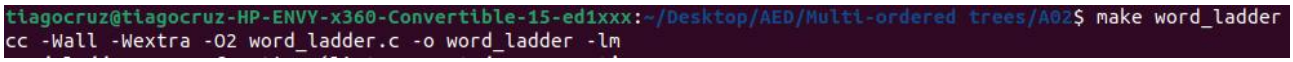

Figura 9 – Compilação do programa

Após a realização da compilação apenas é necessário mais um passo para correr o programa, pelo que temos de usar o seguinte comando:

- *./word_ladder*

Resultados

Em termos de resultados não nos é possível mostrar realmente as funcionalidades do nosso programa pelo facto de não termos completado a globalidade dos programas. O que temos realizado representa uma grande parte do trabalho necessário para o funcionamento do programa, no entanto apenas com a totalidade das funções feitas é que é possível mostrar resultados. Na imagem 10 iremos apresentar o *output* que nos é dado quando tentamos encontrar o caminho mais curto de bem para mal, assim como acima da imagem iremos representar qual seria o resultado suposto.
[0] bem
[1] tem
[2] teu
[3] meu
[4] mau
[5] mal

Figura 10 – Resultados obtidos

5. Conclusão

Na nossa opinião, o trabalho desenvolvido no *script word_ladder* foi bem-sucedido, visto que as funções que completamos foram feitas e encontram-se funcionais sem erros cometidos à primeira vista. Por outro lado, não foi possível completar todas as funções propostas, infelizmente, no entanto consideramos que esse facto não desvaloriza de todo aquilo que conseguimos fazer neste trabalho.

É de opinião geral que este foi provavelmente o projeto mais difícil de executar devido à sua elevada complexidade e ao número de funções a completar. No entanto, com alguma ajuda do professor nas aulas práticas, tanto em dúvidas específicas como em exercícios feitos em aula, fomos capazes de superar os problemas iniciais que não nos permitiam avançar no trabalho. Também foi bastante útil a compreensão inicial do código fornecido para depois saber como o completar, e depois conforme fomos trabalhando ficamos mais familiarizados com o código, tornando mais simples a sua implementação.

Em suma, consideramos que conseguimos alcançar os objetivos propostos pelo trabalho. Todo o projeto permitiu pormos em prática tanto os conhecimentos adquirido nas aulas teóricas como a experiência adquirida através dos exercícios realizados nas aulas práticas.

6. Bibliografia

Uma das ferramentas mais utilizadas para a realização deste trabalho foi, mais uma vez, o PowerPoint fornecido e utilizado pelo professor durante as aulas teóricas e práticas, que se encontra disponível na página do *e-learning* da respetiva unidade curricular.

Alguns dos sites que consultamos foram os seguintes:

- https://pt.stackoverflow.com/
- https://www.javatpoint.com/hash-table
- https://www.geeksforgeeks.org/

7. Código

Por último iremos apresentar todo o código fornecido pelo professor e que nós completamos.

```
#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 compo-
nent this vertex belongs to
  int number_of_vertices;            // number of vertices of the conected compo-
nent (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 hash_table_t *hash_table_create(void)
{
  hash_table_t *hash_table;
  unsigned int i, size=100;

  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->number_of_edges =0;
  hash_table->number_of_entries = 0;
  hash_table->hash_table_size = size;
  hash_table->heads = (hash_table_node_t **) malloc( size * sizeof(hash_ta-
ble_node_t));
  if ( hash_table->heads == NULL ) {
    fprintf(stderr,"create_hash_table: out of memory\n");
    exit(1);
  }
  for( i = 0; i < hash_table->hash_table_size; i++ ) {
    hash_table->heads[i] = NULL;
```

```
  }

  return hash_table;
}

static void hash_table_grow(hash_table_t *hash_table)
{
  hash_table_node_t *node, *temp;
  unsigned int i, new_size;


  new_size = (unsigned int)(hash_table->hash_table_size * 2 );

  hash_table->heads = (hash_table_node_t **)malloc(new_size * sizeof(hash_ta-
ble_node_t *));
  if (hash_table->heads == NULL)
  {
    free(hash_table);
    fprintf(stderr,"hash_table_grow: out of memory\n");
    exit(1);
  }
  for (i = 0; i < new_size; i++)
  {
    hash_table->heads[i] = NULL;
  }
  for (i = 0; i < hash_table->hash_table_size; i++)
  {
    node = hash_table->heads[i];
    while (node != NULL)
    {
      temp = node;
      node = node->next;

      unsigned int new_index = crc32(temp->word) % new_size;
      temp->next = hash_table->heads[new_index];
      hash_table->heads[new_index] = temp;
    }
  }
  hash_table->hash_table_size = new_size;
}



static void hash_table_free(hash_table_t *hash_table)
{
  hash_table_node_t *temp,*node;
  for (unsigned int i = 0; i < hash_table->hash_table_size; i++){
    node = hash_table->heads[i];
    while (node != NULL)
```

```
    {
      temp = node;
      node = node->next;
      free(temp);
    }
  }
  free(hash_table->heads);
  free(hash_table);
}

static hash_table_node_t *find_word(hash_table_t *hash_table,const char *word,int
insert_if_not_found)
{
  hash_table_node_t *node;
  unsigned int i;
  i = crc32(word) % hash_table->hash_table_size;
  node = hash_table->heads[i];
  while(node != NULL){
    if (strcmp(word,node->word) != 0)
      return node;
    node = node->next;
  }
  if (insert_if_not_found)
  {
    node = allocate_hash_table_node();
    strncpy(node->word,word,_max_word_size_);
    node->next = hash_table->heads[i];
    hash_table->heads[i] = node;

  }

  return node;
}


//
// 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;


  representative = node;
  next_node = node->representative;

  for(next_node = node; next_node != representative; next_node = node)
  {
```

```
    if (next_node == NULL) break;
    representative = next_node;
    next_node = representative->representative;
  }
  return representative;
}

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

  to = find_word(hash_table,word,0);

  if (to==NULL){ return;}
  for (link = from->head; link; link = link->next)
  {
    if(link->vertex == to) {return;}
  }

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

  hash_table->number_of_edges++;
  from_representative = find_representative(from);
  to_representative = find_representative(to);
  if (from_representative != to_representative){
    if (from_representative->number_of_vertices < to_representative->number_of_ver-
tices)
    {
      from_representative->representative = to_representative;
      to_representative->number_of_edges += from_representative->number_of_edges;
      to_representative->number_of_vertices += from_representative->number_of_ver-
tices;
    }
    else
    {
      to_representative->representative = from_representative;
      from_representative->number_of_edges += to_representative->number_of_edges;
      from_representative->number_of_vertices += to_representative->number_of_ver-
tices;
    }
  }
}
```

```
//
// 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)
{
  hash_table_node_t *node;
  adjacency_node_t *vizinhanca;
  int head_list_vertices = 0, tail_list_vertices = 0;
  list_of_vertices[0] = origin;
  origin->visited = 1;

  if (maximum_number_of_vertices <= 0 )
  {
    return -1;
  }

  while (head_list_vertices <= tail_list_vertices)
  {
    node = list_of_vertices[head_list_vertices++];

    if (node == goal){
      break;
    }

    node->visited = 1;

    for (vizinhanca = node->head; vizinhanca; vizinhanca = vizinhanca->next)
    {
      if (!vizinhanca->vertex->visited)
      {
        list_of_vertices[++tail_list_vertices] = vizinhanca->vertex;
        vizinhanca->vertex->visited = 1;
        vizinhanca->vertex->previous = node;
      }
    }
  }

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

  if (node == goal)
  {
    return tail_list_vertices + 1;
  } else {
    return -1;
  }
```

```
}


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

static void list_connected_component(hash_table_t *hash_table,const char *word)
{
  hash_table_node_t *node = find_word(hash_table, word, 0);
  hash_table_node_t *representative = find_representative(node);

  int len_list = representative->number_of_vertices;
  hash_table_node_t **list = ((hash_table_node_t*)malloc(len_list*sizeof(hash_ta-
ble_node_t)));

  int number_of_vertices = breadh_first_search(len_list, list, node, NULL);

  for (int i=0; i < number_of_vertices; i++){
    printf("%s\n", list[i]->word);
  }

  free(list);
}


//
// compute the diameter of a connected component (optional)
//
/*
static int largest_diameter;
static hash_table_node_t **largest_diameter_example;

static int connected_component_diameter(hash_table_node_t *node)
{
  int diameter;

  //
  // complete this
  //
  return diameter;
}
*/

//
// find the shortest path from a given word to another given word (to be done)
//
```

```
static void path_finder(hash_table_t *hash_table,const char *from_word,const char
*to_word)
{
  hash_table_node_t *from, *to;
  from = find_word(hash_table, from_word, 0);
  to = find_word(hash_table, to_word, 0);

  if (from == NULL || to == NULL)
  {
    fprintf(stderr, "\nWord not found:\n");
    return ;
  }

  int vertices = find_representative(to)->number_of_vertices;
  hash_table_node_t **list = ((hash_table_node_t* )malloc(vertices *
sizeof(hash_table_node_t)));
  int lenght = breadh_first_search(vertices, list, to, from);

  if (lenght == 0)
  {
    printf("No path was found from %s to %s\n", from_word, to_word);
  }
  else
  {
   printf("Shortest path from %s to %s:\n", from_word, to_word);
    for (int i = lenght - 1; i >= 0; i--)
    {
      printf("  %s\n", list[i]->word);
    }
  }
}


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

static void graph_info(hash_table_t *hash_table)
{
  printf("\nNodes: %u\nEdges: %u\n",
      hash_table->number_of_entries,
      hash_table->number_of_edges);
}


//
// 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,from,to);
    }
    else if(command == 3)
      break;
  }
  // clean up
  hash_table_free(hash_table);
  return 0;
}
```