deti universidade de aveiro
departamento de electrónica, telecomunicações e informática

ALGORITMOS E ESTRUTURAS DE DADOS

Word Ladder

Licenciatura em Engenharia de Computadores e Informática

Gonçalo Cunha – 33,3%

N. Mec.: 108352

Guilherme Santos - 33,3%

N. Mec.: 107961

João Gaspar - 33,3%

N. Mec.: 107708

Índice

O que é uma Word Ladder?

Word Ladder é uma sequência de palavras em que cada palavra difere das adjacentes por uma letra. Cada passo consiste na passagem de uma palavra para uma adjacente. Podemos observar duas sequências exemplo entre as palavras bem e mal, assim como entre tudo e nada.

bem → **t**em → te**u** → **m**eu → m**a**u → ma**l**

tudo → t**o**do → **n**odo → n**a**do → nad**a**

Como cada passo muda apenas uma letra, o número de passos deve de ser pelo menos a distância *Hamming* entre as duas palavras. O objetivo deste trabalho é completar um programa que permita desenvolver um jogo automaticamente, ou seja que o próprio programa seja o jogador e que assim o sendo consiga apresentar de forma quase imediata a solução para o *Word Ladder* pretendido.

Objetivo

O objetivo deste trabalho é completar um programa que construa uma *Word Ladder* entre duas palavras de comprimento igual, registando o número de passos dados nesta.

Também deve ser possível criar um grafo em que se armazene as diferentes palavras como vértices, e caminhos entre palavras nos lados.

Funções Desenvolvidas

```
static hash_table_t *hash_table_create(void)
```

Esta função cria uma *hash table* e executa as seguintes etapas:

1. Declara um instante da estrutura *hash_table_t*.
2. Cria espaço na memória para a *hash_table* usando a função *malloc().* Se a alocação de memória falhar, imprime uma mensagem de erro para *stderr* e sai do programa.
3. Define o tamanho da *hash_table* para 200.
4. Define o campo *number_of_entries* e *number_of_edges* como 0.
6. Reserva também espaço na memória para o campo *heads* de *hash_table*, que é um *array* de ponteiros para *nodes*. Se a alocação de memória falhar, imprime uma mensagem de erro para *stderr* e sai do programa.
7. Inicializa cada elemento do *array heads* como nulo usando um *loop for*.
8. Por fim, retorna a *hash_table*.

Observação:
- Nesta função, *fprintf(stderr,"create_hash_table: out of memory \n")* e *exit(1)* são usados para tratamento de erros. A função imprimirá a mensagem de erro *"create_hash_table: out of memory"* no fluxo de erro e sairá do programa com o código de saída 1.

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

Esta função duplica o tamanho de uma *hash table*, executando as seguintes etapas:

1. Incrementa o contador *GrowNumber*.
2. Declara uma variável *old_hash_table_size* que armazena o tamanho atual da *hash table* e outra variável *old_heads* que armazena a matriz atual de ponteiros para os nós da *hash table*.
3. Duplica o tamanho da *hash_table*.
4. Aloca memória para o campo *heads* de *hash_table* com o novo tamanho.
5. Inicializa cada elemento do array *heads* como nulo usando um *loop for*.
6. Caso a alocação de memória falhar, imprime uma mensagem de erro para *stderr* e sai do programa.
7. Usa outro *loop for* para percorrer o antigo *array heads* e reinsere cada nó no novo array *heads* usando a função *crc32()* para calcular o índice no qual o nó deve ser inserido.
8. Finalmente, liberta a memória alocada para o antigo *array heads* usando a função *free()*.

Observação:
- A função *crc32* é usada para calcular o índice de onde o nó deve ser colocado na nova *hash table*. Esta é uma função da *hash table* que pega uma palavra e retorna um *unsigned int* de 32 bits que é usado como índice para a *hash table*.

```
static void hash_table_free(hash_table_t *hash_table)
```

Esta função é responsável por libertar a memória alocada para a *hash table*, passando por todos os elementos da tabela e libertando a memória de cada elemento, para então libertar a memória alocada para a própria estrutura da tabela.
É composta pelas seguintes etapas:

1. Primeiramente usa um *loop for* para percorrer o campo *heads* da variável *hash_table*.
2. Para cada elemento não nulo do array *heads*, cria uma variável *node* que é definida para o elemento e outra variável *next_node* que é definida para o próximo nó.
3. Usa outro *loop while* que liberta a memória alocada para o nó atual e, em seguida, define a variável *node* como a variável *next_node*.
4. Após o *loop for*, liberta a memória alocada para o *array heads* usando a função *free()*.
5. Finalmente, liberta a memória alocada para a variável *hash_table* usando a função *free()*.

```
static hash_table_node_t *find_word(hash_table_t *hash_table,const char
*word,int insert_if_not_found)
```

Esta função encontra uma palavra na *hash table* e é composta pelas seguintes etapas:

1. Declara uma variável *node* do tipo *hash_table_node_t** e define-a como nula.
2. Calcula o índice da palavra no array *heads* da *hash table* usando a função *crc32()* e o operador módulo.
3. Usa um *loop for* para percorrer a *linked list* de nós no índice calculado.
4. Dentro do *loop for*, compara a word do nó atual com a word passada como parâmetro usando a função *strcmp()*. Se forem iguais, retorna o nó atual.
5. Se a palavra não for encontrada na *hash table* e o parâmetro *insert_if_not_found* for definido como *true*, verifica se o comprimento da palavra é menor que *_max_word_size_*.
6. Se a palavra for menor que *_max_word_size_*, cria um nó usando a função *allocate_hash_table_node()* e copia a palavra passada como parâmetro para o campo word do novo nó usando a função *strncpy()*.
7. Define o campo *representative* do novo nó para si mesmo, inicializa os campos *next*, *previous*, *number_of_edges*, *number_of_vertices*, *visited*, *head* e anexa o novo nó à frente da *linked list* no índice calculado no array *heads*.

8. Incrementa o campo *number_of_entries* da variável *hash_table*.
9. Se o campo *number_of_entries* da variável *hash_table* for maior que 70% do campo *hash_table_size*, chama a função *hash_table_grow()* e aumenta o tamanho da *hash table*.
10. Retorna o novo nó.
11. Se a palavra não for encontrada na *hash table* e o parâmetro *insert_if_not_found* for definido como *false* ou a palavra for maior que *_max_word_size_*, a função retornará nulo.

```
static hash_table_node_t *find_representative(hash_table_node_t *node)
```

Esta função encontra o nó representativo de um determinado nó através dos seguintes passos:

1. Declara uma variável *representative* e *next_node* do tipo *hash_table_node_t** e define-as para o parâmetro *node* fornecido.
2. Usa um loop for para percorrer o campo *representative* da variável *representative* até atingir o nó cujo campo *representative* aponta para si mesmo. Este nó é o nó representativo do conjunto.
3. Usa outro loop for para percorrer o campo *representative* do parâmetro *node* até atingir o nó representativo encontrado na etapa 2.
4. Dentro do segundo loop for, define o campo *representative* do nó atual para o nó representativo.
5. Por fim, retorna o nó *representative*.

```
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)
```

Esta função faz uso de um algoritmo do tipo *Breadth-First Search* e executa as seguintes etapas:

1. Inicializa a variável *found* a 0, que será usada para indicar se o vértice *goal* foi encontrado, e inicializa também a variável *visited_count* a 0, que será usada para rastrear o número de vértices visitados.
2. Define o vértice atual como o vértice *origin*, define o vértice *visited* do vértice *origin* como 1 e incrementa a variável *visited_count*.
3. Usa um *loop while* para ir iterando os vértices do grafo a partir do *current_vertex* até que o vértice atual seja nulo.
4. Dentro do *loop while*, verifica se o *current_vertex* é o vértice *goal*. Se for, define a variável *found* como 1 e sai do *loop*.

5. De seguida, cria uma variável, *current_edge*, que é definida como o campo *head* do *current_vertex*.
6. Usa outro loop while para percorrer a linked list de arestas do vértice atual.
7. Dentro do segundo loop while, cria uma variável *adjacent_vertex* que é definida como o campo *vertex* do *current_edge*.
8. Verifica se o *adjacent_vertex* foi visitado. Caso não, define o campo *visited* do *adjacente_vertex* para 1, define o campo *previous* do *adjacent_vertex* para o *current_vertex*, incrementa a variável *visited_count*, define o *current_vertex* para o *adjacent_vertex* e sai do loop.
9. Se *current_edge* for nulo, define o *current_vertex* como *current_vertex -> previous*.
10. Após os loops while, ele usa um loop for para redefinir o campo *visited* e *previous* de todos os vértices no array *list_of_vertices* para 0.
11. Por fim, a função retorna a variável *visited_count*, que representa o número de vértices visitados durante a busca.

```
void hash_table_stats(hash_table_t * hash_table)
```

O código é uma função chamada hash_table_stats() que recebe com argumento, um ponteiro para a estrutura hash_table_t.

1. Primeiro, verifica se o ponteiro passado como argumento é nulo. Se for, imprime uma mensagem de erro e sai da função antecipadamente.
2. Em seguida, inicializa várias variáveis, incluindo *i*, *chain_count*, *chain_length*, *min_chain_length*, *max_chain_length*, *total_chain_length* e *current*.
3. Inicia um loop for que itera sobre todos os elementos no array *hash_table_size* da estrutura.
4. Dentro do loop for, define *chain_count* e *chain_length* como 0 e *current* para o início da linked list atual.
5. Em seguida, inicia um loop while que continua até que o *current* seja nulo.
6. Dentro do loop while, incrementa *chain_count* e *chain_length* em 1 e atribui *current->next* a *current*.
7. Após o loop while, ele verifica se *chain_length* é maior que 0. Se for, verifica se *chain_length* é menor que *min_chain_length* e, se for, atribui *chain_length* a *min_chain_length*. Em seguida, ele verifica se *chain_length* é maior que *max_chain_length* e, se for, atribui *chain_length* a *max_chain_length*. Por fim, adiciona *chain_length* a *total_chain_length*.
8. Após o loop for, ele verifica se *hash_table->number_of_entries* é igual a 0. Caso seja, imprime uma mensagem de erro e sai da função.
9. Por fim, imprime as estatísticas.

Código em C

```
//
// AED, November 2022 (Tomás Oliveira e Silva)
//
// Second practical assignement (speed run)
//
// Place your student numbers and names here
//   N.Mec. 108352  Name: Gonçalo Cunha
//   N.Mec. 107961  Name: Guilherme Santos
//   N.Mec. 107708  Name: João Gaspar
//
// Do as much as you can
//   1) MANDATORY: complete the hash table code
//      *) hash_table_create---done
//      *) hash_table_grow---done
//      *) hash_table_free---done
//      *) find_word---done
//      +) 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---done
//      *) add_edge---done
//   3) RECOMMENDED: implement breadth-first search in the graph
//      *) breadh_first_search---done
//   4) RECOMMENDED: list all words belonginh to a connected component
//      *) breadh_first_search
//      *) list_connected_component
//   5) RECOMMENDED: find the shortest path between to words
//      *) breadh_first_search
//      *) path_finder
//      *) test the smallest path from bem to mal
//         [ 0] bem
//         [ 1] tem
//         [ 2] teu
//         [ 3] meu
//         [ 4] mau
//         [ 5] mal
//      *) find other interesting word ladders
//   6) OPTIONAL: compute the diameter of a connected component and list
the longest word chain
//      *) breadh_first_search
//      *) connected_component_diameter
//   7) OPTIONAL: print some statistics about the graph
//      *) graph_info
//   8) OPTIONAL: test for memory leaks
```

```c
//

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


//
// static configuration
//

#define _max_word_size_  32


//
// data structures (SUGGESTION --- you may do it in a different way)
//

typedef struct adjacency_node_s  adjacency_node_t;
typedef struct hash_table_node_s hash_table_node_t;
typedef struct hash_table_s      hash_table_t;

struct adjacency_node_s
{
  adjacency_node_t *next;            // link to th enext adjacency list
node
  hash_table_node_t *vertex;         // the other vertex
};

struct hash_table_node_s
{
  // the hash table data
  char word[_max_word_size_];        // the word
  hash_table_node_t *next;           // next hash table linked list node
  // the vertex data
  adjacency_node_t *head;            // head of the linked list of
adjancency edges
  int visited;                       // visited status (while not in use,
keep it at 0)
  hash_table_node_t *previous;       // breadth-first search parent
  // the union find data
  hash_table_node_t *representative; // the representative of the
connected component this vertex belongs to
  int number_of_vertices;            // number of vertices of the
conected component (only correct for the representative of each connected
component)
  int number_of_edges;               // number of edges of the conected
component (only correct for the representative of each connected
component)
```

```
};

struct hash_table_s
{
  unsigned int hash_table_size;      // the size of the hash table array
  unsigned int number_of_entries;    // the number of entries in the hash
table
  unsigned int number_of_edges;      // number of edges (for information
purposes only)
  hash_table_node_t **heads;         // the heads of the linked lists
};


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

static adjacency_node_t *allocate_adjacency_node(void)
{
  adjacency_node_t *node;

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

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

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

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

static void free_hash_table_node(hash_table_node_t *node)
```

```
{
  free(node);
}


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

unsigned int crc32(const char *str)
{
  static unsigned int table[256];
  unsigned int crc;

  if(table[1] == 0u) // do we need to initialize the table[] array?
  {
    unsigned int i,j;

    for(i = 0u;i < 256u;i++)
      for(table[i] = i,j = 0u;j < 8u;j++)
        if(table[i] & 1u)
          table[i] = (table[i] >> 1) ^ 0xAED00022u; // "magic" constant
        else
          table[i] >>= 1;
  }
  crc = 0xAED02022u; // initial value (chosen arbitrarily)
  while(*str != '\0')
    crc = (crc >> 8) ^ table[crc & 0xFFu] ^ ((unsigned int)*str++ << 24);
  return crc;
}

static hash_table_t *hash_table_create(void)
{
  hash_table_t *hash_table;
  unsigned int i;

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

  hash_table->hash_table_size = 200u;
  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; hash_table->hash_table_size;i++){
    hash_table->heads[i] == NULL;
  }

  return hash_table;
}

int GrowNumber = 0;

static void hash_table_grow(hash_table_t *hash_table)
{
    GrowNumber++;
    unsigned int old_hash_table_size = hash_table->hash_table_size;
    hash_table_node_t **old_heads = hash_table->heads;

    hash_table->hash_table_size *= 2u;
    hash_table->heads = (hash_table_node_t **)malloc(hash_table-
>hash_table_size * sizeof(hash_table_node_t *));

    for(unsigned int i = 0u; i < hash_table->hash_table_size; i++) {
        hash_table->heads[i] = NULL;
    }

    if(hash_table->heads == NULL) {
        fprintf(stderr,"hash_table_grow: out of memory");
        exit(1);
    }

    for(unsigned int i = 0u; i < old_hash_table_size; i++) {
        hash_table_node_t *node = old_heads[i];
        while(node != NULL) {
            hash_table_node_t *next_node = node->next;

            size_t index = crc32(node->word) % hash_table-
>hash_table_size;
            node->next = hash_table->heads[index];
            hash_table->heads[index] = node;

            node = next_node;
        }
    }
    free(old_heads);
}

static void hash_table_free(hash_table_t *hash_table)
{
```

```
  for(unsigned int i = 0u; i < hash_table->hash_table_size;i++){
    if(hash_table->heads[i] != NULL)
    {

      //Ciclo while para apagar os nós 1 a 1
      hash_table_node_t *node = hash_table->heads[i];
      while( node != NULL){
        hash_table_node_t *next_node = node->next;
        free(node);
        node = next_node;
      }

    }
  }
  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 = NULL;
    unsigned int i = crc32(word) % hash_table->hash_table_size;

    for(node = hash_table->heads[i]; node != NULL; node = node->next) {
        if(strcmp(node->word, word) == 0) {
            return node;
        }
    }
    if(insert_if_not_found && strlen(word) < _max_word_size_) {
        node = allocate_hash_table_node();
        strncpy(node->word, word, _max_word_size_);
        node->representative = node;
        node->next = hash_table->heads[i];
        node->previous = NULL;
        node->number_of_edges = 0;
        node->number_of_vertices = 1;
        node->visited = 0;
        node->head = NULL;
        hash_table->heads[i] = node;
        hash_table->number_of_entries++;
        if(hash_table->number_of_entries > hash_table-
>hash_table_size*0.70) {
            hash_table_grow(hash_table);
        }
        return node;
    }
    return NULL;
}
```

```c
// Function to print the hash table stats

void hash_table_stats(hash_table_t * hash_table) {
  unsigned int i;
  unsigned int chain_count;
  unsigned int chain_length;
  unsigned int min_chain_length = UINT_MAX;
  unsigned int max_chain_length = 0;
  unsigned int total_chain_length = 0;
  hash_table_node_t * current;
  FILE * fp;

  if (!hash_table) {
    printf("Error: Hash Table is NULL\n");
    return;
  }

  for (i = 0; i < hash_table -> hash_table_size; i++) {
    chain_count = 0;
    chain_length = 0;
    current = hash_table -> heads[i];
    while (current != NULL) {
      chain_count++;
      chain_length++;
      current = current -> next;
    }

    if (chain_length > 0) {
      if (chain_length < min_chain_length)
        min_chain_length = chain_length;
      if (chain_length > max_chain_length)
        max_chain_length = chain_length;
      total_chain_length += chain_length;
    }
  }

  if (hash_table -> number_of_entries == 0) {
    printf("Error: Hash Table is empty\n");
    return;
  }

  fp = fopen("hash_table_stats.json", "w");
  if (!fp) {
    printf("Error: Unable to open file for writing\n");
    return;
  }

  fprintf(fp, "{\n");
```

```
  fprintf(fp, "\t\"number_of_entries\": %u,\n", hash_table ->
number_of_entries);
  fprintf(fp, "\t\"number_of_edges\": %u,\n", hash_table ->
number_of_edges);
  fprintf(fp, "\t\"minimum_chain_length\": %u,\n", min_chain_length);
  fprintf(fp, "\t\"maximum_chain_length\": %u,\n", max_chain_length);
  fprintf(fp, "\t\"average_chain_length\": %f\n", (double)
total_chain_length / (double) hash_table -> hash_table_size);
  fprintf(fp, "}\n");
  fclose(fp);
}

//
// 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;

  for(representative = node; representative != representative-
>representative; representative = representative->representative);

  for( ; node != representative; node = next_node){
    next_node = node->representative;
    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);
  //
  // complete this
  //
}


//
// 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)
//
```

```c
// returns the number of vertices visited; if the last one is goal,
following the previous links gives the shortest path between goal and
origin
//

static int breadh_first_search(int maximum_number_of_vertices,
hash_table_node_t ** list_of_vertices, hash_table_node_t * origin,
hash_table_node_t * goal)
{
  int found = 0;
  int visited_count = 0;
  hash_table_node_t * current_vertex = origin;
  origin -> visited = 1;
  visited_count++;

  while (current_vertex != NULL) {
    if (current_vertex == goal) {
      found = 1;
      break;
    }

    adjacency_node_t * current_edge = current_vertex -> head;
    while (current_edge != NULL) {
      hash_table_node_t * adjacent_vertex = current_edge -> vertex;
      if (!adjacent_vertex -> visited) {
        adjacent_vertex -> visited = 1;
        adjacent_vertex -> previous = current_vertex;
        current_vertex = adjacent_vertex;
        visited_count++;
        break;
      }
      current_edge = current_edge -> next;
    }
    if (current_edge == NULL) {
      current_vertex = current_vertex -> previous;
    }
  }

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

  return visited_count;
}


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

```
//

static void list_connected_component(hash_table_t *hash_table,const char
*word)
{
  //
  // complete this
  //
}


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

static int largest_diameter;
static hash_table_node_t **largest_diameter_example;

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

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


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

static void path_finder(hash_table_t *hash_table,const char
*from_word,const char *to_word)
{
  //
  // complete this
  //
}


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

static void graph_info(hash_table_t *hash_table)
{
```

```
  //
  // complete this
  //
}


//
// 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;
}
```