universidade de aveiro

deti departamento de eletrónica, telecomunicações e informática

Trabalho Word Ladder

Algoritmos e Estruturas de Dados

Universidade de Aveiro

Licenciatura em Engenharia de Computadores e Informática

André Silva 98651

50%

Isaac Moura 105065

50%

15 janeiro 2023

Índice

Introdução ao Problema

Uma Word Ladder é uma sequência de palavras em que duas palavras adjacentes diferem em uma letra. Por exemplo, podemos ir da palavra tudo para a palavra nada: **tudo -> todo -> nado -> nada**. O programa é responsável por encontrar o caminho entre as palavras.

As palavras são provenientes de ficheiros de texto, podemos escolher um passando-o como argumento quando executamos o programa na linha de comandos.

A estrutura de dados consiste numa Hash table que utiliza o algoritmo de hash **crc32**. É depois formado um grafo em que cada palavra da Hash table é um vértice. Existe um edge entre dois vértices se as palavras diferem em uma letra. O programa utiliza então um algoritmo (Breath-First Search) para encontrar o caminho mais curto entre duas palavras, uma inicial e uma final, formando então um word ladder.

Descrição das Funções:

HASH_TABLE_CREATE

Esta função cria a Hash Table, alocando memória para uma estrutura previamente definida (**hash_table_t**), inicializando os seus campos e alocando memória para um array de ponteiros **hash_table_node_t ,** que são utilizados como cabeças das listas ligadas que compõe a tabela Hash.

O tamanho da tabela é definido como 101 por ser um número primo que gera um bom desempenho com a função de hash **crc32** no que toca a colisões. O número de entradas e arestas é definido inicialmente com 0.

A função utiliza **malloc** para alocar memória para a tabela hash e para o array de cabeças. Se a reserva de espaço de memória não for possível é devolvida uma mensagem de erro. O array de cabeças é percorrido modo a inicializar cada elemento com NULL. Finalmente a função devolve um ponteirode modo a inicializar cada elemento com NULL. Finalmente a função devolve um ponteiro para a tabela hash criada.

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

  hash_table = (hash_table_t
*)malloc(sizeof(hash_table_t));
  if(hash_table == NULL)
  {
    fprintf(stderr,"create_hash_table: out of memory\n");
    exit(1);
  }
  //
  // complete this
  //
  hash_table->hash_table_size = 101;
  hash_table->number_of_entries = 0;
  hash_table->number_of_edges = 0;
  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,"hash_table_create: unable to allocate
memory for the hash table heads\n");
    exit(1);
  }

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

  return hash_table;
}
```

Função HASH_TABLE_GROW

Esta função aumenta o tamanho da hash table por um fator de dois (duas vezes o tamanho original). Começa por salvaguardar o tamanho atual e o array de cabeças da tabela hash. De seguida, ela duplica o tamanho da tabela e aloca a memória necessária para o novo array de cabeças. Se não conseguir alocar a memória restaura o array antigo e devolve o tamanho.

Depois inicializa todos os elementos do novo array de cabeças como NULL e ,de seguida, ele percorre o antigo array de cabeças fazendo o re-hash de cada nó para nova tabela. Isto é feito guardando o próximo nó da lista ligada e depois usando a função **crc32** para calcular o novo indíce para inserir o novo nó na tabela, utilizando o ponteiro next e adicionando o nó ao novo array de cabeças no respetivo índice calculado.

Por fim a função liberta a memória utilizada pelo antigo array de cabeças.

```
static void hash_table_grow(hash_table_t *hash_table)
{
  // Create a new hash table with double the size of the
old one
  // Save the old size and heads array
    unsigned int old_size = hash_table->hash_table_size;
    hash_table_node_t **old_heads = hash_table->heads;
    // Double the size of the hash table
    hash_table->hash_table_size = old_size * 2;
    hash_table->heads = malloc(sizeof(hash_table_node_t*) *
hash_table->hash_table_size);
    if (hash_table->heads == NULL) {
        // Error allocating memory for the new heads array
        // Restore the old heads array and size
        hash_table->hash_table_size = old_size;
        hash_table->heads = old_heads;
        return;
    }
    // Initialize all elements in the new heads arraytoNULL
    for (int i = 0; i < hash_table->hash_table_size; i++) {
        hash_table->heads[i] = NULL;
    }
    // Rehash all the entries from the old table into the
new one
    for (int i = 0; i < old_size; i++) {
        hash_table_node_t *node = old_heads[i];
        while (node != NULL) {
            // Save the next node in the linked list
            hash_table_node_t *next = node->next;
            // Rehash the current node
            unsigned int index = crc32(node->word) %
hash_table->hash_table_size;
            node->next = hash_table->heads[index];
            hash_table->heads[index] = node;
            // Move on to the next node
            node = next;
        }
    }
    free(old_heads); // Free memory used by old heads array
}
```

Função HASH_TABLE_FREE

Esta função liberta a memória utilizada pela hash table. O array de cabeças da tabela é percorrido e cada nó da lista ligada é eliminado. Para cada cabeça a função guarda o próximo nó da lista e depois liberta a memória usada pela lista de adjacência do nó atual percorrendo-a e libertando cada adjacency_node_t.

De seguida a função liberta a memória utilizada pelo nó atual e segue para o próximo e finalmente liberta a memória utilizada pelo array de cabeças e a própria tabela hash.

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

  // Free the linked list nodes of each element in the
heads array
    for (int i = 0; i < hash_table->hash_table_size; i++) {
        hash_table_node_t *node = hash_table->heads[i];
        while (node != NULL) {
            // Save the next node in the linked list
            hash_table_node_t *next = node->next;

            // Free the adjacency list of the current node
            adjacency_node_t *adj_node = node->head;
            while (adj_node != NULL) {
                adjacency_node_t *adj_next = adj_node-
>next;
                free(adj_node);
                adj_node = adj_next;
            }

            // Free the current node
            free(node);

            // Move on to the next node
            node = next;
        }
    }

    // Free the memory used by the heads array
    free(hash_table->heads);

    // Free the memory used by the hash table
    free(hash_table);
}
```

Função FIND_WORD

Esta função tem a função de encontrar uma palavra na hash_table e de a inserir se ela não existir. O índice da palavra dada é calculado usado a função de hash **crc32** que é passado como índice do nó no array de cabeças dentro da tabela hash.

Se a palavra está na hash_table a função devolve o nó correspondente, se a palavra não estiver é inserida na hash_table.

Para fazer a inserção da palavra é necessário utilizar a função **Hash_Table_Grow** e ,como temos uma nova tabela, temos de recalcular o índice usando a mesma função de hash. Com o novo índice inserimos o nó da palavra na tabela com o tamanho novo. Este processo exige uma nova reserva de memória através do **malloc**.

Nota: Apesar de este ser o processo correto, a nossa implementação descarta o uso da função **Hash_Table_Grow** uma vez que estava a provocar um erro segmentation fault no nosso programa. Deixamos no código, em comentário, a condição de crescimento da tabela e respetiva chamada da função de crescimento.

```c
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)
    {
      // word is already in the hash table, return the node
      return node;
    }
    node = node->next;
  }

  // word is not in the hash table
  if (insert_if_not_found) // colocar condiçao para grow,
calcular novamente i
  {
    //if (hash_table->number_of_entries > hash_table-
>hash_table_size*0.7)
      //hash_table_grow(hash_table);

    // create a new node and insert it in the hash table
    node = (hash_table_node_t
*)malloc(sizeof(hash_table_node_t));
    if(node == NULL)
      return NULL;
    //i = crc32(word) % hash_table->hash_table_size; //
recalcular i
    node->next = hash_table->heads[i];
    hash_table->heads[i] = node;
    hash_table->number_of_entries++;
    strcpy(node->word, word);
    node->head = NULL;
    node->visited = 0;
    node->previous = NULL;
```

Função FIND_REPRESENTATIVE

Esta função faz parte da implementação do grafo que o programa tem de construir. Serve para encontrar um identificador comum a um conjunto de nos e é uma otimização para o algoritmo union-find. Este algoritmo é utilizado para encontrar componentes ligados no grafo.

```
static hash_table_node_t
*find_representative(hash_table_node_t *node)
{
    hash_table_node_t *representative,*next_node;
    representative = node;
    while( representative->representative != representative
){
        representative = representative->representative;
    }
    next_node = node;
    while( next_node->representative != next_node ){
        next_node = next_node->representative;
        node->representative = representative;
    }
    return representative;
}
```

Função ADD_EDGE

Esta função é responsável por adicionar um edge ao grafo representado pela hash table.

Primeiro é chamada a função **find_word** para encontrar o nó na hash table correspondente ao vértice do edge. Se ele não é encontrado temos de adicionar o edge.

Alocamos memória para os dois nós de adjacência e ligação para o respetivo vértice, vértice from e vértice to. Isto é necessário porque a ligação é bidirecional. O número de edges é incrementado e a função **find_representative** é chamada.

Se o representative não for igual significa que os vértices não estão no mesmo componente ligado e portanto atualiza o union-find data ao juntar os componentes ligados dos dois vértices. O número de vértices e de edges são atualizados.

```
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) {
    // Allocate a new adjacency node and link it to the
'from' vertex
    link = allocate_adjacency_node();
    if(link == NULL)
    return;
    link->next = from->head;
    link->vertex = to;
    from->head = link;

    // CORREÇAO - LINK TAMBEM OCORRE NO SENTIDO INVERSO
    link = allocate_adjacency_node();
    if(link == NULL)
    return;
    link->next = to->head;
    link->vertex = from;
    to->head = link;



    // update edge count
    hash_table->number_of_edges++;

    // update union-find data
    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;
```

Função BREADTH_FIRST_SEARCH

Esta função é responsável por implementar o algoritmo de pesquisa do programa, neste caso um Breadth_First que opera sobre o grafo representado pela tabela hash.

Como condições de pesquisa temos a palavra origem (origin) e a palavra destino (goal). Inicialmente todos os vértices são marcados como não visitados e o nóde origem é adicionado à lista de pesquisa e marcado como visitado.

A função vai então iterar sobre o grafo percorrendo todos os vértices e edges até encontrar a palavra destino. Quando o destino é alcançado a distância entre as palavras é devolvida.

Se o destino não é encontrado a função devolve -1. O objetivo deste algoritmo é encontrar o caminho mais curto entre duas palavras, formando assim uma word ladder.

```
static int breadth_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 i, distance = 0, current_number_of_vertices = 1;
    hash_table_node_t *temp;

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

    list_of_vertices[0] = origin;
    origin->visited = 1;

    while (current_number_of_vertices > 0) {
        int next_number_of_vertices = 0;
        distance++;
        for (i = 0; i < current_number_of_vertices; i++) {
            adjacency_node_t *link = list_of_vertices[i]-
>head;
            while (link != NULL) {
                temp = link->vertex;
                if (!temp->visited) {
                    if (temp == goal) {
                        return distance;
                    }
                    list_of_vertices[next_number_of_vertice
s++] = temp;
                    temp->visited = 1;
                    temp->previous = list_of_vertices[i];
                }
                link = link->next;
            }
        }
        current_number_of_vertices =
next_number_of_vertices;
    }
    return -1; // -1
```

Função LIST_CONNECTED_COMPONENT

Esta função é usada para listar os vértices do componente ligado do vértice de uma dada palavra. A função **find_word** é chamada para encontrar o vértice na hash table que corresponde à palavra dada. Se não for encontrado é devolvida uma mensagem de erro.

Se o vértice existe a função **find_representative** é chamada para encontrar o representante do componente ligado do vértice.

Ocorre uma iteração sobre a hash table e , para cada vértice, é verificado se o representante do componente ligado é igual ao do vértice, se for igual imprime a palavra desse vértice.

São impressos todos os vértices que se encontram ligados ao vértice da palavra dada. Assim é possível verificar o algoritmo de union-find. Esta lista de palavras nada mais é do que todas as palavras às quais existe um caminho com a palavra inicial.

```
static void list_connected_component(hash_table_t
*hash_table,const char *word)
{
    hash_table_node_t *vertex = find_word(hash_table, word,
0);
    if(vertex == NULL) {
        printf("The word '%s' was not found in the hash
table.\n", word);
        return;
    }

    hash_table_node_t *representative =
find_representative(vertex);
    printf("Vertices in the connected component: \n");
    for(int i = 0; i < hash_table->hash_table_size; i++) {
        hash_table_node_t *temp = hash_table->heads[i];
        while(temp != NULL) {
            if(find_representative(temp) == representative)
{
                printf("%s\n", temp->word);
            }
            temp = temp->next;
        }
    }
}
```

Função PATH_FINDER

Esta função é responsável por descobrir o caminho mais curto entre duas palavras. A cada passo do caminho é apenas alterada uma letra. É o caminho mais curto entre dois vértices do grafo representado pela hash table.

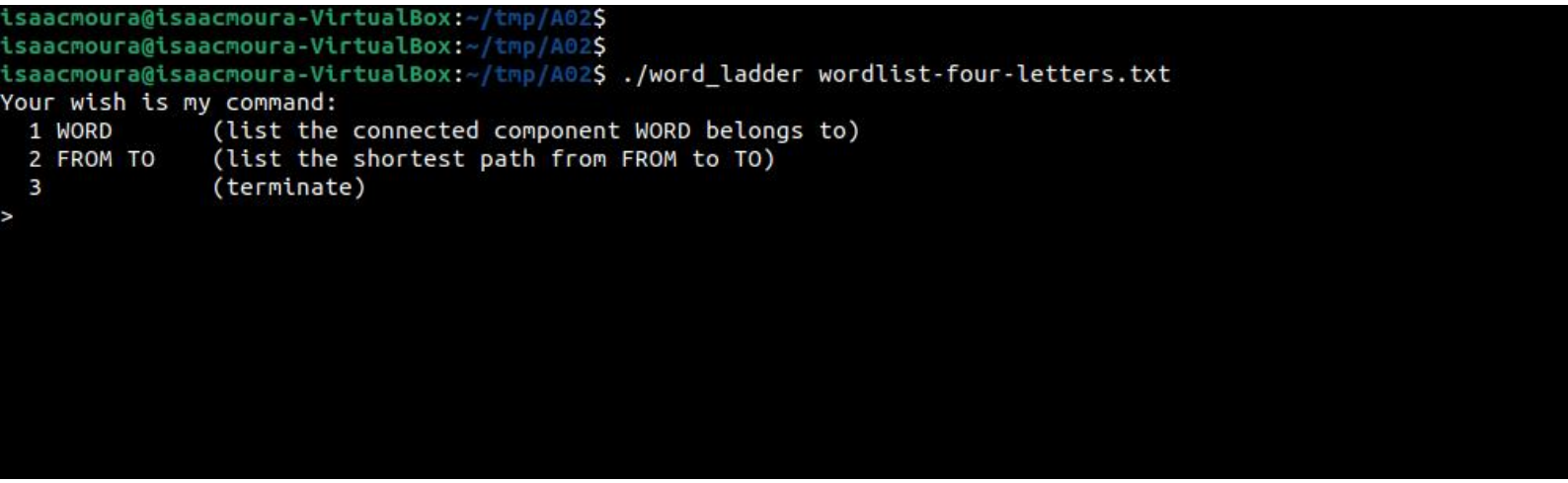
A função **find_word** é chamada para verificar que as duas palavras estão na hash table. Se algum dos vértices não existir é devolvida uma mensagem de erro.

Depois é chamada a **função breadth_first_search** que vai encontrar o caminho mais curto. Se a distancia devolvida por esta função for -1 significa que nenhum caminho foi encontrado. Caso contrário é devolvido o caminho com a distância encontrada. A função path_finder junta todas as funções necessárias para gerar um word ladder.

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

    if(from == NULL) {
        printf("The word '%s' was not found in the hash
table.\n", from_word);
        return;
    }
    if(to == NULL) {
        printf("The word '%s' was not found in the hash
table.\n", to_word);
        return;
    }

    int distance = breadth_first_search(hash_table-
>hash_table_size, hash_table->heads, from, to);
    if(distance == -1) {
        printf("No path was found between the words '%s'
and '%s'\n", from_word, to_word);
        return;
    }

    hash_table_node_t *temp = to;
    printf("Shortest path between '%s' and '%s' with
distance %d: \n", from_word, to_word, distance);
    while(temp != NULL) {
        printf("%s -> ", temp->word);
        temp = temp->previous;
    }
    printf("\n");
}
```

Funcionamento do Programa

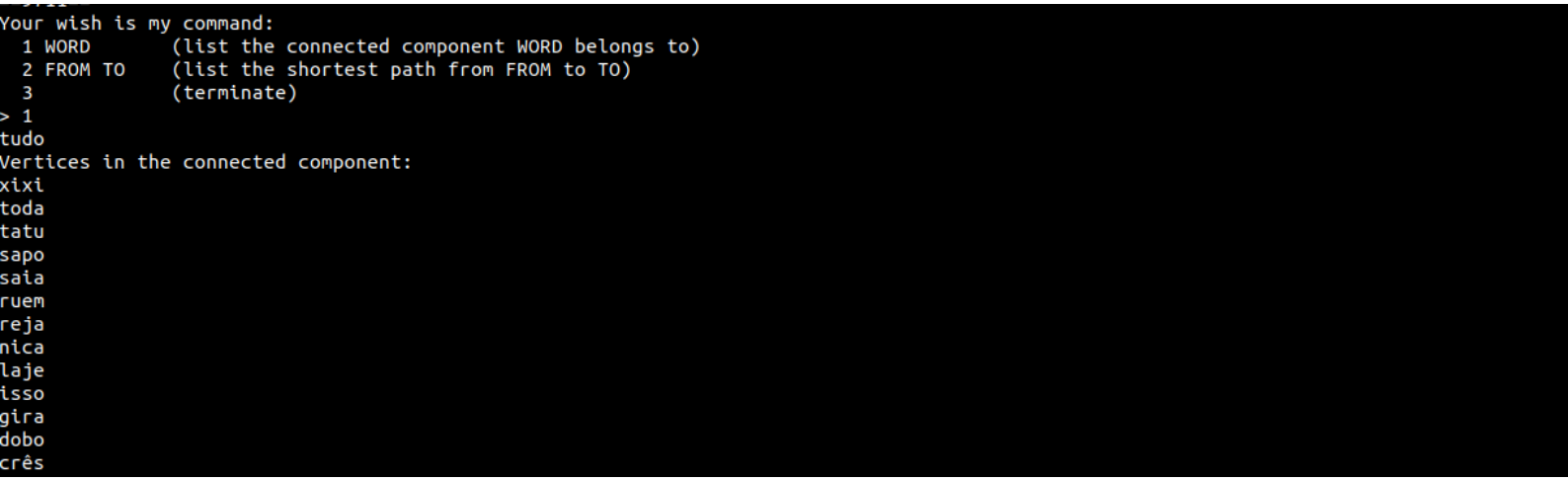
O programa tem um menu que permite ao utilizador realizar 3 operações:



- Ao digitar (1):

Utilizador tem de inserir uma palavra. O programa vai então listar todas as palavras do componente ligado da palavra dada pelo utilizador.

Exemplo:



- Ao digitar (2):

Utilizador tem de inserir a palavra de partida e a palavra destino para a formação da word ladder.

Exemplo:

```
Shortest path between 'tudo' and 'nada' with distance 4:
nada ->
Your wish is my command:
  1 WORD       (list the connected component WORD belongs to)
  2 FROM TO    (list the shortest path from FROM to TO)
  3            (terminate)
>
```

- Ao digitar (3) o programa termina.

Teste de Memory Leaks

Para o teste de memory leaks foi usado o **Valgrind.** Ao longo da testagem do nosso programa encontramos vários comportamentos estranhos que culminavam e **segmentation fault** e **core dumped.** Não conseguimos resolver estes problemas em tempo útil pelo que os demonstramos a seguir.

Alguns problemas:

- Erro “Core Dumped” ao sair do programa:

```
Your wish is my command:
  1 WORD       (list the connected component WORD belongs to)
  2 FROM TO    (list the shortest path from FROM to TO)
  3            (terminate)
> 2
meus teus
Shortest path between 'meus' and 'teus' with distance 1:
teus ->
Your wish is my command:
  1 WORD       (list the connected component WORD belongs to)
  2 FROM TO    (list the shortest path from FROM to TO)
  3            (terminate)
> 3
free(): double free detected in tcache 2
Aborted (core dumped)
```

- Erro “Segmentation Fault” ao realizar a operação (2):

```
isaacmoura@isaacmoura-VirtualBox:~/tmp/A02$ ./word_ladder wordlist-four-letters.txt
Your wish is my command:
  1 WORD       (list the connected component WORD belongs to)
  2 FROM TO    (list the shortest path from FROM to TO)
  3            (terminate)
> 2
tudo nada
Segmentation fault (core dumped)
isaacmoura@isaacmoura-VirtualBox:~/tmp/A02$
isaacmoura@isaacmoura-VirtualBox:~/tmp/A02$
isaacmoura@isaacmoura-VirtualBox:~/tmp/A02$
isaacmoura@isaacmoura-VirtualBox:~/tmp/A02$
```

Uso do Valgrind:

- Teste para memory leaks na primeira operação do programa:

```
Your wish is my command:
  1 WORD       (list the connected component WORD belongs to)
  2 FROM TO    (list the shortest path from FROM to TO)
  3            (terminate)
> 3
==9689==
==9689== HEAP SUMMARY:
==9689==     in use at exit: 0 bytes in 0 blocks
==9689==   total heap usage: 20,689 allocs, 20,689 frees, 475,912 bytes allocate
d
==9689==
==9689== All heap blocks were freed -- no leaks are possible
==9689==
==9689== For lists of detected and suppressed errors, rerun with: -s
==9689== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
isaacmoura@isaacmoura-VirtualBox:~/tmp/A02$
```

- Teste de memory leaks na segunda operação do programa:

```
==9680==
==9680==
==9680== HEAP SUMMARY:
==9680==     in use at exit: 278,800 bytes in 12,393 blocks
==9680==   total heap usage: 20,689 allocs, 11,423 frees, 475,912 bytes allocate
d
==9680==
==9680== 278,800 (1,520 direct, 277,280 indirect) bytes in 19 blocks are definit
ely lost in loss record 4 of 4
==9680==    at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-am
d64-linux.so)
==9680==    by 0x1098E9: find_word (in /home/isaacmoura/tmp/A02/word_ladder.out)
==9680==    by 0x10A339: main (in /home/isaacmoura/tmp/A02/word_ladder.out)
==9680==
==9680== LEAK SUMMARY:
==9680==    definitely lost: 1,520 bytes in 19 blocks
==9680==    indirectly lost: 277,280 bytes in 12,374 blocks
==9680==      possibly lost: 0 bytes in 0 blocks
==9680==    still reachable: 0 bytes in 0 blocks
==9680==         suppressed: 0 bytes in 0 blocks
==9680==
==9680== For lists of detected and suppressed errors, rerun with: -s
==9680== ERROR SUMMARY: 6938 errors from 9 contexts (suppressed: 0 from 0)
isaacmoura@isaacmoura-VirtualBox:~/tmp/A02$
```

Foram encontradas memory leaks na segunda operação do programa. Estas leaks comprometem o normal funcionamento do programa o que o leva por vezes a não produzir o resultado esperado.

Anexo com código completo do programa

```
//
// AED, November 2022 (Tomás Oliveira e Silva)
//
// Second practical assignement (word ladder)
//
// Place your student numbers and names here
//   105065 Isaac Moura
//   98651  André Silva
//
// Do as much as you can
//   1) MANDATORY: complete the hash table code
//      *) hash_table_create
//      *) hash_table_grow
//      *) hash_table_free
//      *) find_word
//      +) add code to get some statistical data about the hash
table
//   2) HIGHLY RECOMMENDED: build the graph (including union-
find data) -- use the similar_words function...
//      *) find_representative
//      *) add_edge
//   3) RECOMMENDED: implement breadth-first search in the
graph
//      *) breadh_first_search
//   4) RECOMMENDED: list all words belonginh to a connected
component
//      *) breadh_first_search
//      *) list_connected_component
//   5) RECOMMENDED: find the shortest path between to words
//      *) breadh_first_search
//      *) path_finder
```

```
//      *) test the smallest path from bem to mal
//         [ 0] bem
//         [ 1] tem
//         [ 2] teu
//         [ 3] meu
//         [ 4] mau
//         [ 5] mal
//      *) find other interesting word ladders
//   6) OPTIONAL: compute the diameter of a connected component
and list the longest word chain
//      *) breadh_first_search
//      *) connected_component_diameter
//   7) OPTIONAL: print some statistics about the graph
//      *) graph_info
//   8) OPTIONAL: test for memory leaks
//

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


//
// static configuration
//

#define _max_word_size_  32


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

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

struct adjacency_node_s
{
```

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

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

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

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

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

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

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

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

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

static void free_hash_table_node(hash_table_node_t *node)
```

```
{
  free(node);
}


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

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

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

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

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);
  }
  //
  // complete this
  //
  hash_table->hash_table_size = 101;
  hash_table->number_of_entries = 0;
  hash_table->number_of_edges = 0;
  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,"hash_table_create: unable to allocate
memory for the hash table heads\n");
    exit(1);
  }

  for(int 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)
{
  //
  // complete this
  //
  // Create a new hash table with double the size of the old
one

  // Save the old size and heads array
    unsigned int old_size = hash_table->hash_table_size;
    hash_table_node_t **old_heads = hash_table->heads;
```

```
    // Double the size of the hash table
    hash_table->hash_table_size = old_size * 2;
    hash_table->heads = malloc(sizeof(hash_table_node_t*) *
hash_table->hash_table_size);
    if (hash_table->heads == NULL) {
        // Error allocating memory for the new heads array
        // Restore the old heads array and size
        hash_table->hash_table_size = old_size;
        hash_table->heads = old_heads;
        return;
    }

    // Initialize all elements in the new heads array to NULL
    for (int i = 0; i < hash_table->hash_table_size; i++) {
        hash_table->heads[i] = NULL;
    }

    // Rehash all the entries from the old table into the new
one
    for (int i = 0; i < old_size; i++) {
        hash_table_node_t *node = old_heads[i];
        while (node != NULL) {
            // Save the next node in the linked list
            hash_table_node_t *next = node->next;

            // Rehash the current node
            unsigned int index = crc32(node->word) %
hash_table->hash_table_size;
            node->next = hash_table->heads[index];
            hash_table->heads[index] = node;

            // Move on to the next node
            node = next;
        }
    }

    // Free the memory used by the old heads array
    free(old_heads);
}
```

```c
static void hash_table_free(hash_table_t *hash_table)
{
  //
  // complete this
  //
  // Free the linked list nodes of each element in the heads
array
    for (int i = 0; i < hash_table->hash_table_size; i++) {
        hash_table_node_t *node = hash_table->heads[i];
        while (node != NULL) {
            // Save the next node in the linked list
            hash_table_node_t *next = node->next;

            // Free the adjacency list of the current node
            adjacency_node_t *adj_node = node->head;
            while (adj_node != NULL) {
                adjacency_node_t *adj_next = adj_node->next;
                free(adj_node);
                adj_node = adj_next;
            }

            // Free the current node
            free(node);

            // Move on to the next node
            node = next;
        }
    }

    // Free the memory used by the heads array
    free(hash_table->heads);

    // Free the memory used by the hash table
    free(hash_table);
}
```

```c
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)
    {
      // word is already in the hash table, return the node
      return node;
    }
    node = node->next;
  }

  // word is not in the hash table
  if (insert_if_not_found) // colocar condiçao para grow,
calcular novamente i
  {
    //if (hash_table->number_of_entries > hash_table-
>hash_table_size*0.7)
      //hash_table_grow(hash_table);


    //hash_table_node_t* node;

    // create a new node and insert it in the hash table
    node = (hash_table_node_t
*)malloc(sizeof(hash_table_node_t));
    if(node == NULL)
      return NULL;
    //i = crc32(word) % hash_table->hash_table_size;
    node->next = hash_table->heads[i];
    hash_table->heads[i] = node;
    hash_table->number_of_entries++;
    strcpy(node->word, word);
    node->head = NULL;
```

```
    node->visited = 0;
    node->previous = NULL;
    node->representative = node;
    node->number_of_vertices = 1;
    node->number_of_edges = 0;

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

```
  if(to != NULL) {
    // Allocate a new adjacency node and link it to the 'from'
vertex
    link = allocate_adjacency_node();
    if(link == NULL)
    return;
    link->next = from->head;
    link->vertex = to;
    from->head = link;

    // CORREÇAO - LINK TAMBEM OCORRE NO SENTIDO INVERSO
    link = allocate_adjacency_node();
    if(link == NULL)
    return;
    link->next = to->head;
    link->vertex = from;
    to->head = link;



    // update edge count
    hash_table->number_of_edges++;

    // update union-find data
    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_vertices +=
from_representative->number_of_vertices;
        to_representative->number_of_edges +=
from_representative->number_of_edges;
      } else {
        to_representative->representative =
from_representative;
        from_representative->number_of_vertices +=
to_representative->number_of_vertices;
```

```
        from_representative->number_of_edges +=
to_representative->number_of_edges+1; // CORREÇAO - NECESSÁRIO
ADICIONAR 1 AO NUMBER_OF_EDGES
      }
    }
  }
}


//
// 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
//
```

```c
static int breadth_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 i, distance = 0, current_number_of_vertices = 1;
    hash_table_node_t *temp;

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

    list_of_vertices[0] = origin;
    origin->visited = 1;

    while (current_number_of_vertices > 0) {
        int next_number_of_vertices = 0;
        distance++;
        for (i = 0; i < current_number_of_vertices; i++) {
            adjacency_node_t *link = list_of_vertices[i]->head;
            while (link != NULL) {
                temp = link->vertex;
                if (!temp->visited) {
                    if (temp == goal) {
                        return distance;
                    }
                    list_of_vertices[next_number_of_vertices++]
= temp;
                    temp->visited = 1;
                    temp->previous = list_of_vertices[i];
                }
                link = link->next;
            }
        }
        current_number_of_vertices = next_number_of_vertices;
    }
    return -1; // -1
}

//
```

```c
// 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 *vertex = find_word(hash_table, word, 0);
    if(vertex == NULL) {
        printf("The word '%s' was not found in the hash
table.\n", word);
        return;
    }

    hash_table_node_t *representative =
find_representative(vertex);
    printf("Vertices in the connected component: \n");
    for(int i = 0; i < hash_table->hash_table_size; i++) {
        hash_table_node_t *temp = hash_table->heads[i];
        while(temp != NULL) {
            if(find_representative(temp) == representative) {
                printf("%s\n", temp->word);
            }
            temp = temp->next;
        }
    }
}

//
// 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 = find_word(hash_table, from_word,
0);
    hash_table_node_t *to = find_word(hash_table, to_word, 0);

    if(from == NULL) {
        printf("The word '%s' was not found in the hash
table.\n", from_word);
        return;
    }
    if(to == NULL) {
        printf("The word '%s' was not found in the hash
table.\n", to_word);
        return;
    }

    int distance = breadth_first_search(hash_table-
>hash_table_size, hash_table->heads, from, to);
    if(distance == -1) {
        printf("No path was found between the words '%s' and
'%s'\n", from_word, to_word);
        return;
    }

    hash_table_node_t *temp = to;
```

```
    printf("Shortest path between '%s' and '%s' with distance
%d: \n", from_word, to_word, distance);
    while(temp != NULL) {
        printf("%s -> ", temp->word);
        temp = temp->previous;
    }
    printf("\n");
}


//
// 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;
}
```