Word Ladder

Licenciatura em Engenharia Informática

Algoritmos e Estrutura de Dados

Docentes:

Professor Tomás Oliveira e Silva

Professor João Manuel Rodrigues

Alunos:

Bárbara Nóbrega Galiza – 105937 (50%)

Tomás António de Oliveira Victal - 109018 (50%)

Dezembro 2022

Conteúdo

1 Introdução

Esse trabalho tem como objetivo a criação de uma estrutura de dados do tipo hash table para guardar palavras contidas num ficheiro ".txt". Essa mesma estrutura serve como base para um grafo que contém as palavras lidas do ficheiro. Este grafo será usado para implementar uma "word ladder".

Segundo o website Weaver (Weaver,2022), word ladder é um jogo de palavras que foi inventado em 1877 por Lewis Carroll. Este jogo inicia com a escolha de duas palavras com o mesmo tamanho, e seu objetivo é conseguir encontrar uma sequência de palavras adjacentes de modo a conectar as duas palavras iniciais. Duas palavras são adjacentes quando diferem em apenas uma letra, por exemplo: se as palavras iniciais forem **cara** e **luva** uma sequência possível para resolver o puzzle seria **cara** $\rightarrow$ **cura** $\rightarrow$ **lura** $\rightarrow$ **luva**.

No grafo, cada palavra é um vértice, e palavras adjacentes são conectadas formando assim cada aresta do grafo. Com o grafo criado, fica fácil encontrar a sequência entre quaisquer duas palavras, bastando usar um método de procura pelo grafo. Nesta implementação, o método usado na procura é o breadth first search.

2 Hash Table

2.1 Estrutura

A informação relevante da **Hash Table** está guardada na estrutura **hash_table_s**, e na estrutura **hash_table_node_s**, que guarda a informação de cada node contido na tabela.

A estrutura **hash_table_s** contém quatro campos de informação referentes à hash table, sendo estes:

-hash_table_size: Tamanho da tabela.

-number_of_entries: Número de entradas na tabela.

-nodes_per_head: Array relativo à quantidade de nodes em cada hash code (variável extra adicionada por nós para fins estatísticos).

-heads: Array onde são guardados os nodes da tabela.

A estrutura **hash_table_node_s** contém dois campos de informação referentes à hash table:

-word: Array de caracteres para guardar a palavra.

-next: Próximo node da mesma linked list da tabela.

2.2 Funções

2.2.1 hash_table_create

Esta função tem como objetivo inicializar uma hash table, e não requer qualquer argumento.

Quando é executada, começa por alocar espaço para uma hash_table. Inicialmente seu tamanho é 100, e sabendo o tamanho da tabela podemos criar os arrays heads e nodes_per_head com esse tamanho. Para todas a funções funcionarem corretamente, igualamos todas as entradas de heads a NULL.

Por fim, retorna a hash table.

2.2.2 hash_table_grow

Como o próprio nome indica, esta função aumenta o tamanho da hash table, sendo necessário receber como argumento a hash table que desejamos aumentar. Nela, dobramos o tamanho da tabela, e com o novo size alocamos memória para um new_heads e um new_heads_per_node. De seguida, inicializamos todos as

new_head com NULL e percorremos todos os nodes da tabela antiga, gerando um novo hash code para cada um e colocando cada um deles no new_heads. Além disso, passamos os valores de nodes_per_head para os novos índices correspondentes em new_nodes_per_head. Por fim, libertamos a memória de heads e de nodes_per_head originais e colocamos new_heads e new_heads_per_node na tabela.

2.2.3 find_word

A função find_word() desempenha duas funções:

1. Adicionar um novo nó à hash table
2. Devolver o ponteiro para um nó existente

O comportamento da função é especificado pelo argumento insert_if_not_found: Se após não achar um node correspondente à palavra enviada, esse argumento valer 1, inserimos um novo node. Se este valer 0, não inserimos.

Para cumprir seu objetivo, a função recebe como argumento um ponteiro para a hash table, a palavra a procurar/adicionar e o argumento descrito no parágrafo anterior. Nela, obtemos o hashcode relativo à palavra e salvamos na variável node o valor presente na posição indexada por esse hashcode no array heads da hash table. Em seguida, percorremos num ciclo todos os nós da lista ligada iniciada por node, e comparamos as palavras de cada um com a palavra que procuramos. Caso seja encontrada, damos return do node correspondente. Caso não seja encontrada e o próximo node seja NULL, salvamos em uma variável o valor de node, que irá ser utilizado na inserção (caso ela seja pedida). Além disso, em cada iteração incrementamos a variável size_linked_list que será usada para fins estatísticos posteriormente.

Em seguida, caso insert_if_not_found tenha valor 1, entramos num if no qual alocamos um novo node, inicializamos todos os seus campos e copiamos a palavra recebida para o campo word do node. Depois, adicionamos esse novo node à hash table, logo na head[i] se essa valer NULL, ou após o último nó presente se já houver nodes mapeados para essa mesma posição. Por fim, atualizamos as variáveis de propósitos estatísticos e, caso o número de entradas ultrapasse 70%, fazemos o redimensionamento da hash table, através da chamada à função hash_table_grow().

2.2.4 hash_table_free

A função hash_table_free tem o intuito de libertar a memória que foi alocada para construir a hash table. Para esse efeito, percorremos tudo que foi alocado,

libertando primeiro aquilo que é apontado por último, que é o caso dos adjacency_nodes. Estes ainda não foram mencionados, mas serão usados para a construção do grafo e são listas ligadas presentes em cada hash_table_node. Portanto, percorremos todos os hash node, para cada head[i], e para cada hash node, percorremos seus ajacency nodes.

Para remover, usamos sempre uma variável previous para guardar a posição que iremos libertar e assim poder atualizar o valor do adjacency node atual com o valor do próximo adjacency node. Seguindo esse mesmo processo, também libertamos todos os hash nodes alocados. Por fim, libertamos o array nodes_per_head, o array heads e a memória alocada para a hash_table.

2.2.5 print_hash_table_statistics

A função print_hash_table_statistics não estava definida no código que nos foi dado, mas optamos por introduzi-la a fim de obter algumas estatísticas acerca da hash table. Nela, imprimimos seu número de entradas, seu tamanho, o número de colisões, o número de redimensionamentos, o tamanho médio da lista ligada e o tempo gasto para a adição de todas as palavras na hash table. A informação relacionada às colisões foi obtida durante a execução da função find_word(), quando verificamos que a posição head[i] não estava NULL.

Além disso, inserimos um ciclo for para percorrer cada hashcode e imprimir o número de nós mapeados para ele, a partir da variável nodes_per_head. Isso foi feito com o propósito de fazer um histograma para analisar a dispersão gerada pela hash function, resultado que será discutido em uma secção adiante. Após obtermos os dados, apagamos esta secção do código, visto que não possuía utilidade para a solução em si.

2.3 Estatísticas

Como dito anteriormente na secção 2.2.5, desenvolvemos a função print_hash_table_statistics a fim de coletar e analisar alguns dados referentes à hash table. Nesta secção, serão apresentados os resultados obtidos.

Ao correr o programa com o ficheiro ”wordlist-big-latest.txt”, obtivemos a seguintes informações:

number of entries: 999282

size: 1638400

number of resizes: 14

number of collisions: 400183

average linked list size: 0.609913

elapsed time dispended on adding all nodes to hash table: 0.9237

A partir desses dados, concluímos que a hash function gerou um elevado número de colisões. Além disso, concluímos que o redimensionamento da hash table está a funcionar bem: foram feitos 14 redimensionamentos (esse número foi incrementado em cada chamada da função hash_table_grow), e em cada um dobrou-se o tamanho da hash table, o que deveria resultar, e de facto resultou, em um tamanho final de $2^{14} * n$, sendo n = 100 o tamanho inicial da hash table.

Ainda através desses números, certificamos que nossa solução adicionou todas as palavras contidas no ficheiro, ao comparar o number of entries ao número de linhas do mesmo, e que o tempo necessário para fazê-lo foi curto, menor que 1 segundo (em outras execuções foi ainda menor).

Em adição, para acrescentar ao que já concluímos, imprimimos o número de nós presentes em cada índice da hash table (array nodes_per_head), a fim de verificar se a hash function produzia hashcodes bem distribuídos e de analisar a quantidade de posições da hashtable que possuiam *x* nodes. Para isso, utilizamos um script MATLAB e geramos os dois histogramas a seguir:

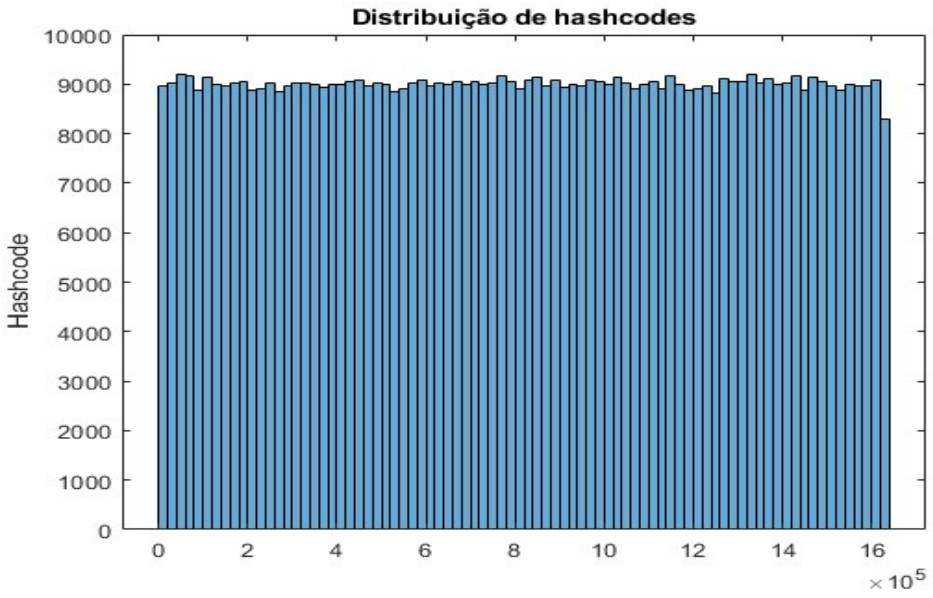

Figura 1: Distribuição de hashcodes ao longo da hash table.

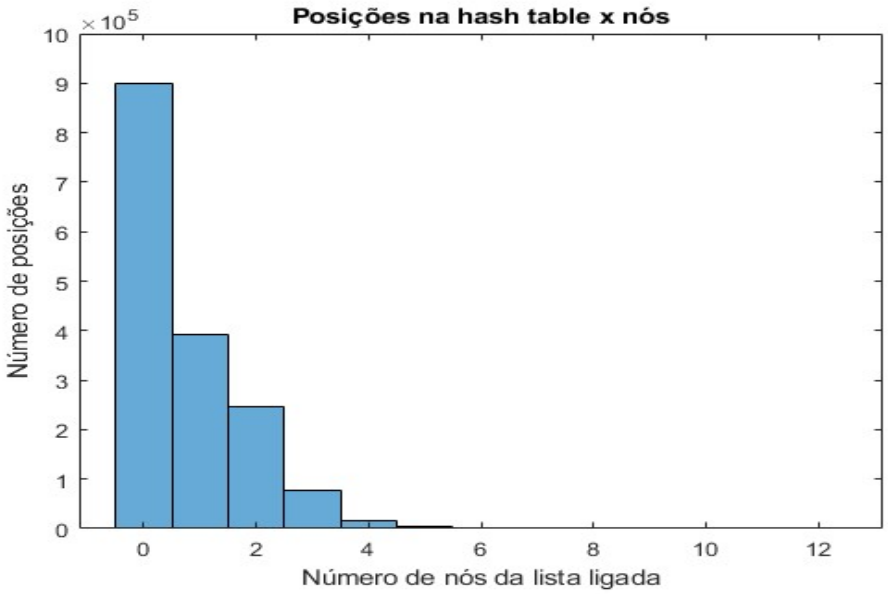

Figura 2: Quantidade de posições da hash table com *x* número de nós em sua lista ligada.

Com o histograma da figura 1, concluímos que a hash function faz uma boa distribuição dos hashcodes. Já com a figura 2, notamos que a distribuição desses hashcodes pela hash table não é tão eficiente: muitas posições possuem 0 nós mapeados. Além disso, apesar de não aparcer na figura, ao utilizarmos a instrução max() no MATLAB constatamos que a maior lista ligada possui 12 nós, o que é um número relativamente grande, apesar de acontecer poucas vezes (nesse caso, foram apenas 2).

3 Grafo

3.1 Estrutura

O grafo utilizado na resolução do problema tem a sua informação guardada nas estruturas da hash table (hash_table_node_s e hash_table_s) e na estrutura adjacency_node_s. Para além disso, também é usada uma queue para auxiliar no processo de procura no grafo.

A estrutura **hash_table_s** contém um campo referente ao grafo:

-number_of_edges Número de arestas do grafo.

A estrutura **hash_table_node_s** contém seis campos referentes ao grafo, sendo estes:

-head: Uma linked list de nodes adjacentes.

-visited: Um inteiro que serve como booleano para auxiliar durante a pesquisa do grafo.

-previous: Usado na pesquisa do grafo para indicar o node anterior.

-representative: Representante da componente conexa a qual o node pertence.

-number_of_vertice: Número de vértices da componente conexa (este valor só está correto no representante).

-number_of_edges: Número de arestas da componente conexa (este valor só está correto no representante).

A estrutura **adjacency_node_s** serve para criar a linked list em cada head e tem dois campos:

-next: Próximo node da lista ligada.

-vertex: Ponteiro para um vértice (node).

3.2 Funções

3.2.1 find_representative

Esta função tem duas finalidades. A primeira, como o próprio nome indica, é encontrar o representante, e a segunda é melhorar o acesso ao representante caso este não seja direto (path-compression).

Para encontrar o representante, percorreremos o representante de cada node até encontrar um node cujo representante aponte para si próprio, sendo esse o representante que procuramos. Caso o caminho ao representante do node não seja direto, ou seja, os representantes dos nós que percorremos não apontam logo para ele, então percorremos todos os nodes novamente, mas desta vez trocamos o representante de cada um pelo o representante obtido anteriormente.

3.2.2 add_edge

A função add_edge tem como intuito a adição de arestas entre vértices adjacentes, ou seja, cuja palavra só possui uma letra de diferença. Além disso, também efetua a fase de união do union-find.

A função é chamada por outra função, a similar_words, para cada par de palavras adjacentes. Em seu corpo, a variável to recebe o nó correspondente à palavra

enviada como argumento. Verificamos se esse nó existe na tabela, ou seja, se não tem valor NULL, e caso verdade, procedemos com a adição da aresta.

Para isso, alocamos um adjacency node e o adicionamos na lista de cada node, tanto para o node to (adjacency node aponta para o from) como para o node from (adjacency node aponta para o to). Depois, incrementamos o número total de arestas no grafo e prosseguimos para a fase de união do union-find.

Para fazer a união, guardamos os representantes de cada node em duas variáveis, from_representative e to_representative, a partir dos valores devolvidos pela função find_representative. Então, verificamos se estes são diferentes; caso verdade, fazemos com que o representante do representante do node cuja componente conexa possui menos vértices aponte para o representante do outro nó.

3.2.3 breadth_first_search

Esta função implementa o método de pesquisa breadth fisrt search no grafo. Para isso, requer dois argumentos, origin e goal, sendo ambos nodes da hash table. Para além disso, é utilizada uma variável global que guarda o endereço de uma queue (que será usada como linked list para guardar todos os nodes de cada componente conexa).

Primeiramente, inicializamos duas queues (queue e all_vertices), colocamos o node origin em cada queue e alteramos o valor visited do node para 1. Enquanto a queue tiver elementos vamos fazer **dequeue** (função referente a queue para remover elemento, retorna o elemento) de um elemento e vamos percorrer todos os nodes adjacentes desse mesmo elemento. Para todos os nodes que ainda não haviam sido visitados, colocamos o valor visited igual a 1 e o valor previous a apontar para o node que inicialmente demos **dequeue**. Por fim, fazemos **enqueue** (função referente a queue para inserir elemento) do node nas duas queues.

Caso o node devolvido pelo **dequeue** seja igual ao valor goal, paramos o ciclo while, pois não vale a pena continuar a pesquisa já que chegamos ao objetivo.

Por fim, retornamos o número de vértices visitados, que é igual ao tamanho da queue all_vertices e eliminamos a queue.

3.2.4 breadth_first_search_reset

Esta função tem como objetivo preparar o grafo para fazer outra pesquisa com o metodo breadth first search. Para executar essa função é necessário utilizar a hash table, que contém o grafo, como argumento da função. Para preparar o grafo para outra pesquisa basta percorrer todos os nodes da tabela e repor os valores originais das variáveis visited e previous com os valores 0 e NULL, respetivamente.

3.2.5 list_connected_component

A finalidade desta função é obter todas as palavras da componente conexa de uma certa palavra. Para esse efeito, a função recebe como argumento uma string (uma palavra) e a hash table que estamos a analisar.

A função começa por verificar se a palavra é valida, e caso não seja é impressa uma mensagem de erro. Após as verificações usamos breadh_first_search para percorrer todos os nodes da componente conexa, imprimimos o valor retornado da função e usamos a função print_queue_items(all_vertices) para imprimir todos os nodes colocados na queue all_vertices. Por fim, usamos breadth_first_search_reset e libertamos a memória de all_vertices.

3.2.6 path_finder

A função path_finder recebe como argumentos uma hash_table e duas strings: from_word e to_word. A função tem como objetivo encontrar o menor caminho possível entre as duas palavras.

Para iniciarmos o processo, encontramos os nodes que contém as palavras, e caso alguma das palavras não esteja no grafo o programa imprime uma mensagem de erro na consola e retorna. Após ter os dois nodes, testamos se ambos estão na mesma componente conexa. Caso não estejam, não existe caminho possível entre as duas palavras, e por isso imprimimos uma mensagem indicando que não há caminho e o programa retorna.

Se o programa ainda não retornou, sabemos que existe caminho entre as palavras, e por isso usamos o breadh_first_search para obter o caminho entre elas. Para obtermos as palavras que compõem o caminho basta fazer backtracking a partir do node goal, o que é feito através de um while loop no qual percorremos a variável previous de cada node formando assim a string com o caminho que vamos imprimir. Depois, usamos uma função auxiliar para colocar o caminho por ordem (da origem até ao objetivo), imprimimos a string, chamamos a função breadth_first_search_reset e limpamos a memória de all_vertices.

3.2.7 graph_info

A função graph_info tem o intuito de imprimir as estatísticas relacionadas ao grafo. Nela, imprimimos o número de arestas e o tempo gasto para adiciona-las e realizar as operações relacionadas ao union-find, através do ficheiro ”/P02/elapsed_time.h”. Além disso, nela criamos um ciclo para percorrer todos os nós e imprimir seu representante. Fizemos isso a fim de obter dados suficientes para gerar estatísticas como o número de componentes conexas, componente conexa com mais vértices, e outras, que serão evidenciadas na secção a seguir. Como

esse procedimento demora um certo tempo, optamos por definir uma macro para que essa região de código só seja incluída quando desejamos.

3.3 Estatísticas

Como mencionado, a partir da função graph_info, obtivemos dados suficientes para gerar estatísticas sobre o grafo. Para fazê-lo, construímos um código em MATLAB (ver apêndice) para obter da lista de representantes de cada palavra os representantes únicos, ou seja, o conjunto de representantes. Com isso, como cada componente conexa possui apenas 1 representante, concluímos que o número de componentes conexas do grafo é 377233.

Além disso, obtivemos as seguintes informações: a maior componente conexa possui 16698 vértices, e seu representante corresponde à palavra ”manamos”; o número de componentes conexas com apenas 1 vértice é 184869 (cerca de 49% das componentes conexas, o que é bastante); a maior palavra é ”constitucionalizar-lhes-íamos”, com 29 letras; o número de vértices, como é óbvio, tem o mesmo valor que a variável number_of_entries, que é 999282.

Diante disso, em decorrência do elevado número de vértices isolados, concluímos que o grafo é esparso, apesar de possuir componentes com muitos vértices.

Já em relação aos dados obtidos diretamente de graph_info, foi impresso o seguinte:

number of edges: 1060534

elapsed time dispended on adding edges and performing union-find

operations to the graph: 159.3454

Com isso, concluímos que a demora na execução do programa vem das operações relacionadas ao grafo, bem mais que as operações relacionadas à hash table. São cerca de 2 minutos e meio para adicionar todas as arestas e definir os representantes da componentes conexas, o que faz sentido, visto o grande número de arestas (1060534).

Por fim, aproveitamos para fazer um histograma com o tamanho das palavras, o que nos permitiu concluir que este segue uma distribuição normal:

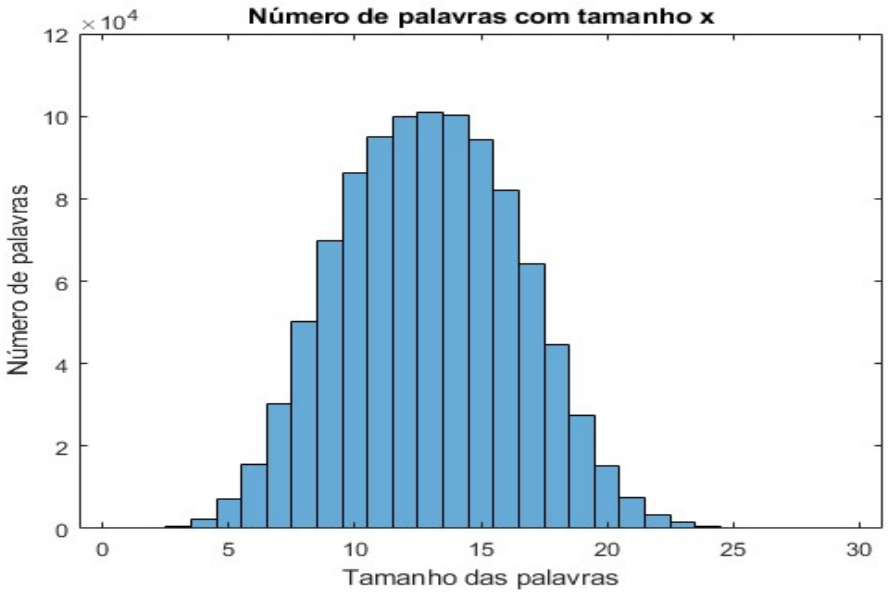

Figura 3: Distribuição do tamanho das palavras.

4 Alguns dos word ladders encontrados

Alguns dos word ladders interessantes que encontramos, incluindo os já mostrados pelo professor Tomás Oliveira e Silva, foram:

```
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 bem mal

Shortest path from bem to mal:
bem -> sem -> sei -> sai -> sal -> mal
```

Figura 4: Word ladder de ”bem”para ”mal”. Mostra um caminho diferente ao dado como exemplo, mas tem o mesmo número de vértices.

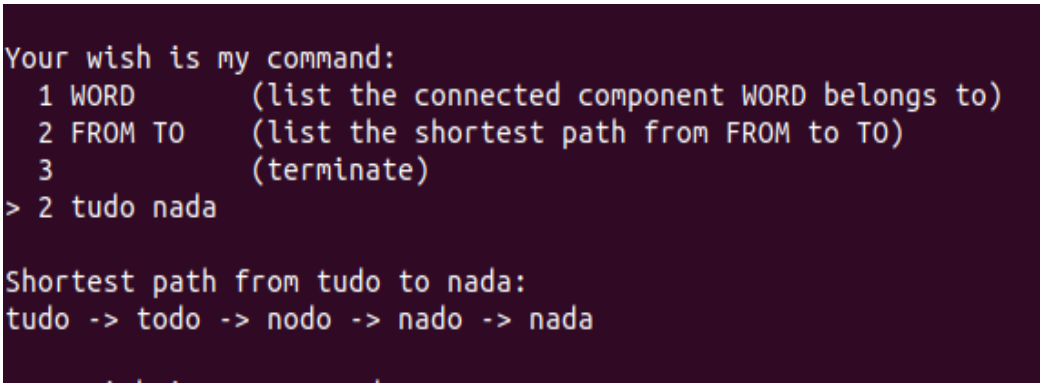

Figura 5: Word ladder de ”tudo”para ”nada”.

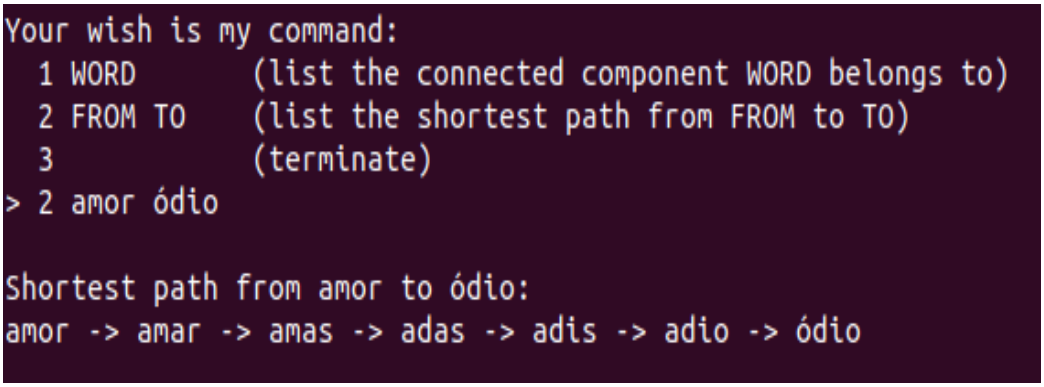

Figura 6: Word ladder de ”amor”para ”ódio”.

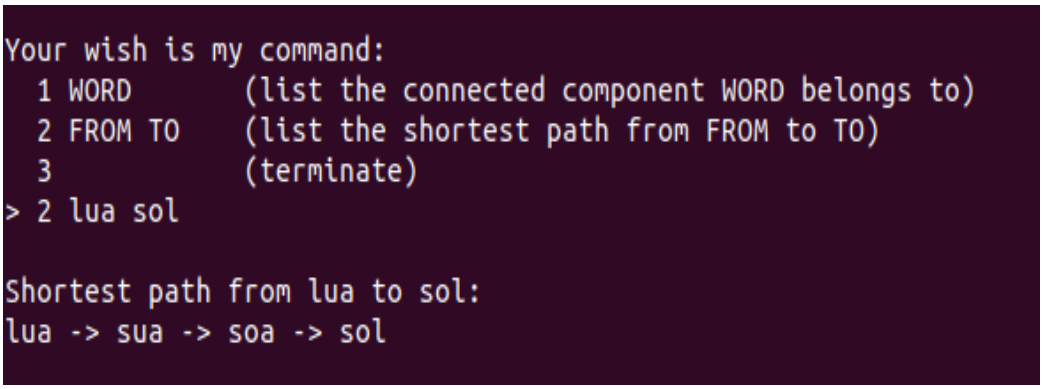

Figura 7: Word ladder de ”lua”para ”sol”.

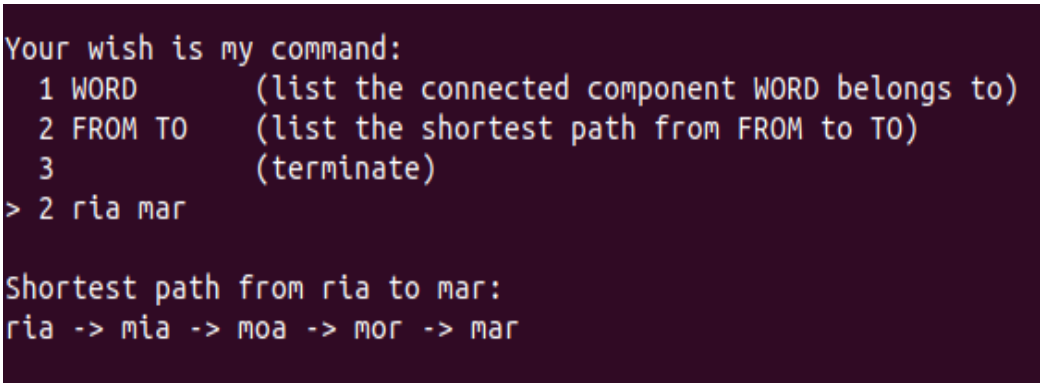

Figura 8: Word ladder de ”ria”para ”mar”.

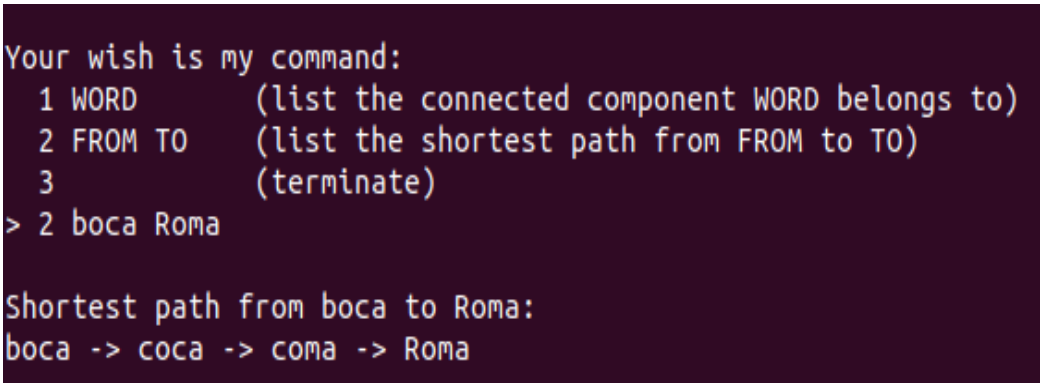

Figura 9: Word ladder de ”boca”para ”Roma”. Afinal, o ditado tem lógica: quem tem ”boca”vai à ”Roma”!

5 Teste de vazamentos de memória

A fim de confirmar que nosso programa não possui qualquer vazamento de memória (memory leaks), corremos o programa com o Valgrind. Obtivemos o resultado a seguir[1], que comprova que de fato libertamos toda a memória que alocamos:

[1] Aqui ambos os ”elapsed times”estão maiores que o normal devido ao uso do Valgrind, e por isso não correspondem aos valores reais.

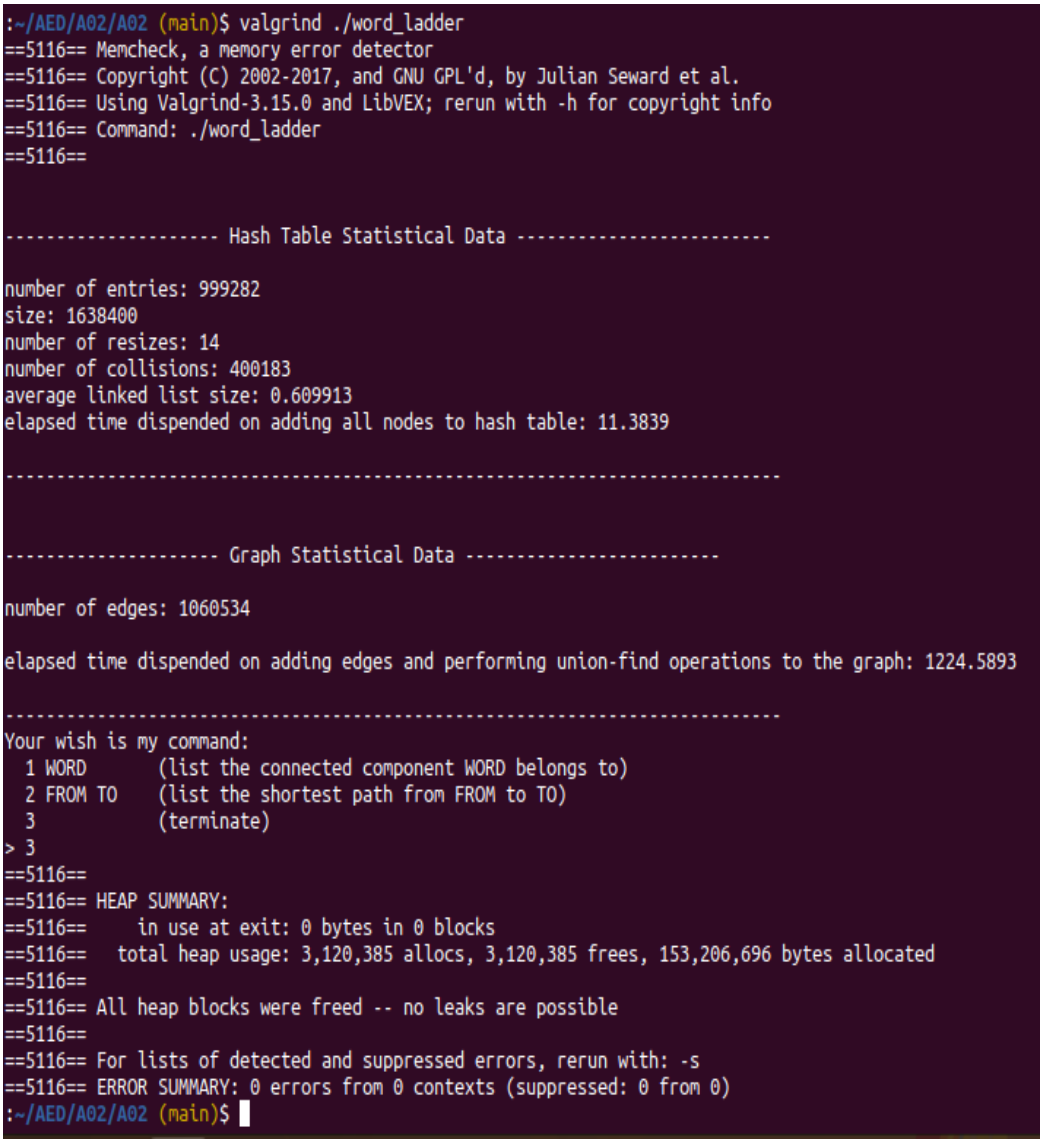

Figura 10: Resultado do Valgrind.

6 Conclusão

A realização desse trabalho nos permitiu aprofundar os conhecimentos sobre as estruturas hash table, grafo, union-find e queue, sobre a técnica de procura breadth first search e sobre ponteiros e linguagem C em geral.

Um dos principais desafios foi a compreensão da lógica por trás do union-find e os representantes. Após estudar o problema em detalhe e compreender seus objetivos, fomos capazes de aplicar os conceitos que obtivemos nas aulas práticas e teóricas sobre os algoritmos e as estruturas de dados utilizados.

7 Webgrafia

Site Weaver - A daily word ladder game (2022). Disponível em:
https://wordwormdormdork.com/
Acesso em 14/01/2022

8 Apêndice

8.1 Estatísticas Hash Table

```
hash_table_stats = readtable(”hash_table_stats.txt”);
hash_table_stats = hash_table_stats(4:end-2,:);
indexes = table2array(hash_table_stats(:,2));
nodes = table2array(hash_table_stats(:,5));
    %% obter hashcodes
hashcodes = [];
indexes = indexes’;
for i = indexes
if nodes(i+1) > 0
hashcodes(end + 1) = i;
end
end
    %% distribuição hashcodes
histogram(hashcodes)
title(”Distribuição de hashcodes”)
ylabel(”Hashcode”)

    %% quantas posições abrigam x = 0,1,2,3.. nós

max(nodes)

histogram(nodes)

title(”Posições na hash table x nós”)

ylabel(”Número de posições”)

xlabel(”Número de nós da lista ligada”)
```

8.2 Estatísticas Grafo

```
graph_info = readtable(”graph_info2.txt”);
graph_info = graph_info(1:end-1,:);
words = table2array(graph_info(:,2));
repList = table2cell(graph_info(:,4));
nvcc = table2array(graph_info(:,6));

    %% tamanho das palavras
wordsSizes = zeros(length(words),1);
for i = 1:length(words)
wordsSizes(i) = length(words{i});
end

wordsSizes = wordsSizes’;
histogram(wordsSizes)
title(”Número de palavras com tamanho x”)
ylabel(”Número de palavras”)
xlabel(”Tamanho das palavras”)
[m, i] = max(wordsSizes);
m
words{i}


    %% número de componentes conexas

rep = unique(repList);

rep = convertCharsToStrings(rep);

length(rep)
```

```
repList = convertCharsToStrings(repList);
biggestCCsize = 0;
[rep, iall, irep] = unique(repList);
iall = iall';
for i=iall
if nvcc(i) > biggestCCsize
biggestCCsize = nvcc(i);
biggestCC = repList{i};
end
end

sum(nvcc(iall)==1)
sum(nvcc(iall))
histogram(nvcc(iall))
biggestCC
biggestCCsize
```

8.3 word_ladder.c

```
//
// AED, November 2022 (Tomás Oliveira e Silva)
//
// Second practical assignement (speed run)
//
// Place your student numbers and names here
//   N.Mec. 105937  Name: Bárbara Nóbrega Galiza
//   N.Mec. 109018  Name: Tomás António de Oliveira Victal
//
// 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>
#include "../P02/elapsed_time.h"
#define  PRINT_ALL_WORDS 0


//
// static configuration
//

#define _max_word_size_  32

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;
typedef struct queue_node        queue_node;
typedef struct queue_l           queue_l;

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)
  unsigned int *nodes_per_head;      // number of nodes mapped to the same hashcode (information purposes)
  hash_table_node_t **heads;         // the heads of the linked lists
};

//QUEUE-struct ------------------------------------------------------
struct queue_node{
  hash_table_node_t *hash_node;
  queue_node *next;
};
struct queue_l{
  unsigned int size;
  queue_node *head;
  queue_node *tail;
};

//QUEUE-funct -------------------------------------------------------
static queue_l *initialize_queue(){
  queue_l *queue = calloc(1,sizeof(queue_l));
  queue->head = NULL;
  queue->size = 0u;
  queue->tail = NULL;

  return queue;
}

static queue_node *allocate_queue_node(){
  queue_node *node = calloc(1,sizeof(queue_node));
  node->next = NULL;
  node->hash_node = NULL;

  return node;
}

static void enqueue(queue_l *queue,hash_table_node_t *hash_node){
  queue->size++;
  queue_node *q_node = allocate_queue_node();
  q_node->hash_node = hash_node;

  if(queue->size != 1u){
    queue->tail->next = q_node;
    queue->tail = q_node;
  }
  else{
    queue->head = q_node;
    queue->tail = q_node;
  }
}

static hash_table_node_t *dequeue(queue_l *queue){

  hash_table_node_t *head_node = queue->head->hash_node;

  if(queue->size > 0u){
    queue->size--;
    queue_node *new_head = queue->head->next;
    free(queue->head);
    if(new_head != NULL) queue->head = new_head;
  }

  return head_node;
}

static void delete_queue(queue_l *queue){
  queue_node *node,*prev_node;
  if(queue->size > 0u){
    node = queue->head;
    while(node != NULL){
      prev_node = node;
      node = node->next;
      free(prev_node);
    }
  }
  free(queue);
}
```

```
static void print_queue_items(queue_l *queue){
  printf("\n- ");
  for(queue_node *node = queue->head;node != NULL;node = node->next){
    hash_table_node_t *hash_node = node->hash_node;
    printf(" %s -",hash_node->word);
  }
  printf("\n\n");
}
//-------------------------------------------------------------------------------------


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



// declaration of functions
static hash_table_t *hash_table_create(void);
static void hash_table_grow(hash_table_t *hash_table);
static void hash_table_free(hash_table_t *hash_table);
static hash_table_node_t *find_word(hash_table_t *hash_table,const char *word,int insert_if_not_found);

//
// 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? no, bc its static and therefore
  ↪  initialized with 0u by default
  {
    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 = 100u;
  hash_table->number_of_entries = 0u;
  hash_table->number_of_edges = 0u;
  hash_table->heads = (hash_table_node_t **)calloc(hash_table->hash_table_size, sizeof(hash_table_node_t*));
  hash_table->nodes_per_head = (unsigned int *)calloc(hash_table->hash_table_size, sizeof(unsigned int));

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

static void hash_table_grow(hash_table_t *hash_table)
{
  //
  // complete this
  //
  unsigned int new_index;
  hash_table_node_t *node,*next_node,*new_node_spot;
  unsigned int new_size = hash_table->hash_table_size * 2;
  hash_table_node_t **new_heads = (hash_table_node_t **)calloc(new_size,sizeof(hash_table_node_t*));
  unsigned int *new_nodes_per_head = (unsigned int *)calloc(new_size, sizeof(unsigned int));
  for(unsigned int i = 0u; i < new_size; i++){
    new_heads[i]= NULL;
  }

  for(unsigned int i = 0u; i < hash_table->hash_table_size; i++){
    node = hash_table->heads[i];
    while(node != NULL){
      next_node = node->next;
      node->next = NULL;
      new_index = crc32(node->word) % new_size;
      new_node_spot = new_heads[new_index];
      if(new_node_spot == NULL){
        new_heads[new_index] = node;
      }
      else{
        while(new_node_spot->next != NULL){
          new_node_spot = new_node_spot->next;
        }
        new_node_spot->next = node;
      }
      node = next_node;
      new_nodes_per_head[new_index] = hash_table->nodes_per_head[i];
    }
  }
  free(hash_table->heads);
  free(hash_table->nodes_per_head);
  hash_table->nodes_per_head = new_nodes_per_head;
  hash_table->heads = new_heads;
  hash_table->hash_table_size = new_size;
}

static void hash_table_free(hash_table_t *hash_table)
{
  //
  // complete this
  //
  for (unsigned int i=0u; i<hash_table->hash_table_size; i++) {
      hash_table_node_t *node = hash_table->heads[i];
      hash_table_node_t *previousNode;
      adjacency_node_t *adjNode;
```

```
      adjacency_node_t *previousAdjNode;
      while (node != NULL){
        adjNode = node->head;
        while(adjNode != NULL){
          previousAdjNode = adjNode;
          adjNode = adjNode->next;
          free_adjacency_node(previousAdjNode);
        }

        previousNode = node;
        node = node->next;

        free_hash_table_node(previousNode);
      }
    }
  free(hash_table->nodes_per_head);
  free(hash_table->heads);
  free(hash_table);
}

// variables to get statistical data about the hash table
int number_of_resizes = 0;
unsigned int number_of_collisions = 0u;
double elapsed_time;

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;
  //
  // complete this
  //
  hash_table_node_t *previous_node = NULL;
  node = hash_table->heads[i];
  unsigned int size_linked_list = 0u;
  // find node
  while(node != NULL){
    if(strcmp(word,node->word) == 0) return node;
    if(node->next == NULL) previous_node = node; // save last non-null node
    node = node->next;
    size_linked_list++;
  }
  if(insert_if_not_found){
    // create new node
    hash_table_node_t *new_node = allocate_hash_table_node();
    new_node->head = NULL;
    new_node->next = NULL;
    new_node->previous = NULL;
    new_node->visited = 0;
    new_node->representative = new_node;
    new_node->number_of_vertices = 1;
    new_node->number_of_edges = 0;
    strcpy(new_node->word,word);
    // link node to hash table
    if(hash_table->heads[i] == NULL){
      hash_table->heads[i] = new_node;
    }
    else{
      previous_node->next = new_node;
      number_of_collisions++;
    }
    // update current linked list size
    size_linked_list++;
    hash_table->nodes_per_head[i] = size_linked_list;
    // updtae number of entries
    hash_table->number_of_entries++;
  }
  // resize hash table
  if(hash_table->number_of_entries >= hash_table->hash_table_size*0.7){
    hash_table_grow(hash_table);
    number_of_resizes++;
  }
  return node;
}


void print_hash_table_statistics(hash_table_t *hash_table)
{
  printf("\n\n---------------------- Hash Table Statistical Data -------------------------\n");
  printf("\nnumber of entries: %u",hash_table->number_of_entries);
  printf("\nsize: %u", hash_table->hash_table_size);
```

```
  printf("\nnumber of resizes: %d", number_of_resizes);
  printf("\nnumber of collisions: %u", number_of_collisions);
  printf("\naverage linked list size: %f",
  ↪  (float)hash_table->number_of_entries/(float)hash_table->hash_table_size);
  printf("\nelapsed time dispended on adding all nodes to hash table: %.4f\n", elapsed_time);
  printf("\n--------------------------------------------------------------------------------\n");
}


//
// 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, *local_representative;

  //
  // complete this
  //
  representative = node->representative;
  next_node = node;
  // find
  while(next_node != representative){
    next_node = next_node->representative;
    representative = next_node->representative;
  }
  // path compression
  next_node = node;
  while(next_node->representative != representative){
    local_representative = next_node->representative;
    next_node->representative = representative;
    next_node = local_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
  //
  if(to != NULL){
    // add link to node "from"
    link = allocate_adjacency_node();
    link->next = NULL;
    link->vertex = to;
    adjacency_node_t *from_links = from->head;
    if(from_links == NULL){
      from->head = link;
    }
    else{
      while(from_links->next != NULL)
        from_links = from_links->next;
      from_links->next = link;
    }

    // add link to node "to"
    link = allocate_adjacency_node();
    link->next = NULL;
    link->vertex = from;
    adjacency_node_t *to_links = to->head;
    if(to_links == NULL){
      to->head = link;
    }
    else{
      while(to_links->next != NULL)
        to_links = to_links->next;
      to_links->next = link;
    }

    // increment total number of edges
    hash_table->number_of_edges++;

    from_representative = find_representative(from);
    to_representative = find_representative(to);
    // union
    if(from_representative != to_representative){
      if(to_representative->number_of_vertices > from_representative->number_of_vertices){
```

```
        to->representative->number_of_vertices += from_representative->number_of_vertices;
        to->representative->number_of_edges += from_representative->number_of_edges + 1;
        from->representative->representative = to_representative;
      }
      else{
        from->representative->number_of_vertices += to_representative->number_of_vertices;
        from->representative->number_of_edges += to_representative->number_of_edges + 1;
        to->representative->representative = from_representative;
      }
    }
  }
}


//
// 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
//
hash_table_node_t *last_vertex = NULL;
queue_l *all_vertices;

static int breadh_first_search(hash_table_node_t *origin,hash_table_node_t *goal)
{
  //
  // complete this
  //
  queue_l *queue = initialize_queue();
  all_vertices = initialize_queue();
  hash_table_node_t *hash_node;
  hash_table_node_t *vertex = NULL;
  enqueue(queue,origin);
  enqueue(all_vertices,origin);
  origin->visited = 1;

  while(queue->size > 0u){
    hash_node = dequeue(queue);
    if(hash_node == goal) break;
    for(adjacency_node_t *adj_node = hash_node->head; adj_node != NULL; adj_node = adj_node->next){
      vertex = adj_node->vertex;
      if(vertex->visited == 1){
        continue;
      }
      else{
        vertex->visited = 1;
        vertex->previous = hash_node;
        enqueue(queue,vertex);
        enqueue(all_vertices,vertex);
      }
    }
  }
  int number_of_vertices_visited = all_vertices->size;

  delete_queue(queue);
  return number_of_vertices_visited;
}

static void breadth_first_search_reset(hash_table_t *hash_table){
  hash_table_node_t *node;
  unsigned int i;
  for(i = 0u;i < hash_table->hash_table_size;i++)
    for(node = hash_table->heads[i];node != NULL;node = node->next){
      node->previous = NULL;
      node->visited = 0;
    }

}

//
// 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
  //
  hash_table_node_t *node;
```

```
  // get node
  node = find_word(hash_table,word,0);
  if(node == NULL){
    printf("\n%s isn't on the list of words!\n\n",word);
    return;
  }

  // print connected component
  printf("This connected component has the total of %d vertices, which are:\n",
  ↪  breadh_first_search(node,NULL));
  print_queue_items(all_vertices);

  // clear data
  breadth_first_search_reset(hash_table);
  delete_queue(all_vertices);
}


//
// 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 swapWordsOrder(char* str, const char* separator)
{
    int len = strlen(separator);
    memmove(str + len, str, strlen(str) + 1);
    memcpy(str, separator, len);
}

static void path_finder(hash_table_t *hash_table,const char *from_word,const char *to_word)
{
  //
  // complete this
  //
  hash_table_node_t *origin,*goal;

  // get origin and goal nodes
  origin = find_word(hash_table,from_word,0);
  goal = find_word(hash_table,to_word,0);
  if(origin == NULL){
    printf("\n%s isn't on the list of words!\n\n",from_word);
    return;
  }
  else if(goal == NULL){
    printf("\n%s isn't on the list of words!\n\n",to_word);
    return;
  }

  // check if there is a possible path (if both are from same connected component)

  if(find_representative(origin) != find_representative(goal)){
    printf("\nThere is no path!\n\n");
    return;
  }

  // breadth first search to find shortest path
  breadh_first_search(origin,goal);

  // print shortest path
  char shortestPath[1024] = "";
  int firstWord = 1;

  printf("\nShortest path from %s to %s:\n", from_word, to_word);

  while(goal != NULL){
    if(!firstWord) swapWordsOrder(shortestPath, " -> ");
```

```
    swapWordsOrder(shortestPath, goal->word);
    if(goal->word == origin->word) break;
    goal = goal->previous;
    firstWord = 0;
  }

  printf("%s", shortestPath);
  printf("\n\n");

  // clear data
  breadth_first_search_reset(hash_table);
  delete_queue(all_vertices);
}


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

static void graph_info(hash_table_t *hash_table)
{
  //
  // complete this
  //
  printf("\n\n---------------------- Graph Statistical Data --------------------------\n");
  printf("\nnumber of edges: %u\n",hash_table->number_of_edges);
  #if PRINT_ALL_WORDS
  hash_table_node_t *node, *rep;
  for(unsigned int i=0u; i<hash_table->hash_table_size; i++){
    node = hash_table->heads[i];
    while(node != NULL){
      rep = find_representative(node);
      printf("node: %-46s representative: %-46s \nnumber of vertices of connected component: %-d      number
      ↪  of edges of connected component: %-d\n", node->word, rep->word, rep->number_of_vertices,
      ↪  rep->number_of_edges);
      node = node->next;
    }
  }
  #endif
  printf("\nelapsed time dispended on adding edges and performing union-find operations to the graph:
  ↪  %.4f\n", elapsed_time);
  printf("\n------------------------------------------------------------------------------\n");
}


//
// 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);
  }

  elapsed_time = cpu_time();

  while(fscanf(fp,"%99s",word) == 1)
    (void)find_word(hash_table,word,1);

  elapsed_time = cpu_time() - elapsed_time;
  fclose(fp);

  print_hash_table_statistics(hash_table);

  // find all similar words
  elapsed_time = cpu_time();

  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);

  elapsed_time = cpu_time() - elapsed_time;

  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;
}
```