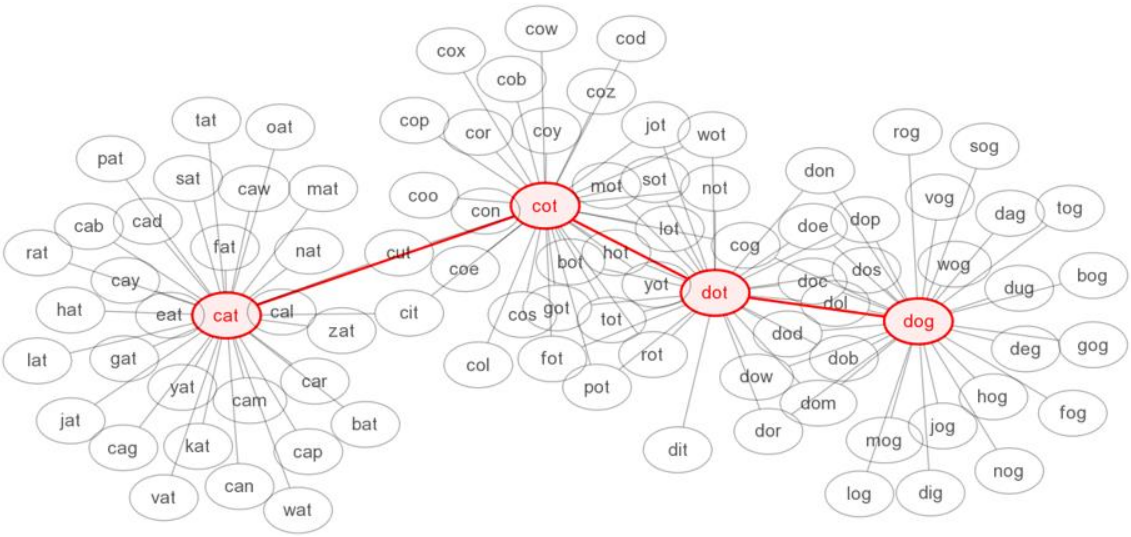
Trabalho Word Ladder

2º Relatório

Diogo Ferreira – 99984 (25%)

Miguel Miragaia – 108317 (37,5%)

Gonçalo Lopes – 107572 (37,5%)



DETI

Universidade Aveiro

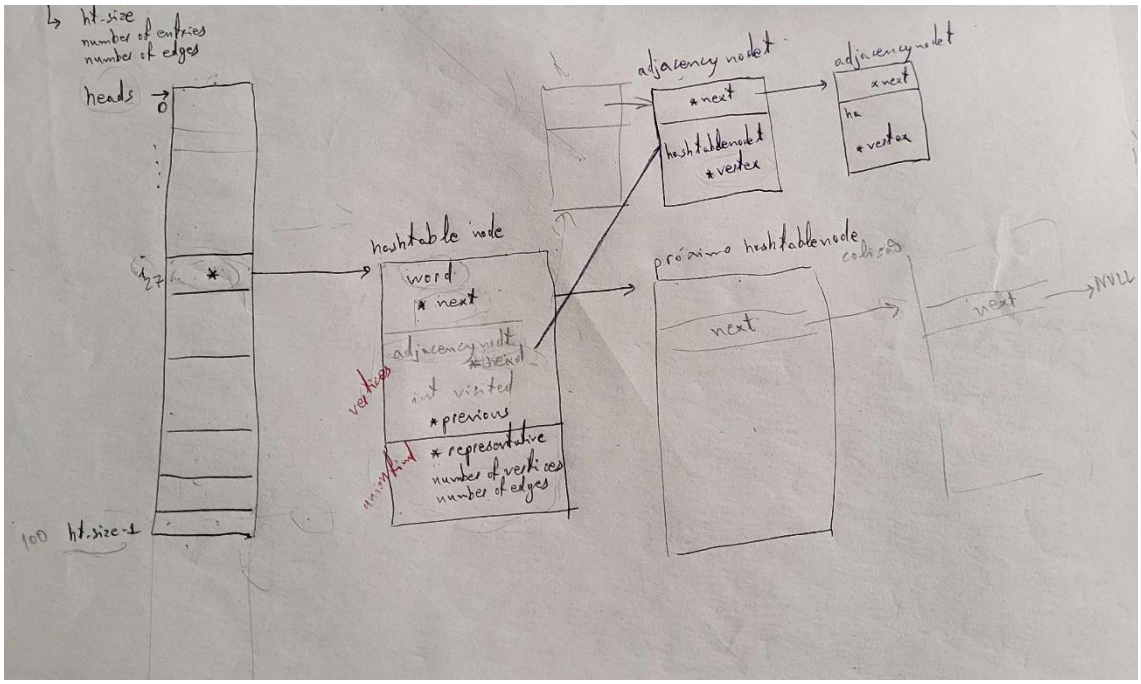
03/12/2022

Índice

1 – Introdução

O segundo trabalho consiste, de uma forma muito sucinta, na construção de uma ***"word ladder"*** usando como base operacional uma *hash table*.

"Word ladder" consiste de uma sequência de palavras das quais duas letras adjacentes diferem por um letra, ou seja, em português é possível ir de *tudo* a *nada* em apenas quatro passos: *tudo* → *todo* → *nodo* → *nado* → *nada*.



Todo o codigo utilizado ao longo deste relatório estará na sua integra no **Anexo** final.

2 – Resolução

Para conseguirmos obter uma implementação viável para a meta de termos todas as palavras do dicionário na ***"word ladder"***, foi nos fornecido um codigo com uma função hash table muito elementar, onde o nosso propósito inicial foi completar as funções obrigatórias para um funcionamento elementar do trabalho, bem como outras funções opcionais além disso para uma melhor complementação e facilidade de funcionamento e tratamento de dados.

2.1 – Obrigatório

Sendo a implementação inicial do programa insuficiente para podermos gerar a tal escada de palavras tivemos de pegar em partes existentes do codigo e alterá-las, sendo nuns casos mudanças ligeiras e noutros mudanças significativas.

Assim tivemos que completar 5 funções diferentes.

2.1.1 – Hash_Table_Create

A primeira mudança imperial para o sucesso deste trabalho era termos uma Hash Table onde conseguíssemos guardar as palavras para que estas pudessem mais tarde ser usadas na nossa ***"word ladder"***.

Com isto em mente reservamos memoria do nosso sistema operativo para que as palavras permaneçam até serem invocadas, sendo que inicialmente pedimos memoria suficiente para 1000 palavras. Caso a memoria não seja o bastante o sistema irá

imprimir uma mensagem de erro a avisar a falta de memoria para a hash table.

2.1.2 – Hash_Table_Grow

Tendo a nossa meta em mente bem como a função **2.1.1 – Hash_Table_Create**, podemos ver um problema claro na nossa execução, sendo ela a falta de espaço.

Deste modo surgiu a necessidade de a Hash Table poder indo crescendo ao longo que mais palavras lhe são adicionadas, tendo sido modificada esta função, emitindo uma mensagem de erro quando chegamos ao nosso limite.

```
new_head = malloc(new_size *sizeof(hash_table_node_t*));
    if( new_head == NULL ) {
      fprintf(stderr,"grow_hash_table: out of memory\n");
      exit(1);
```

Fazer crescer a nossa *Hash Table* consiste em mapearmos o quanto de espaço é preciso para a nova *"hash function"* através de condições internas e iterarmos cada elemento da antiga *Hash Table* para a nova.

No fim destes procedimentos, atualizamos a nossa *Hash Table* e libertamos também toda memoria da antiga através do método seguinte, **2.1.3**.

```
free(hash_table->heads);
hash_table->heads=new_head;
hash_table->hash_table_size=new_size;
```

À medida que temos crescimento de palavras, para mantermos uma boa execução do programa decidimos forçar a *Hash Table* a ter um tamanho ímpar

```
unsigned int new_size= 2 * (hash_table->hash_table_size + 1);
```

Mantendo assim o seu bom desempenho, sendo que para conseguirmos obter o máximo potencial de desempenho seria preciso termos um tamanho primo de dados.

2.1.3 – Hash_Table_Free

Como fazemos a nossa Hash Table ter um crescimento a própria função irá ter vários "resizings" onde cada um deles irá precisar e usar cada vez mais espaço para poder albergar um número crescente de chaves para as nossa palavras. Assim é imperial após cada um destes "resizings" fazer com que a memoria seja limpa após cada ciclo e cada utilização.

Deste modo vasculhamos todas as entradas da nossa *"hash function"* onde de cada vez que acedemos a uma "head" atualizamo-la para um valor *NULL* permitindo-nos no final conseguir limpar tudo com o comando *free().*

2.1.4 – Find Word

Com cerca de 1 milhão de palavras é importante sabermos se no futuro quisermos adicionar outras, sabermos quais palavras já se encontram disponíveis nos nossos índices. Assim expandimos esta função de forma, a que ela receba uma chave de uma palavra especifica proveniente do sou "pointer" que queremos verificar, em que usando a fórmula:

```
    i = crc32(word) % hash_table->hash_table_size;
```

Podemos obter o índice que essa palavra tenha dentro da nossa tabela. Daqui temos 2 opções: Ela já existe na *hash table* ou não.

Se ela existir muito bem, mas caso não exista a função ira atualizar o seu número de entradas, bem como a suas listas de adjacência para poder incluir a palavra, usando o método **2.1.2**.

2.1.5 – Dados Estatísticos

Ao contrário de todas as outras funções até agora, nesta para podermos ter um melhor olhar da nossa *hash table*, temos só de pedir á própria um output do seu número de entradas, colisões internas, bem como o seu tamanho atual.

```
printf("Number of entries: %d\nHash Table size: %d\n",
    hash_table->number_of_entries,
    hash_table->hash_table_size);
```

2.2 – Opcional

Após termos completado a implementação mandatória da *Hash Table*, decidimos encarar os desafios recomendados listados no cabeçalho de trabalho, consistentes em criar um grafo onde listamos todas as componentes conexas entre palavras similares bem, como através deste grafo conseguirmos mostrar o caminho mais curto de duas palavras sendo a inicial introduzida pelo utilizador.

2.2.1 – Find Representative

Representative, ou representantes, são elementos armazenados na *Hash Table* que possuem uma chave específica.

```
hash_table_node_t *representative,*next_node;
representative = node;
```

Para podermos encontrar um representante é necessário aplicar uma função de *hashing* á chave desejada para podermos encontrar o índice do elemento desejado corresponde na tabela.

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

Caso tal representante não seja encontrado, então imprime uma mensagem de erro:

```
while (representative != representative->representative) {
    if (representative == NULL){
      fprintf(stderr,"find_representative: representative not found\n");
      exit(1);
    }
```

```
    representative = representative->representative;
  }
```

2.2.2 – Add_Edge

Esta função recebe da *hash table* um nó de onde vimos, **from*, e de uma "try word", para onde poderemos ir.

Usando:

```
        to = find_word(hash_table,word,0);
```

sabemos se a "try word" é uma palavra já existente na *hash table* para a criação de uma *edge*. Caso a palavra não existe ela não será incluída na *hash table* devido ao facto de nesta altura já termos incluído na nossa tabela todas as palavras possíveis do

dicionário, ou seja, se a nossa "try word" não estiver na nossa tabela esta é uma palavra invalida.

Sendo assim, no caso invalido:

```
if(to == NULL)
    return;
```

onde como se trata de uma palavra invalida será impossível a criação de um nó para o nosso grafo.

No caso inverso, onde de facto a palavra já se encontra na *hash table* teremos de alocar espaço de memória através da função insert_edge, onde usamos a função allocate_adjacency_node_t. Com este processo temos assim o nosso nó criado, pronto a ser usado no grafo das componentes ligadas.

2.2.3 – Insert_Edge

Esta função na nossa *Hash Table* basicamente adiciona uma nova chave-valor na tabela. Esta chave é usada para calcular o índice da posição na tabela onde existia previamente o valor antigo. Caso exista uma chave igual na tabela, o valor antigo é substituído pelo novo, sendo que contrariamente uma nova posição é criada na tabela para armazenar o valor.

```
  adjacency_node_t *link;

  link = allocate_adjacency_node();
  link->vertex = to;
  link->next = from->head;
  from->head = link;
  hash_table->number_of_edge_nodes++;
```

2.2.4 – Mark All Vertices

Esta função de forma muito breve, visita de forma recursiva todo o espaço da *Hash Table* em procura de entradas vazias, *NULL*, na qual marcamos estas entradas de forma a não serem incluídas nas estatísticas dos vertices para a montagem do grafo.

```
for (unsigned int i = 0; i < hash_table->hash_table_size; i++) {
        hash_table_node_t *node = hash_table->heads[i];
        while (node != NULL) {
            node->visited = 0;
            node = node->next;
        }
    }
```

2.2.5 – Breadth_First_Search

Para podermos avaliar quais palavras são elegíveis para a formação de um nó, através da **Mark_All_Vertices** (**2.2.3**), temos que primeiro encontrar todas as possibilidades para assim as

poder avaliar e analisar. Assim surge a necessidade da implementação de um algoritmo que possa realizar tal tarefa, tendo sido escolhido no cabeçalho inicial do trabalho o **Breadth First Search** ou **BFS**, como será referido daqui em diante.

O **BFS** é um algoritmo de tratamento de dados para essencialmente árvores e grafos, sendo no último importante ter precauções para vértices repetidos. Ele é ótimo para podermos encontrar qualquer solução rasa pretendida.

O que torna isto possível é a maneira em como temos de implementar o **BFS**, onde evitamos ao máximo o avanço para ramos mais profundos dos nossos dados, procurando assim possíveis soluções em todos os elementos de cada nível antes de passar ao próximo. Este modo de resolução apesar de ser o mais eficiente para grandes quantidades de dados, consume também mais memoria do que os seus métodos homólogos, pois para cada nível e solução ainda não avaliada temos de ter a certeza que o sistema tem memoria suficiente para a sua realização.

Para implementarmos o nosso **BFS** primeiro criamos um *deque(**2.2.5.1**)*:

```
unsigned int    num_visited;
hash_table_node_t *node;
adjacency_node_t  *neighbour;
deque_t     *deque;

deque = create_deque(maximum_number_of_vertices);

num_visited = 0;
```

Agora, adicionamos o vértice de onde começamos ao *deque*:

```
put_hi(deque, origin);
```

De seguida, enquanto a nossa *deque* não estiver vazia, obtemos o vértice atual (**1**) e iteramos pelos nós vizinhos não visitados e adicionamo-los a uma lista com o índice referente ao número de nós visitados (**2**) adicionando também no nosso *deque* quaisquer vizinhos não visitados (**3**):

1:

```
while (deque->size > 0 && num_visited < maximum_number_of_vertices)
  {
    node = get_low(deque);
    node->visited = 1;
```

2:

```
if (list_of_vertices)
      list_of_vertices[num_visited] = node;
    num_visited++;
    if (node == goal)
      break;
     for(neighbour = node->head; neighbour ; neighbour
= neighbour->next)
```

3:

```
if (!neighbour->vertex->visited)
      {
        neighbour->vertex->visited = 1;
        neighbour->vertex->previous = node;
        put_hi(deque, neighbour->vertex);
      }
```

Após terem sido visitado todos os vértices vizinhos do inicial, fazemos “reset” á lista de visitados recentemente para usos futuros. Finalmente apagamos o deque e retornamos o número total de vértices visitados caso a criação de um nó seja possível.

Todo este processo será repetido todas as vezes que precisaremos de avaliar os nós para todas as palavras dentro da nossa *Hash Table*.

2.2.5.1 – Deques

Os *deques*, conhecidos também por pilhas de dados, são estruturas de dados que nos permitem a inserção e remoção de dados em ambos as extremidades. Implementa-se geralmente por base de listas ligadas e arranjos, sendo que neste trabalho optamos por usar em conjunto com a *Hash Table.*

Inicialmente introduzimos a *structure* do nosso deque composta pelo seu tamanho, variável de verificação de estar preenchida (*full*), etc.

```
typedef struct deque_s
{
  void **items;
  unsigned int hi;
  unsigned int low;
  unsigned int size;
  unsigned int maxsize;
  int full;
} deque_t;
```

```
static deque_t *create_deque(int maxsize)
{
  deque_t *deque = malloc(sizeof(deque_t));
  if(deque == NULL)
  {
    fprintf(stderr,"create_deque: out of memory\n");
    exit(1);
  }
  deque->items = (void **)malloc(sizeof(void *) * maxsize);
  if(deque->items == NULL){
    fprintf(stderr,"create_deque->circular_array: out of memory\n");
    free(deque);
    exit(1);
  }
```

```
  deque->maxsize = maxsize;
  deque->hi = 0;
  deque->low = 0;
  deque->full = 0;
  deque->size = 0;
  return deque;
}
```

Têm por operações comuns *push* (inserir elementos) e *pop* (remover elementos), sendo também usados *front* (início do *deque*) e *back* (final do *deque*) como diretrizes da direção da pilha.

No nosso trabalho prático usamos variações destas operações bases, sendo elas: *put_hi* e *get_low*, sendo estas os equivalente ao usarmos *push_front* e *pop_back*, adicionando um elemento ao final do *deque* e removendo um elemento no início do *deque* respetivamente.

```
static void put_hi(deque_t *deque, void *item)
{
  assert(deque->size < deque->maxsize);
  deque->items[deque->hi] = item;
  deque->hi = (deque->hi + 1) % deque->maxsize;
  deque->size++;
}

static void *get_low(deque_t *deque)
{
  assert(deque->size > 0);
  void *ret = deque->items[deque->low];
  deque->low = (deque->low + 1) % deque->maxsize;
  deque->size--;
  return ret;
}
```

Existem várias maneiras de podermos implementar *deques*, sendo as mais comuns por base arranjos, onde usamos diferentes variáveis para armazenamento dos índices do início e final do *deque*, podendo mover estes índices utilizando as operações de adição e remoção.

Por fim é necessário libertarmos a memoria alocada para o deque, usamos a função delete_deque.

```
void delete_deque(deque_t *deque)
{
  free(deque->items);
  free(deque);
}
```

2.2.6 – List_Connected_Components

Esta lista tem o objetivo de verificar se uma especifica palavra existe na *Hash Table* com o auxílio da **Find_Word** (**2.1.4**), onde em caso negativo fazemos um "ouput" de uma mensagem de erro para avisar o utilizador.

```
origin = find_word(hash_table, word, 0);
  if (!origin)
  {
    printf("\nThe word: %s doesn't exist\n", word);
    return;
  }
```

Em caso positivo invoca a função **Mark all vertices** ().

```
mark_all_vertices(hash_table);
```

Após este processo, primeiramente é localizado o seu representante usando **Find_Representative** (**2.2.1**), de onde de seguida alocamos memoria para um *array* chamado **List of vertices**, guardando assim o array com os vértices do componente conexo inicial.

```
representative = find_representative(origin);
list_of_vertices = malloc(representative-
>number_of_vertices * sizeof(hash_table_node_t *));
```

Finalmente a **BFS** (**2.2.5**) é utilizada com o propósito de preencher o *array* com o vertices conexos, sendo que o tamanho final será

guardado no **List_length**, fazendo uma iteração sobre cada vértice do *array*, fazendo novamente mais um "output" na forma de uma *printf* de cada palavra, libertando simultaneamente a memoria alocada para o array.

```
list_length = breadh_first_search(representative->number_of_vertices,
list_of_vertices, origin, NULL);
  for (i=0; i < list_length; i++){
    printf("%s\n", list_of_vertices[i]->word);
  }
 free(list_of_vertices);
```

2.2.7 – Path_Finder

Na introdução foi explicado que para pudermos ter uma ***"word ladder"*** é necessário termos uma palavra de onde começamos e outra para a qual queremos chegar.

Esta função consiste no verificamento de ambas as palavras de onde começamos, início, e para onde vamos, destino.

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

Em caso negativo, ou seja, se uma das duas palavras não existir simplesmente retornamos ao utilizador uma mensagem de erro.

```
if (!from) {
        fprintf(stderr, "\nWord not found: %s\n", from_word);
        return;
    }
    if (!to) {
        fprintf(stderr, "\nWord not found: %s\n", to_word);
        return;
    }
```

Em caso positivo é chamada a função **BFS** (**2.2.5**) para o mapeamento do caminho mais curto entre ambas as palavras, sendo neste processo também utilizada a função **Mark all vertices** (**2.2.4**). Se o tamanho do percurso for igual a 0 significa que as

palavras não se encontram conectadas sendo neste caso novamente imprimido no terminal uma mensagem de erro. No caso oposto, onde realmente existe um caminho usamos um ciclo *while*, iterando todas as palavras presentes no caminho bem como o seu índice.

Quando este ciclo encontra por fim a sua palavra destino, imprime tudo mais uma vez no terminal.

```
mark_all_vertices(hash_table);
    size_t list_len =
breadh_first_search(find_representative(to)->number_of_vertices,
NULL, to, from);
    if (list_len == 0) {
        fprintf(stderr, "Words are not connected\n");
    } else {
        size_t i = 0;
        while (from && from != to) {
            printf("  [%zu] %s\n", i++, from->word);
            from = from->previous;
        }
        printf("  [%zu] %s\n", i++, from->word);
    }
```

2.2.8 – Graph_Info

Similarmente ao **2.1.5**, é possível saber quantas palavras, arestas (**2.2.2**) e componentes temos ao nosso dispor usando um simples output:

```
printf("\nNodes: %u\nEdges: %u\nComponents: %u\n",
      hash_table->number_of_entries,
      hash_table->number_of_edges,
      hash_table->number_of_components);
```

Desta forma é sempre possível termos noção nas fazes iniciais do trabalho de sabermos quantas palavras já foram adicionadas e com isso da escala de interligação de componentes entre estas.

3 – Resultados

Com todas as obrigatoriedades e grande parte das opcionalidades totalmente cumpridas, estamos prontos para a execução da nossa ***"word ladder"***.

Quando corremos inicialmente o programa, é mostrado ao utilizador um menu de diferentes funcionalidades do programa:

```
Nodes: 999282
Edges: 1060534
Components: 377234
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            (list hash table stats)
  4            (list graph info)
  0            (terminate)
>
```

De forma a verificarmos se parte **obrigatória (2.1)** foi realmente bem implementada, decidimos começar com a segunda opção do programa, o caminho mais curto de uma palavra a outra. Com isto em mente, decidimos testar as palavras antónimas, veloz – lento:

```
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            (list hash table stats)
  4            (list graph info)
  0            (terminate)
> 2
veloz
lento
  [0] veloz
  [1] velou
  [2] melou
  [3] meloa
  [4] melga
  [5] meiga
  [6] meigo
  [7] leigo
  [8] leito
  [9] lento
```

Tendo sido a listagem entre as palavras um sucesso, movemos os nossos testes para a terceira opção, as estatísticas da *Hash Table*:

```
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            (list hash table stats)
  4            (list graph info)
  0            (terminate)
> 3
Number of entries: 999282
Collisions: 208481
Hash Table size: 2052094
```

Com estes dados, conseguimos observar um número de entradas perto de 1 milhão, sendo estas todas as palavras do dicionário Português, realizando-se assim o nosso objetivo de conseguirmos atingir cerca de 1 milhão de palavras.

Finalmente com o auxílio da quarta opção, conseguimos obter os dados do grafo:

```
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            (list hash table stats)
  4            (list graph info)
  0            (terminate)
> 4

Nodes: 999282
Edges: 1060534
Components: 377234
```

4 – Conclusão

Ao longo deste relatório, mostramos todas as diferentes componentes necessárias para o funcionamento da ***"word ladder"***, algumas tendo sido de realização **obrigatória**, cruciais para a integridade operacional do programa, e outras **opcionais**, visando o tratamento de dados e estatísticas para o grafo.

Conseguimos assim, o cumprimento de praticamente todas as nossas metas sobre o funcionamento do codigo, bem como a meta de usarmos todas as palavras do dicionário português.

5 - Anexo

```
//
// AED, November 2022 (Tomás Oliveira e Silva)
//
// Second practical assignement (speed run)
//
// Place your student numbers and names here
//   N.Mec. 107572  Name: Gonçalo Lopes
//   N.Mec. 108317  Name: Miguel Miragaia
//   N.Mec. 99984   Name: Diogo Ferreira
//
// Do as much as you can
//   1) MANDATORY: complete the hash table code
//      *) hash_table_create  ##feito n
//      *) hash_table_grow    ##feito n
//      *) hash_table_free    ##feito n
//      *) find_word          ##feito d
//      +) add code to get some statistical data about the hash table
##feito n/d
//   2) HIGHLY RECOMMENDED: build the graph (including union-find
data) -- use the similar_words function...
//      *) find_representative  ##feito n/d
//      *) add_edge   ##feito n/d
//   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   ##feito d/n
//   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 <stdbool.h>
```

```c
#include <assert.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(key)
  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;
  unsigned int number_of_collisions;
  unsigned int number_of_components;
  unsigned int number_of_edge_nodes;      // number of edges (for
information purposes only)
  hash_table_node_t **heads;         // the heads of the linked lists
};

typedef struct deque_s
{
  void **items;
  unsigned int hi;
  unsigned int low;
  unsigned int size;
  unsigned int maxsize;
  int full;
} deque_t;

static deque_t *create_deque(int maxsize)
{
  deque_t *deque = malloc(sizeof(deque_t));
  if(deque == NULL)
      {
            fprintf(stderr,"create_deque: out of memory\n");
            exit(1);
      }
  deque->items = (void **)malloc(sizeof(void *) * maxsize);
  if(deque->items == NULL){
            fprintf(stderr,"create_deque->circular_array: out of
memory\n");
            free(deque);
            exit(1);
      }
  deque->maxsize = maxsize;
  deque->hi = 0;
  deque->low = 0;
  deque->full = 0;
  deque->size = 0;
  return deque;
}

static void put_hi(deque_t *deque, void *item)
{
      assert(deque->size < deque->maxsize);
      deque->items[deque->hi] = item;
      deque->hi = (deque->hi + 1) % deque->maxsize;
      deque->size++;
}

static void *get_low(deque_t *deque)
{
      assert(deque->size > 0);
      void *ret = deque->items[deque->low];
      deque->low = (deque->low + 1) % deque->maxsize;
```

```
      deque->size--;
      return ret;
}

void delete_deque(deque_t *deque)
{
  free(deque->items);
  free(deque);
}


//
// 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 = 1000;
  hash_table->number_of_entries = 0u;
  hash_table->number_of_components = 0u;
  hash_table->number_of_collisions = 0u;
  hash_table->number_of_edges =0u;
  hash_table->number_of_edge_nodes = 0u;
  if( ( hash_table->heads = (hash_table_node_t **) malloc( hash_table-
>hash_table_size * sizeof(hash_table_node_t*) ) ) == NULL ) {
            fprintf(stderr,"create_hash_table: out of memory\n");
    exit(1);
      }
      for( i = 0; i < hash_table->hash_table_size; i++ ) {
```

```
            hash_table->heads[i] = NULL;
      }

      return hash_table;
}

static void hash_table_grow(hash_table_t *hash_table)
{
  //
  // complete this
  //
  unsigned int new_index;
  unsigned int new_size= 2 * (hash_table->hash_table_size + 1);
  hash_table_node_t **new_head;
  hash_table_node_t *temp_next;
  if (hash_table->number_of_collisions > 0 && (hash_table-
>hash_table_size / hash_table->number_of_collisions) < 5)
      {
    new_head = malloc(new_size *sizeof(hash_table_node_t*));
    if( new_head == NULL ) {
      fprintf(stderr,"grow_hash_table: out of memory\n");
      exit(1);
    }
    hash_table->number_of_collisions = 0u;
    for(unsigned int i=0; i < hash_table->hash_table_size;i++){
      hash_table_node_t *tmp = hash_table->heads[i];
      while(tmp!=NULL){
        new_index = crc32(tmp->word)%new_size;
        temp_next = tmp->next;
        tmp->next = new_head[new_index];
        if (tmp->next){
          hash_table->number_of_collisions++;
        }
        new_head[new_index] = tmp;
        tmp = temp_next;
      }
    }
    free(hash_table->heads);
    hash_table->heads=new_head;
    hash_table->hash_table_size=new_size;
  }
}


static void hash_table_free(hash_table_t *hash_table)
{
  hash_table_node_t *temp_next;

  for(unsigned int i=0; i < hash_table->hash_table_size;i++){
    hash_table_node_t *tmp = hash_table->heads[i];
    while(tmp!=NULL){
      temp_next = tmp->next;
```

```
      //quando houver adjacencias tenho de as libertar aqui. as
adjacencias estao no campo head ( adjacencie_node_t) ciclo while
percorrer a lista de adjacencias e libertar cada um dos nós
      adjacency_node_t *adj_tmp = tmp->head;
      while (adj_tmp != NULL) {
        adjacency_node_t *adj_temp_next = adj_tmp->next;
        free_adjacency_node(adj_tmp);
        adj_tmp = adj_temp_next;
      }
      free(tmp);
      tmp = temp_next;
    }
  }
  free(hash_table->heads);
  free(hash_table);
}

//função que cria um novo nó
static hash_table_node_t *create_node(const char *word)
{
      hash_table_node_t *node = allocate_hash_table_node();
  node->next = NULL;
  node->head = NULL;
  node->visited = 0;
      node->representative = node;
      node->previous = NULL;
      node->number_of_vertices = 1;
      node->number_of_edges = 0;
      strcpy(node->word, word);
      return node;
}


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)
      {
            if (strcmp(node->word, word) == 0)
                  return node;
            node = node->next;
      }
      if (insert_if_not_found)
      {
            node = create_node(word);
            if (hash_table->heads[i]){
      hash_table->number_of_collisions++;
    }
```

```
            node->next = hash_table->heads[i];
            hash_table->heads[i] = node;
    hash_table->number_of_components++;
            hash_table->number_of_entries++;
            hash_table_grow(hash_table);
      }
  return node;
}

static void hash_table_stats(hash_table_t *hash_table)
{
      printf("Number of entries: %u\nCollisions: %u\nHash Table size:
%u\n",
                  hash_table->number_of_entries,
      hash_table->number_of_collisions,
                  hash_table->hash_table_size);
}



//
// 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) {
    if (representative == NULL){
      fprintf(stderr,"find_representative: representative not
found\n");
      exit(1);
    }
    representative = representative->representative;
  }

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

static void insert_edge(hash_table_t *hash_table, hash_table_node_t
*from, hash_table_node_t *to)
{
      adjacency_node_t *link;

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

```
      from->head = link;
      hash_table->number_of_edge_nodes++;
}

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

  to = find_word(hash_table,word,0);
  if(to == NULL){
         return;
  }
  link = from->head;
  while (link != NULL && link->vertex != to) {
    link = link->next;
  }
  if (link)
           return;
  link = to->head;
  while (link != NULL && link->vertex != from) {
    link = link->next;
  }
  if (link)
           return;
  hash_table->number_of_edges++;

  insert_edge(hash_table, from, to);
      insert_edge(hash_table, to, from);

  from_representative = find_representative(from);
      to_representative = find_representative(to);
  if (from_representative != to_representative) {
    unsigned int vert_sum = from_representative->number_of_vertices +
to_representative->number_of_vertices;
    unsigned int edge_sum = from_representative->number_of_edges +
to_representative->number_of_edges;
    hash_table_node_t *new_rep;

    if (to_representative->number_of_vertices > from_representative-
>number_of_vertices) {
        new_rep = from_representative;
        to_representative->representative = new_rep;
    }
    else {
        new_rep = to_representative;
        from_representative->representative = new_rep;
    }
    new_rep->number_of_vertices = vert_sum;
    new_rep->number_of_edges = edge_sum;
    hash_table->number_of_components--;
  }
}
```

```
//
// generates a list of similar words and calls the function add_edge
for each one (done)
//
// man utf8 for details on the uft8 encoding
//

static void break_utf8_string(const char *word,int
*individual_characters)
{
  int byte0,byte1;

  while(*word != '\0')
  {
    byte0 = (int)(*(word++)) & 0xFF;
    if(byte0 < 0x80)
      *(individual_characters++) = byte0; // plain ASCII character
    else
    {
      byte1 = (int)(*(word++)) & 0xFF;
      if((byte0 & 0b11100000) != 0b11000000 || (byte1 & 0b11000000) !=
0b10000000)
      {
        fprintf(stderr,"break_utf8_string: unexpected UFT-8
character\n");
        exit(1);
      }
      *(individual_characters++) = ((byte0 & 0b00011111) << 6) |
(byte1 & 0b00111111); // utf8 -> unicode
    }
  }
  *individual_characters = 0; // mark the end!
}

static void make_utf8_string(const int *individual_characters,char
word[_max_word_size_])
{
  int code;

  while(*individual_characters != 0)
  {
    code = *(individual_characters++);
    if(code < 0x80)
      *(word++) = (char)code;
    else if(code < (1 << 11))
    { // unicode -> utf8
      *(word++) = 0b11000000 | (code >> 6);
      *(word++) = 0b10000000 | (code & 0b00111111);
    }
    else
    {
```

```
      fprintf(stderr,"make_utf8_string: unexpected UFT-8
character\n");
      exit(1);
    }
  }
  *word = '\0';  // mark the end
}

static void similar_words(hash_table_t *hash_table,hash_table_node_t
*from)
{
  static const int valid_characters[] =
  { // unicode!
    0x2D,
// -
    0x41,0x42,0x43,0x44,0x45,0x46,0x47,0x48,0x49,0x4A,0x4B,0x4C,0x4D,
// A B C D E F G H I J K L M
    0x4E,0x4F,0x50,0x51,0x52,0x53,0x54,0x55,0x56,0x57,0x58,0x59,0x5A,
// N O P Q R S T U V W X Y Z
    0x61,0x62,0x63,0x64,0x65,0x66,0x67,0x68,0x69,0x6A,0x6B,0x6C,0x6D,
// a b c d e f g h i j k l m
    0x6E,0x6F,0x70,0x71,0x72,0x73,0x74,0x75,0x76,0x77,0x78,0x79,0x7A,
// n o p q r s t u v w x y z
    0xC1,0xC2,0xC9,0xCD,0xD3,0xDA,
// Á Â É Í Ó Ú

0xE0,0xE1,0xE2,0xE3,0xE7,0xE8,0xE9,0xEA,0xED,0xEE,0xF3,0xF4,0xF5,0xFA,
0xFC, // à á â ã ç è é ê í î ó ô õ ú ü
    0
  };
  int i,j,k,individual_characters[_max_word_size_];
  char new_word[2 * _max_word_size_];

  break_utf8_string(from->word,individual_characters);
  for(i = 0;individual_characters[i] != 0;i++)
  {
    k = individual_characters[i];
    for(j = 0;valid_characters[j] != 0;j++)
    {
      individual_characters[i] = valid_characters[j];
      make_utf8_string(individual_characters,new_word);
      // avoid duplicate cases
      if(strcmp(new_word,from->word) > 0)
        add_edge(hash_table,from,new_word);
    }
    individual_characters[i] = k;
  }
}


//
// breadth-first search (to be done)
//
```

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

static unsigned int breadh_first_search(unsigned int
maximum_number_of_vertices,hash_table_node_t
**list_of_vertices,hash_table_node_t *origin,hash_table_node_t *goal)
{
      unsigned int           num_visited;
      hash_table_node_t *node;
      adjacency_node_t  *neighbour;
      deque_t           *deque;

      deque = create_deque(maximum_number_of_vertices);

      num_visited = 0;
      put_hi(deque, origin);
      while (deque->size > 0 && num_visited <
maximum_number_of_vertices)
      {
            node = get_low(deque);
            node->visited = 1;
            if (list_of_vertices)
                  list_of_vertices[num_visited] = node;
            num_visited++;
            if (node == goal)
                  break;
            for(neighbour = node->head; neighbour ; neighbour =
neighbour->next)
            {
                  if (!neighbour->vertex->visited)
                  {
                        neighbour->vertex->visited = 1;
                        neighbour->vertex->previous = node;
                        put_hi(deque, neighbour->vertex);
                  }
            }
      }
      delete_deque(deque);
      return num_visited;
}

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

//
```

```
// 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 *origin, *representative;
      hash_table_node_t **list_of_vertices;
      unsigned int            list_length, i;

      origin = find_word(hash_table, word, 0);
      if (!origin)
      {
            printf("\nThe word: %s doesn't exist\n", word);
            return;
      }

      mark_all_vertices(hash_table);
      representative = find_representative(origin);
      list_of_vertices = malloc(representative->number_of_vertices *
sizeof(hash_table_node_t *));
  if (!list_of_vertices) {
    fprintf(stderr, "list_connected_component: out of memory\n");
    exit(1);
  }
      list_length = breadh_first_search(representative-
>number_of_vertices, list_of_vertices, origin, NULL);
      for (i=0; i < list_length; i++){
    printf("%s\n", list_of_vertices[i]->word);
  }
      free(list_of_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)
//

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) {
        fprintf(stderr, "\nWord not found: %s\n", from_word);
        return;
    }
    if (!to) {
        fprintf(stderr, "\nWord not found: %s\n", to_word);
        return;
    }

    mark_all_vertices(hash_table);
    size_t list_len = breadh_first_search(find_representative(to)-
>number_of_vertices, NULL, to, from);
    if (list_len == 0) {
        fprintf(stderr, "Words are not connected\n");
    } else {
        size_t i = 0;
        while (from && from != to) {
            printf("  [%zu] %s\n", i++, from->word);
            from = from->previous;
        }
        printf("  [%zu] %s\n", i++, from->word);
    }
}



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

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


//
// 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            (list hash table stats)\n");
    fprintf(stderr,"  4            (list graph info)\n");
    fprintf(stderr,"  0            (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 == 0)
      break;
    else if(command == 3)
                  hash_table_stats(hash_table);
           else if(command == 4)
                  graph_info(hash_table);
  }
  // clean up
  hash_table_free(hash_table);
  return 0;
}
```