Projeto 2

Word Ladder

Docentes:

Tomás Oliveira e Silva

João Manuel Rodrigues

Trabalho realizado por:

Vasco Faria, nº 107323, 40%

Miguel Pinto, nº 107449, 60%

Índice

1 – Introdução

Neste trabalho, foi proposta a resolução de um problema que pode ser interpretado como a construção de uma *Word Ladder*, que consiste numa sequência de palavras, tal que 2 elementos consecutivos só podem diferir por uma letra.

Dado um conjunto de palavras, o programa irá ligar aquelas que diferem em apenas 1 letra, sendo criando um grafo de um ou mais componentes conexos que permitirá operações, como:

- Busca de palavras dentro do mesmo componente;
- Indicação do caminho mais curto entre 2 palavras;
- Dados de um determinado componente (ex.: diâmetro);

Para tal problema, já nos foram fornecidos alguns ficheiros de texto que servirão para o teste do programa e também um caminho mais curto entre “bem” e “mal”.

Ao longo do documento serão explicadas e analisadas todas as funções desenvolvidas por nós com vista na resolução do problema.

2 – Funções implementadas

Nesta secção do documento são apresentadas e explicadas as funções que foram implementadas no programa.

2.1 – hash_table_create()

```c
static hash_table_t *hash_table_create(void)
{
  hash_table_t *hash_table;
  unsigned int i, size=1024;

//Allocate memory for the hash_table
  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);
  }

//Modify some of the hash_table values
  hash_table->hash_table_size = size;
  hash_table->number_of_entries = 0;
  hash_table->number_of_edges = 0;

//Allocate memory for the heads array
  hash_table->heads = (hash_table_node_t **)malloc(size * sizeof(hash_table_node_t*));
  if (hash_table->heads == NULL)
  {
    free(hash_table);
    fprintf(stderr,"create_hash_table->heads: out of memory\n");
    exit(1);
  }

//Initialize the linked list heads to NULL
  for (i = 0; i < size; i++)
    hash_table->heads[i] = NULL;

  return hash_table;
}
```

Figura 1 - Função hash_table_create()

O objetivo da função *hash_table_create()* é criar a *hash_table* que nos ajudará para guardar informação para o nosso *word_ladder*.

Inicialmente, a função cria um ponteiro do tipo *hash_table_t* e declara o tamanho inicial da *hash_table* (seria mais correto este valor ser dado como argumento ao chamar a função, mas preferimos não alterar o cabeçalho da mesma).

Aloca espaço para os dados da *hash_table* na memória com a função *malloc()*, verificando se é possível ser criado esse espaço, caso contrário é imprida uma mensagem de erro. Atribui valores a alguns atributos da *hash_table* como o seu tamanho e o número de entradas.

De seguida, criamos novamente espaço na memória, mas desta vez para as cabeças de listas das palavras conectadas. Caso não seja possível, é chamada a função *free()*, que desaloca o bloco de espaço na memória, anteriormente criado para a *hash_table*.

Por fim, as cabeças de listas são inicializadas na *hash_table* com "NULL" e a *hash_table* é retornada.

2.2 – hash_table_grow()

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

  // Determine the new size of the hash table
  new_size = (unsigned int)(hash_table->hash_table_size / _hash_table_load_factor_);

  // Allocate memory for the new heads array
  hash_table->heads = (hash_table_node_t **)realloc(hash_table->heads,new_size*sizeof(hash_table_node_t*));
  if (hash_table->heads == NULL)
  {
    free(hash_table);
    fprintf(stderr,"create_hash_table->heads: out of memory\n");
    exit(1);
  }

  // Initialize the linked list heads to NULL
  for (i = 0; i < new_size; i++)
    hash_table->heads[i] = NULL;
```

Figura 2 - Função hash_table_grow() (1 de 2)

```c
  // Rehash the entries in the old heads array and insert them into the new heads array
  for (i = 0; i < hash_table->hash_table_size; i++)
  {
    // Rehash each entry in the linked list and insert it into the new heads array
    node = hash_table->heads[i];
    while (node != NULL)
    {
      temp = node;
      node = node->next;

      // Rehash the entry and insert it into the new heads array
      unsigned int new_index = crc32(temp->word) % new_size;
      temp->next = hash_table->heads[new_index];
      hash_table->heads[new_index] = temp;
    }
  }

  // Update the hash table size
  hash_table->hash_table_size = new_size;
}
```

Figura 3 - Função hash_table_grow() (2 de 2)

Esta função tem como objetivo expandir o espaço da *hash_table* criada na função anterior, de modo a ser possível minimizar o número de colisões e o tamanho de cada *linked_list* gerada, aumentando a eficiência na procura de uma palavra.

Dentro da função é declarado o novo tamanho que será dado ao espaço da memória para as cabeças de listas da *hash_table,* usando *load_factor* (= 0.5) para aumentar o tamanho para o dobro.

Com isto é realocado o espaço na memória das cabeças de listas com a função *realloc()*. Caso esta ação não seja possivel, é libertado o espaço na memória.

Tal como na função *hash_table_create()*, as cabeças de listas são inicializadas com *NULL*.

De seguida irá refazer as entradas do antigo array de cabeças de listas e inseri-las-á no novo array de cabeças de listas. Para isto, é percorrida a *hash_table* e por cada entrada, a informação contida nas *linked_lists* é transferida para o novo array de cabeças de listas.

Por fim o tamanho da *hash_table* é atualizado.

2.3 – hash_table_free()

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

  // Free the memory for each linked list in the hash table
  for (i = 0; i < hash_table->hash_table_size; i++)
  {
    // Free the memory for each node in the linked list
    node = hash_table->heads[i];
    while (node != NULL)
    {
      temp = node;
      free_hash_table_node(temp);
      node = node->next;
    }
  }

  // Free the memory for the heads array and the hash table structure
  free(hash_table->heads);
  free(hash_table);
}
```

Figura 4 - Função hash_table_delete()

Esta função tem como objetivo fazer com que seja libertado o espaço ocupado pela estrututra da *hash_table* na memória.

Para isso, inicialmente em cada lista encadeada, libertamos o espaço alocado de cada nó. Posteriomente libertamos o espaço alocado das cabeças de listas e por fim libertamos o espaço da estrutura da hash table.

Tudo isto, foi implementado usando a função *free()* que liberta o espaço da memória do argumento em questão.

2.4 – find_word()

```
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];

  //Find the word given
  while(node != NULL){
    if (strcmp(word,node->word) == 0)
      return node;
    node = node->next;
  }

  //Insert if the word was not found
  if (insert_if_not_found)
  {
    node = allocate_hash_table_node();
    strncpy(node->word,word,_max_word_size_);
    node->next = hash_table->heads[i];
    hash_table->heads[i] = node;
    hash_table->number_of_entries++;
    node->number_of_edges = 1;
    node->number_of_vertices = 1;
    node->representative = node;
  }

  return node;
}
```

Figura 5 - Função find_word()

O objetivo desta função é encontrar onde está guardada uma determinada palavra na *hash_table* com a possibilidade de a inserir se não a encontrar.

Inicialmente, a variável *i* fica com o valor de *hash* da palavra que entra como argumento na função e cria-se um nó que vai percorre a *linked_list* nessa mesma posição.

De seguida compara-se cada node que existir nessa *linked_list*, sendo retornado caso exista.

Se não for encontrada a palavra pretendida, o valor do nó fica *NULL*, embora possa ser pedido para inserir com o último parâmetro da função.

Nesse caso recorre-se às funções *allocate_hash_node()* e *strncpy()*, de forma a criar e copiar a palavra para a posição *i* da *hash_table*, sendo inserido no início da *linked_list*.

Por fim, são atualizados atributos da *hash_table* e do próprio *node*.

2.5 – find_representative()

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

  representative = node;
  next_node = node->representative;

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

  return representative;
}
```

Figura 6 - Função find_representative()

O objetivo da função *find_representative()* é encontrar o nó representante de um determinado componente conexo (cada nó começa por ser o seu próprio representante).

Inicialmente, a função trata o representante como o nó atual e o próximo nó como sendo o representante deste.

De seguida, entra-se num ciclo até que os 2 ponteiros anteriores apontem para o mesmo nó, mantendo a lógica da frase anterior.

Por fim, o representante é retornado.

2.6 – add_edge()

```c
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_from, *link_to;

  to = find_word(hash_table,word,0);
  // Check if the destination word exists
  if (to == NULL) return;

  // Check if the edge already exists
  for (link_from = from->head; link_from; link_from = link_from->next)
  {
    if (link_from->vertex == to) return;
  }

  // Create adjacency node
  link_from = allocate_adjacency_node();
  link_to = allocate_adjacency_node();

  //Link both words
  link_from->vertex = from;
  link_to->vertex = to;
  link_from->next = to->head;
  link_to->next = from->head;
  to->head = link_from;
  from->head = link_to;

  // Increment the number of edges in the graph
  hash_table->number_of_edges+=2;
```

Figura 7 - Função add_edge() (1 de 2)

```c
  // Find representatives of the connected components
  from_representative = find_representative(from);
  to_representative = find_representative(to);

  // Check if the 'from' and 'to' vertices belong to different connected components
  if (from_representative != to_representative)
  {
    // Merge the smaller connected component into the bigger one
    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;
    }
  }
}
```

Figura 8 - - Função add_edge() (2 de 2)

O objetivo da função *add_edge()* é criar ligações as palavras semelhantes, gerando assim vários componentes conexos.

Inicialmente, a função procura onde a palavra introduzida se encontra dentro da *hash_table*, se não a encontrar, irá retornar.

De seguida, verifica se a ligação entre as 2 palavras já existe e retorna se for o caso.

Está-se nas condições necessárias para criar a ligação, então a função aloca 2 nós de adjacência (para fazer 2 ligações entre cada para de palavras semelhantes), liga as 2 palavras e incrementa o número de ligações da *hash_table*.

Agora é necessário verificar se as 2 palavras têm o mesmo representante, pois poderá ser necessário unir os componentes conexos se este não for o mesmo.

Por fim, procede-se à união dos componentes conexos (quando necessário), optando por deixar o maior absorver o menor dos dois, incrementando os atributos necessários (número de palavras e ligações).

2.7 – breadth_first_search()

```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)
{
  hash_table_node_t *node;
  adjacency_node_t *neighbor;
  int current = 0, end = 0; // Head and tail of the list_of_vertices
  list_of_vertices[0] = origin;
  origin->visited = 1;

  while (current <= end)
  {
    // Get the current vertex and move to the next for the following iteration
    node = list_of_vertices[current++];
    //printf("passei aqui %d vezes\n",current);

    // Check if we have reached the goal
    if (node == goal)  break;

    // Mark the current vertex as visited
    node->visited = 1;

    // Visit the neighbors of the current vertex
    for (neighbor = node->head; neighbor; neighbor = neighbor->next)
    {
      if (!neighbor->vertex->visited)
      {
        // Add the unvisited neighbor to the list of vertices
        list_of_vertices[++end] = neighbor->vertex;
        neighbor->vertex->visited = 1;
        neighbor->vertex->previous = node;
      }
    }
  }
```

Figura 9 - Função breadth_first_search() (1 de 2)

```
  // Reset visited vertices for future searches
  for (int i = 0; i < maximum_number_of_vertices; i++)  list_of_vertices[i]->visited = 0;

  if (node == goal)
  {
    return end + 1; // Return the number of visited vertices
  }
  else
  {
    return -1; // Return -1 if the goal was not reached
  }
}
```

Figura 10 - Função breadth_first_search() (2 de 2)

O objetivo da função *breadth_first_search()* é fazer pesquisas entre as palavras que foram ligadas, dando oportunidade de listar um componente conexo, encontrar um caminho mais curto (explorado nas 2 funções abaixo) entre outros que não foram implementados.

Inicialmente, a função adiciona a palavra dada ao início da *list_of_vertices* e marca-a como visitada.

De seguida, entra num ciclo onde procura as ligações que a palavra introduzida tem e, se estas não tiverem sido visitadas, adiciona-as à lista, marca-as como visitadas e adiciona a palavra atual como a anterior na pesquisa.

O processo é repetido até chegar à palavra destino ou quando o número máximo de vértices tiver sido adicionado à lista (deve corresponder ao tamanho do componente conexo).

No final da busca, marca-se novamente todos os vértices como não visitados (para buscas futuras) e decide-se o valor a retornar.

Se tiver sido alcançado a palavra pretendida, é devolvido o número de vértices percorridos, caso contrário, devolve-se -1, o que infelizmente aconteceu sempre nos nossos testes.

2.8 – list_connected_component()

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

  //Verify that the word exists
  if (node == NULL)
  {
    fprintf(stderr,"list_connected_component: word %s does not exist\n",word);
    return;
  }

  //Get the number_of_vertices of the connected component
  hash_table_node_t *representative = find_representative(node);
  int size = representative->number_of_vertices;

  //Search all words belonging to the connected component
  hash_table_node_t **list_of_vertices = malloc(size * sizeof(hash_table_node_t*));
  int number_of_vertices = breadth_first_search(size, list_of_vertices, node, NULL);

  //List connected component
  for (int i = 0; i < number_of_vertices; i++)
    printf("[ %d] %s\n",i,list_of_vertices[i]->word);

  free(list_of_vertices);
}
```

Figura 11 - Função list_connected_component()

O objetivo da função *list_connected_component()* é listar todas as palavras pertencentes ao mesmo componente conexo.

Inicialmente, a função procura onde a palavra introduzida se encontra dentro da *hash_table*, se não a encontrar, irá retornar com uma mensagem de erro.

Com a localização da palavra, procura o seu representante, de forma a saber o tamanho do componente conexo (número que indica quantas palavras terá de listar).

Aloca espaço numa lista para o número de vértices que encontrou e faz a procura, recorrendo à função *breadth_first_search()* com o parâmetro '*goal'* com o valor *NULL*, para que todo o componente seja percorrido.

Por fim, percorre a lista criada para mostrar as palavras pertencentes ao componente conexo e liberta a memória alocada para a lista criada anteriormente.

2.9 – path_finder()

```c
static void path_finder(hash_table_t *hash_table,const char *from_word,const char *to_word)
{
  hash_table_node_t *from, *to, *node, *from_representative, *to_representative;

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

  //Verify that both words exist
  if (from == NULL)
  {
    fprintf(stderr,"path_finder: word %s does not exist\n",from_word);
    return;
  }
  if (to == NULL)
  {
    fprintf(stderr,"path_finder: word %s does not exist\n",to_word);
    return;
  }

  // Verify both words belong to the same connected component
  from_representative = find_representative(from);
  to_representative = find_representative(to);
  if (to_representative != from_representative)
  {
    fprintf(stderr,"path_finder: No path was found from %s to %s\n", from_word, to_word);
    return;
  }
```

Figura 12 - Função path_finder() (1 de 2)

```c
//Get the size of the connected component and search a path between both words
int size = from_representative->number_of_vertices;
hash_table_node_t **list_of_vertices = malloc(size * sizeof(hash_table_node_t*));
int visited_vertices = breadth_first_search(size, list_of_vertices, to, from);

//List the path found
if (visited_vertices > 0)
{
  node = list_of_vertices[0];
  printf("Shortest path from %s to %s:\n", to_word, from_word);
  printf("  %s\n", to_word);
  for (int i = 1; i < visited_vertices; i++)
  {
    if (node == list_of_vertices[i]->previous)
    {
      printf("  %s\n", list_of_vertices[i]->word);
      node = list_of_vertices[i];
    }
  }
}
else  fprintf(stderr,"path_finder: No path was found from %s to %s\n", from_word, to_word);

free(list_of_vertices);
```

Figura 13 - Função path_finder() (2 de 2)

O objetivo da função *path_finder()* é listar todas as palavras percorridas num caminho entre 2 palavras fornecidas.

Inicialmente, a função procura se as 2 palavras introduzidas se encontram dentro da *hash_table*, se não encontrar pelo menos 1 delas, irá retornar com uma mensagem de erro.

Com as localizações das palavras, procura os seus representantes e verifica que são o mesmo (pois é a única maneira de existir um caminho) e retorna com uma mensagem de erro caso não sejam.

Descobre o tamanho do componente conexo, alocando espaço numa lista para o número de vértices que encontrou e faz a procura, recorrendo à função *breadth_first_search()* com ambas as palavras.

Por fim, percorre a lista criada para mostrar as palavras pertencentes ao caminho encontrado, movendo-se pelo ponteiro anterior criado na função *breadth_first_search()* e liberta a memória alocada essa mesma lista.

2.10 – graph_info()

```c
static void graph_info(hash_table_t *hash_table)
{
   fprintf(stderr,"\nHash table data:\nSize: %u \nWords inserted: %u\nEdges created: %u\n",
               hash_table->hash_table_size,
               hash_table->number_of_entries,
               hash_table->number_of_edges);
}
```

O objetivo da função *graph_info()* é mostrar alguma informação sobre a *hash_table* criada ao correr o programa.

É mostrada a seguinte informação:

- Tamanho da *hash_table*
- Número de palavras adicionadas
- Total de ligações criadas

3 – Testes

Já nos tinha sido fornecida uma função *main()* que, tirando proveito das várias funções do documento, lê um ficheiro de texto, insere todas as palavras do mesmo na *hash*_table, incentiva à ligação das que são semelhantes, apresenta um menu para o utilizador escolher o que pretende ver e no final do programa limpa a memória.

De maneira a testar mais facilmente, foi criado um ficheiro de texto adicional (‘teste.txt’) apenas com as seguintes palavras: “bem, tem, teu, meu, mau, mal, ola, xau e fui”.

Entre as opções que era possível desenvolver (encontrar um caminho mais curto e listar um componente conexo), apenas tivemos sucesso na segunda e exclusivamente com este ficheiro mais pequeno, sendo que todos os outros testar originavam *segmentation fault* ou algum tipo de falha na procura dentro da função *breadth_first_search()* para o qual não foi encontrada solução após várias tentativas.

```
Hash table data:
Size: 1024
Words inserted: 9
Edges created: 12

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 mal to bem:
  mal
  mau
  meu
  teu
  tem
  bem
```

Figura 14 - Teste com ficheiro 'teste.txt'

Dado a obtenção de erros em todos os outros testes, não consideramos oportuno colocá-los no documento, embora tenham sido feitos testes com os ficheiros fornecidos.

4 – Conclusão

Após a conclusão do relatório, pode-se afirmar que não se conseguiram completar todos os objetivos pretendidos, foi construída a estrutura da *hash_table*, inseridas as palavras contidas nos ficheiros e criadas as suas ligações, mas não foram implementadas com sucesso todas as funções propostas, nomeadamente que mostrassem dados em concreto da *word_ladder*.

No entanto, foram aprimorados os conhecimentos sobre o funcionamento da linguagem C e de vários assuntos lecionados ao longo da disciplina de Algoritmos e Estruturas de Dados, nomeadamente *hash_tables*.

Pode-se também afirmar que, com o decorrer do trabalho, enfrentámos vários problemas, o que criou bastante dinâmica entre o grupo e contribuiu para o desenvolvimento do espírito de equipa.

5 – Webgrafia

Para a realização deste trabalho forma consultados, para além do documento PDF da Unidade Curricular:

- DigitalOcean
- StackOverFlow
- GeeksForGeeks

6 – Apêndice

Abaixo está apresentado todo o código C desenvolvido:

```
//
// AED, November 2022 (Tomás Oliveira e Silva)
//
// Second practical assignement (work ladder)
//
// Place your student numbers and names here
//   N.Mec. 107323  Name: 107449
//
// 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
//      *) breadth_first_search
//   4) RECOMMENDED: list all words belonging to a connected component
//      *) breadth_first_search
//      *) list_connected_component
//   5) RECOMMENDED: find the shortest path between two words
//      *) breadth_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
//      *) breadth_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
#define _hash_table_load_factor_  0.5

//
// 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 the next 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, size=1024;
```

```
//Allocate memory for the hash_table
  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);
  }

//Modify some of the hash_table values
  hash_table->hash_table_size = size;
  hash_table->number_of_entries = 0;
  hash_table->number_of_edges = 0;

//Allocate memory for the heads array
  hash_table->heads = (hash_table_node_t **)malloc(size * sizeof(hash_table_node_t*));
  if (hash_table->heads == NULL)
  {
    free(hash_table);
    fprintf(stderr,"create_hash_table->heads: out of memory\n");
    exit(1);
  }

//Initialize the linked list heads to NULL
  for (i = 0; i < size; i++)
    hash_table->heads[i] = NULL;

  return hash_table;
}

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

  // Determine the new size of the hash table
  new_size = (unsigned int)(hash_table->hash_table_size / _hash_table_load_factor_);

  // Allocate memory for the new heads array
  hash_table->heads = (hash_table_node_t **)realloc(hash_table-
>heads,new_size*sizeof(hash_table_node_t*));
  if (hash_table->heads == NULL)
  {
    free(hash_table);
    fprintf(stderr,"create_hash_table->heads: out of memory\n");
    exit(1);
  }

  // Initialize the linked list heads to NULL
  for (i = 0; i < new_size; i++)
    hash_table->heads[i] = NULL;

  // Rehash the entries in the old heads array and insert them into the new heads array
  for (i = 0; i < hash_table->hash_table_size; i++)
  {
    // Rehash each entry in the linked list and insert it into the new heads array
    node = hash_table->heads[i];
    while (node != NULL)
    {
      temp = node;
      node = node->next;

      // Rehash the entry and insert it into the new heads array
      unsigned int new_index = crc32(temp->word) % new_size;
      temp->next = hash_table->heads[new_index];
      hash_table->heads[new_index] = temp;
    }
  }
```

```
  // Update the hash table size
  hash_table->hash_table_size = new_size;
}

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

  // Free the memory for each linked list in the hash table
  for (i = 0; i < hash_table->hash_table_size; i++)
  {
    // Free the memory for each node in the linked list
    node = hash_table->heads[i];
    while (node != NULL)
    {
      temp = node;
      free_hash_table_node(temp);
      node = node->next;
    }
  }

  // Free the memory for the heads array and the hash table structure
  free(hash_table->heads);
  free(hash_table);
}

static hash_table_node_t *find_word(hash_table_t *hash_table,const char *word,int
insert_if_not_found)
{
  hash_table_node_t *node;
  unsigned int i;

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

  node = hash_table->heads[i];

  //Find the word given
  while(node != NULL){
    if (strcmp(word,node->word) == 0)
      return node;
    node = node->next;
  }

  //Insert if the word was not found
  if (insert_if_not_found)
  {
    node = allocate_hash_table_node();
    strncpy(node->word,word,_max_word_size_);
    node->next = hash_table->heads[i];
    hash_table->heads[i] = node;
    hash_table->number_of_entries++;
    node->number_of_edges = 1;
    node->number_of_vertices = 1;
    node->representative = node;
  }

  return node;
}

//
// add edges to the word ladder graph (Path compression was not done)
//

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

  representative = node;
```

```
  next_node = node->representative;

  while (representative != next_node)
  {
    representative = next_node;
    next_node = next_node->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_from, *link_to;

  to = find_word(hash_table,word,0);
  // Check if the destination word exists
  if (to == NULL) return;

  // Check if the edge already exists
  for (link_from = from->head; link_from; link_from = link_from->next)
  {
    if (link_from->vertex == to) return;
  }

  // Create adjacency node
  link_from = allocate_adjacency_node();
  link_to = allocate_adjacency_node();

  //Link both words
  link_from->vertex = from;
  link_to->vertex = to;
  link_from->next = to->head;
  link_to->next = from->head;
  to->head = link_from;
  from->head = link_to;

  // Increment the number of edges in the graph
  hash_table->number_of_edges+=2;

  // Find representatives of the connected components
  from_representative = find_representative(from);
  to_representative = find_representative(to);

  // Check if the 'from' and 'to' vertices belong to different connected components
  if (from_representative != to_representative)
  {
    // Merge the smaller connected component into the bigger one
    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;
    }
  }
}


//
// 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 (Has an error that we couldn't solve)
//
// returns the number of vertices visited; if the last one is goal, following the previous links
gives the shortest path between goal and origin
//

static int 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)
{
  hash_table_node_t *node;
  adjacency_node_t *neighbor;
  int current = 0, end = 0; // Head and tail of the list_of_vertices
  list_of_vertices[0] = origin;
  origin->visited = 1;

  while (current <= end)
  {
    // Get the current vertex and move to the next for the following iteration
    node = list_of_vertices[current++];
    //printf("passei aqui %d vezes\n",current);

    // Check if we have reached the goal
    if (node == goal)  break;

    // Mark the current vertex as visited
    node->visited = 1;

    // Visit the neighbors of the current vertex
    for (neighbor = node->head; neighbor; neighbor = neighbor->next)
    {
      if (!neighbor->vertex->visited)
      {
        // Add the unvisited neighbor to the list of vertices
        list_of_vertices[++end] = neighbor->vertex;
        neighbor->vertex->visited = 1;
        neighbor->vertex->previous = node;
      }
    }
  }

  // Reset visited vertices for future searches
  for (int i = 0; i < maximum_number_of_vertices; i++)  list_of_vertices[i]->visited = 0;

  if (node == goal)
  {
    return end + 1; // Return the number of visited vertices
  }
  else
  {
    return -1; // Return -1 if the goal was not reached
```

```
  }
}



//
// list all vertices belonging to a connected component (Not working due to bfs error somewhere)
//

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

  //Verify that the word exists
  if (node == NULL)
  {
    fprintf(stderr,"list_connected_component: word %s does not exist\n",word);
    return;
  }

  //Get the size of the connected component
  hash_table_node_t *representative = find_representative(node);
  int size = representative->number_of_vertices;

  //Search all words belonging to the connected component
  hash_table_node_t **list_of_vertices = malloc(size * sizeof(hash_table_node_t*));
  int number_of_vertices = breadth_first_search(size, list_of_vertices, node, NULL);

  //List connected component
  for (int i = 0; i < number_of_vertices; i++)
    printf("[ %d] %s\n",i,list_of_vertices[i]->word);

  free(list_of_vertices);
}


//
// find the shortest path from a given word to another given word (Only works with smaller text
file)
//

static void path_finder(hash_table_t *hash_table,const char *from_word,const char *to_word)
{
  hash_table_node_t *from, *to, *node, *from_representative, *to_representative;

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

  //Verify that both words exist
  if (from == NULL)
  {
    fprintf(stderr,"path_finder: word %s does not exist\n",from_word);
    return;
  }
  if (to == NULL)
  {
    fprintf(stderr,"path_finder: word %s does not exist\n",to_word);
    return;
  }

  // Verify both words belong to the same connected component
  from_representative = find_representative(from);
  to_representative = find_representative(to);
  if (to_representative != from_representative)
  {
    fprintf(stderr,"path_finder: No path was found from %s to %s\n", from_word, to_word);
    return;
  }
```

```
  //Get the size of the connected component and search a path between both words
  int size = from_representative->number_of_vertices;
  hash_table_node_t **list_of_vertices = malloc(size * sizeof(hash_table_node_t*));
  int visited_vertices = breadth_first_search(size, list_of_vertices, to, from);

  //List the path found
  if (visited_vertices > 0)
  {
    node = list_of_vertices[0];
    printf("Shortest path from %s to %s:\n", to_word, from_word);
    printf("  %s\n", to_word);
    for (int i = 1; i < visited_vertices; i++)
    {
      if (node == list_of_vertices[i]->previous)
      {
        printf("  %s\n", list_of_vertices[i]->word);
        node = list_of_vertices[i];
      }
    }
  }
  else  fprintf(stderr,"path_finder: No path was found from %s to %s\n", from_word, to_word);

  free(list_of_vertices);
}


//
// some graph information
//

static void graph_info(hash_table_t *hash_table)
{
   fprintf(stderr,"\nHash table data:\nSize: %u \nWords inserted: %u\nEdges created: %u\n",
            hash_table->hash_table_size,
            hash_table->number_of_entries,
            hash_table->number_of_edges);
}


//
// main program
//

int main(int argc,char **argv)
{
  char word[100],from[100],to[100];
  hash_table_t *hash_table;
  hash_table_node_t *node;
  unsigned int i;
  int command;
  FILE *fp;

  // initialize hash table
  hash_table = hash_table_create();
  // read words
  fp = fopen((argc < 2) ? "wordlist-big-latest.txt" : argv[1],"rb");
  if(fp == NULL)
  {
    fprintf(stderr,"main: unable to open the words file\n");
    exit(1);
  }
  while(fscanf(fp,"%99s",word) == 1)
    (void)find_word(hash_table,word,1);
  fclose(fp);

  // find all similar words
  for(i = 0u;i < hash_table->hash_table_size;i++)
    for(node = hash_table->heads[i];node != NULL;node = node->next)
      similar_words(hash_table,node);
```

```
  graph_info(hash_table);
  // ask what to do
  for(;;)
  {
    fprintf(stderr,"\nYour 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;
}
```