AED - Relatório

Universidade de Aveiro(*UA*)

Anderson Lourenço, Sara Almeida, Hugo Correia

Projeto nº2 - Word Ladder

Algoritmos e Estrutura de Dados

DETI - Departamento de Electrónica, Telecomunicações e Informática

Anderson Lourenço, Sara Almeida, Hugo Correia

(108579) aaokilourenco@ua.pt,

(108796) sarafalmeida@ua.pt,

(108215) hf.correia@ua.pt

10/01/2023

Índice

Capítulo 1

Introdução

Word Ladder

No contexto da disciplina de Algoritmos e Estrutura de Dados foi-nos proposto o segundo assignment que se baseia numa "word ladder", ou seja, uma "escada"ou sequência de palavras na qual duas palavras adjacentes diferem apenas por uma letra.

Diferindo um pouco do projeto anterior, neste foi-nos disponibilizado um ficheiro já com algum código base incompleto para que o pudéssemos completar, para além de alguns documentos de texto com listas de palavras portuguesas, para podermos testar o nosso programa.

Assim, o objetivo deste trabalho prático foi criar uma implementação correta e funcional da estrutura de dados hash table que será a utilizada para guardar a respetiva "escada"de palavras. Particularmente, uma hash table que modifica dinamicamente de tamanho quando são adicionadas palavras.

Contextualização do Problema

Como mencionado anteriormente, para este projeto foram-nos disponibilizados ficheiros de texto com listas de palavras portuguesas. O objetivo principal é que o programa que vamos completar, cujas funções teremos de escrever, represente um grafo cujos nós consecutivos são representados por palavras que diferem entre si de apenas uma letra, a partir da construção e manipulação de uma hash table.

Uma vez completo, o programa deve ser capaz de mostrar ao utilizador opções como o caminho mais curto de uma palavra a outra, informação sobre o grafo, informação sobre a hash table e, especialmente, a lista de componentes conexas a que uma palavra pertence.

O desafio a que nos propusemos é, além de completar as funções que criam, aumentam e libertam a hash table, encontram informações estatísticas sobre a hash table e procuram as palavras, completarmos as outras funções recomendadas que nos permitem mostrar as funcionalidades descritas acima ao utilizador.

Capítulo 2

Função hash_table_create

Esta função tem como objetivo de criar a hash table vazia e retorná-la. Primeiramente, aloca memória para a estrutura da hash table e verifica se a mesma foi bem sucedida. De seguida, são inicializadas alguns campos da estrutura da tabela como o tamanho da hash table, o seu número de entradas, ou seja, de linked lists e o número de arestas do grafo com o valor DEFAULT e um ponteiro para um array de heads da lista ligada para cada indice da hash table À posteriori, é alocada memória para este mesmo array e verificada a sucessão do mesmo. Finalmente, é percorrido inicializando cada elemento como NULL, indicando, assim que não há nenhum nó da lista ligada a fazer ligação a esse mesmo indice. Por fim é retornado o endereço da hash table criada.

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

  hash_table = (hash_table_t *)malloc(sizeof(hash_table_t));
  if(hash_table == NULL)
  {
    fprintf(stderr,"create_hash_table: out of memory\n");
    exit(1);
  }
  //
  // complete this
  hash_table->hash_table_size = 3000507;
  hash_table->number_of_entries = 0;
  hash_table->number_of_edges = 0;
  hash_table->heads = (hash_table_node_t **)malloc(hash_table->
hash_table_size * sizeof(hash_table_node_t *));
  if (hash_table->heads == NULL) {
    fprintf(stderr,"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;
}
```

Capítulo 3

Função hash_table_grow

Esta função será chamada sempre que for necessário **aumentar** o tamanho da hash table, para possibilitar uma lista maior de palavras, sempre que o número de entradas na hash table ultrapassa um determinado limite. Resumidamente, aqui é criada uma nova hash table de maior tamanho, que nós definimos como sendo o dobro do tamanho da hash table já existente, que vai receber as heads da mesma e as novas, sendo para tal alocada memória através do comando calloc que define todos os bytes da mesma como 0. Assim, através de um ciclo for que percorre cada entrada da hash table anterior, todas essas entradas serão inseridas na nova hash table sendo calculado um novo valor de hash para cada palavra correspondente. De seguida, adicionamos o nó à nova hash table e assim sucessivamente para os restantes nós na linked list.

Por fim, a função "liberta"as heads da hash table anterior através do comando free e redifine a nova hash table com o novo tamanho e o novo array de heads. Esta função é importante na permanência da eficiência da hasht table, pois quanto mais cheia a mesma estiver maior serão o número de colisões existentes aumentando o tempo de busca por palavras. Ao aumentar esse valor é reduzido diminuindo consecutivamente o tempo de busca.

```
static void hash_table_grow(hash_table_t *hash_table)
{
  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 *));
  if (new_heads == NULL)
  {
    fprintf(stderr, "hash_table_grow: out of memory\n");
    exit(1);
  }
  for (unsigned int i = 0; i < hash_table->hash_table_size; i++)
  {
    hash_table_node_t *node = hash_table->heads[i];
    while (node != NULL)
    {
      unsigned int hash = crc32(node->word) % new_size;
      node->next = new_heads[hash];
      new_heads[hash] = node;
      node = node->next;
    }
  }
  free(hash_table->heads);
  hash_table->hash_table_size = new_size;
  hash_table->heads = new_heads;
}
```

Capítulo 4

Função hash_table_free

Tal como na função anterior, esta função percorre todas as entradas da hash table com o objetivo de *libertar* todos os nós da mesma libertando.Primeiramente, percorre cada elemento da hash table libertando cada nó adjacente na lista de adjacência daquele nó, através de um *ciclo while* e com a ajuda das funções free_adjacency_node e free_hash_table_node. Depois de libertados todos os nós da hash table, a função continua libertando o array das heads da mesma e, por fim, libertando a própria hash table. Assim, a memória uma vez alocada para esta hash table será totalmente libertada e ficará novamente disponível não causando memory leaks no sistema.

```
1   static void hash_table_free(hash_table_t *hash_table)
2   {
3     for (int i = 0; i < hash_table->hash_table_size; i++)
4     {
5       hash_table_node_t *node = hash_table->heads[i];
6       while (node != NULL)
7       {
8         hash_table_node_t *next_node = node->next;
9         adjacency_node_t *adj_node = node->head;
10        while (adj_node != NULL)
11        {
12          adjacency_node_t *next_adj_node = adj_node->next;
13          free_adjacency_node(adj_node);
14          adj_node = next_adj_node;
15        }
16
17        free_hash_table_node(node);
18        node = next_node;
19      }
20    }
21    free(hash_table->heads);
22    free(hash_table);
23  }
```

Capítulo 5

Função find_word

A função find_word é usada, como o próprio nome indica, na **procura de uma palavra** dentro da hash table. Caso a palavra já tenha sido anteriormente inserida na hash, é retornado o nó da mesma contendo a respetiva palavra. Caso contrário, ou seja, se (insert_if_not_found == True), a função insere a palavra na hash table, através da função allocate_hash_table_node() retornando o nó que foi criado no seu armazenamento.Se neste processo a alocação de memória para o nó falhar, a função retorna NULL. O mesmo é aplicado caso a palavra tenha número de carateres superior ao limite estipulado pela variável *_max_word_size_* (32 carateres).

Esta função não só é útil pela sua eficácia na procura em termos de tempo de execução, como também por evitar que a mesma palavra seja inserida na tabela várias vezes.

No funcionamento da função é, inicialmente, calculado o índice da tabela (hashCode) onde a palavra está a ser armazenada através da função crc32. De seguida, através da aplicação do módulo pelo tamanho da hash table, reduz-se o valor do hash para um indice validado perante o tamanho da tabela. De seguida, é percorrida, através de um ciclo While, a lista encadeada de palavras armazenadas na posição da tabela correspondente ao índice previamente calculado. Cada um dos nós tem associado uma palavra diferente. Através de um função strcmp() é comparada a palavra procurada com a palavra associada aquele nó da lista. Em caso afirmativo, o nó é retornado. Se a palavra procurada não for encontrada na lista, a função verifica se a flag insert_if_not_found está definida e se a palavra a ser inserida é menor que o tamanho máximo permitido (max_word_size) é alocado através da função allocate_hash_table_-node, um novo nó da tabela hash. Caso haja algum erro na alocação do node será retornado NULL. Caso o mesmo não se verifique a função prossegue na inicialização da palavra com os valores padrão. É colocada a palavra associada ao respetivo nó e adicionado o novo nó da lista ligada. De seguida são armazenadas informações sobre a componente conexa que a palavra representa. São recolhidos os numeros de vertices e de egdes com valores de 1 e 0 respetivamente. O número de entradas na tabela hash é incrementado em um. Se o número de entradas na tabela hash é maior que 75percentagem do tamanho da tabela, a função hash_table_grow() é chamada para redimensionar a tabela hash para um tamanho maior. Se a palavra não foi encontrada e a flag insert_if_not_found não está definida ou a palavra é maior que o tamanho máximo permitido, a função retorna NULL.

Capítulo 6

Função find_representative

Esta função é usada para encontrar o representante do componente conectado ao qual um determinado vértice pertence. Leva um argumento, um ponteiro para um vértice, e retorna um ponteiro para o vértice representativo do componente conectado.

A função começa inicializando uma variável de ponteiro "representativa"para o vértice de entrada. Em seguida, ele entra em um loop ***while***, que é executado até que a variável representativa esteja apontando para si mesma. Durante cada iteração do loop, a variável representativa é atualizada para apontar para o representante do seu valor atual. Esse processo continua até que a variável representativa aponte para si mesma, o que significa que ela é a representante do componente conectado.

A função também inclui uma técnica de otimização chamada "*path compression*". Essa é uma técnica usada para reduzir o número de ponteiros que precisam ser percorridos ao encontrar o representante de um componente conectado. Depois de encontrar o nó representativo, a função varre o caminho do vértice de entrada até o representante novamente e, para cada nó, atualiza o campo representativo para apontar para o nó representativo que foi encontrado anteriormente. Desta forma, da próxima vez que usarmos esta função para encontrar o representante do mesmo vértice, o processo será mais rápido, pois o caminho desse vértice até o nó representativo é mais curto.

Essa função encontra o representante de forma eficiente, pois utiliza a otimização de "*path compression*", o que reduz o número médio de passos necessários para encontrar o representante de um componente conectado.

Capítulo 7

Função add_edge

Esta função é usada para adicionar uma "*edge*"entre duas palavras no grafo armazenado na **hash table**. Utiliza três argumentos:

1. Pointer to hash table;
2. Pointer para a "*from* word's vertex";
3. String a representar a "*to* word".

A função começa por encontrar as "*representatives*"dos componentes conectados que as "*from*"e "*to*"words pertencem ao utilizar a função "*find_representative*". Se a "*to*"word não for encontrada na hash table ou é a mesma que a "*from*"word, a função retorna sem fazer nada.

Depois verifica se a "*from*"e "*to*"words já estão conectadas no mesmo componente, ao comparar as "*representatives*". Se estiverem, incrementa o número de vértices na componente. Caso não estiverem, a função performa uma "*union operation*"ao fazer a "*representative*"da componente com alguns pontos dos vértices para a "*representative*"da componente com mais vértices. Também atualiza o número de vértices e "*edges*"na mesma componente.

Em seguida, a função cria dois novos nós de adjacência, um para o vértice "*from*"e outro para o vértice "*to*". Esses nós são usados para armazenar as arestas no grafo. Ele também define o campo de vértice de cada nó para apontar para o outro vértice e o próximo campo para apontar para o topo da lista de adjacências atual.

Por fim, a função incrementa o número de arestas para os representantes dos vértices "from"e "to", bem como para a hash table e retorna.

Capítulo 8

Função breadth_first_search

A função breadth_first_search() é um algoritmo de busca que é usado para percorrer um grafo ou uma árvore. Ele vai expandindo todos os nós vizinhos de um nó raiz antes de se espicificar em qualquer um dos ramos. No contexto do problema, a função é util para percorrer os vertices de um grafo entre duas palavras dentro de uma determinada componente conexa.

Inicialmente é dada uma *"palavra origem"* (vertice origem pelo qual a busca deve ser iniciada) e uma *"palavra goal"* (vertice que se pretende encontrar. A busca é realizada com o auxílio de uma estrutura de dados intituladade de queue, onde todos os vertices são armazenados para serem futuramente *visitados*.

Numa primeira iteração adiciona o vértice origem na fila e marca-o como *visitado*. Em seguida, enquanto a fila não estiver vazia, é largado o primeiro vértice da fila e são visitados todos os seus vizinhos, adicionando-os à fila e marcando-os como *visitado*. Este processo é repetido até que seja encontrado o vértice goal ou até que todos os vértices sejam visitados.

Mais abaixo está um exemplo de como este processo funciona sendo que na figura x esta o grafo criado com os diferentes niveis e mais abaixo o gráfico do processo até encontrar a palavra definida como *goal* ou, caso não a encontre, retorne o ultimo elemento do grafo.

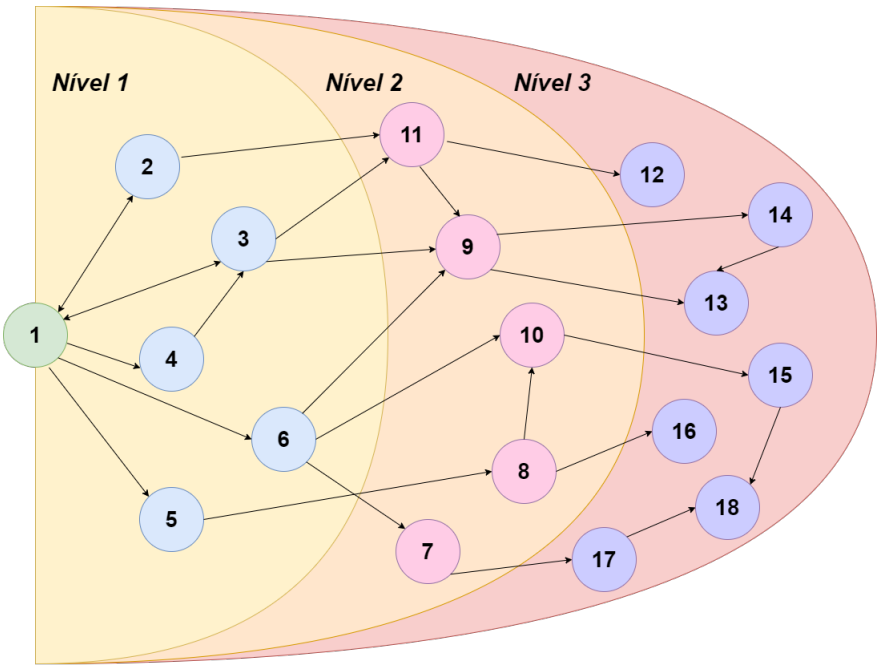

Figura 8.1: Grafo relativo ao Breadth_first_search

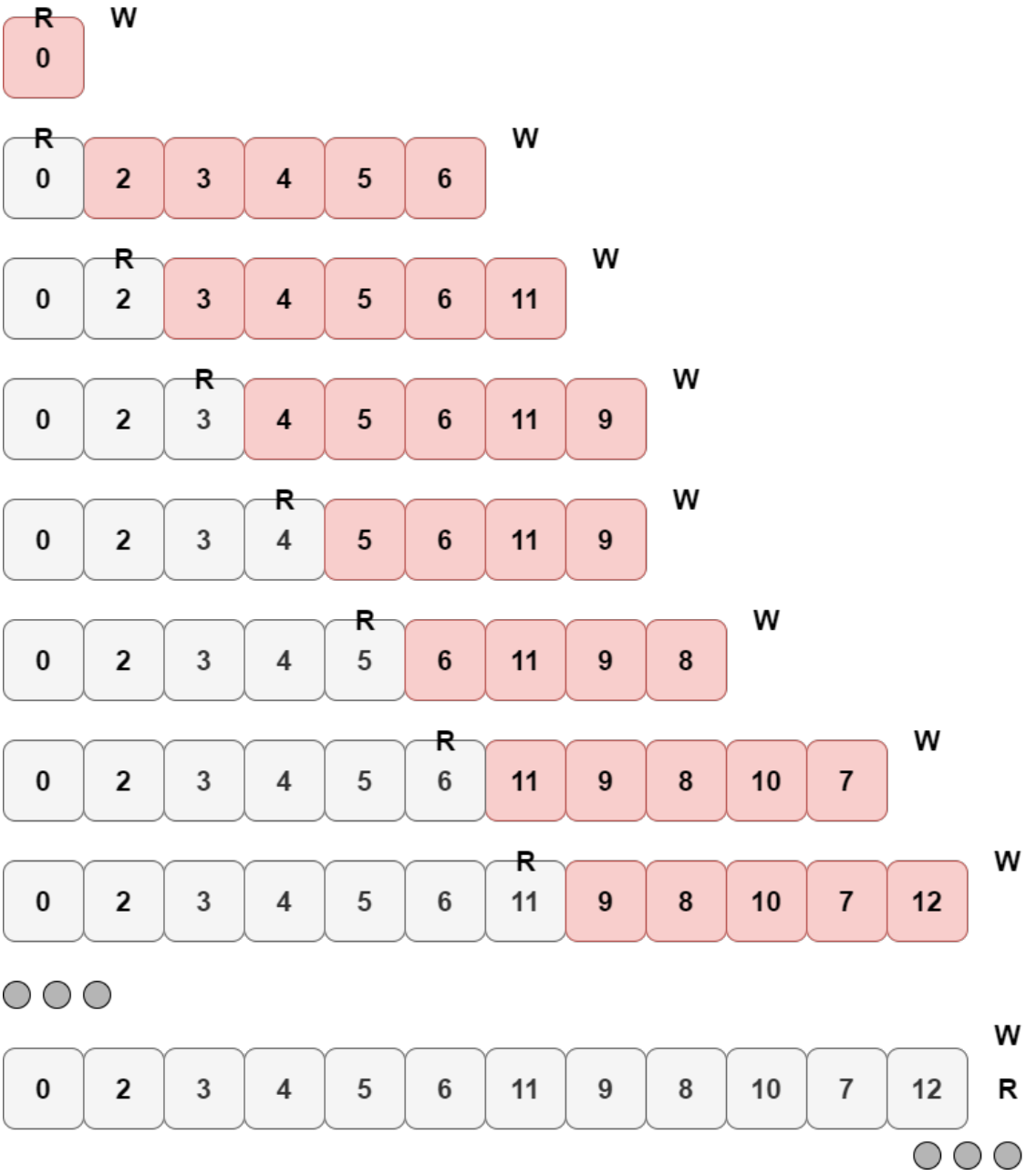

Figura 8.2: Processo até encontrar a palavra

Capítulo 9

Função list_connected_component

Esta função serve para encontrar o nó para a palavra dada. Caso esta não exista na hash table:

```
    if (node == NULL) {
    printf("The word '%s' does not exist in the hash table.\n", word);
    return;
}
```

Procura o nó representativo para a componente conectada que contén o nó dado:

```
    hash_table_node_t *representative = find_representative(node);
    hash_table_node_t **list_of_vertices = malloc(representative->
    number_of_vertices * sizeof(hash_table_node_t *));
    printf("The connected component containing '%s' has %d vertices.\n", word,
    representative->number_of_vertices);

    int number_of_vertices = breadth_first_search(hash_table->number_of_entries
    , list_of_vertices, node, NULL);
```

E por fim, imprime todas as palavras pertencentes à componente conectada:

```
    for (unsigned int i = 0; i < number_of_vertices; i++) {
    printf("%s\n", list_of_vertices[i]->word);
    v++;
}
    printf("Number of vertices: %d\n", v);
    printf("Number of edges: %d\n", representative->number_of_edges);
    free(list_of_vertices);
```

Capítulo 10

Função path_finder

Esta função, path_finder, tem como objetivo encontrar o caminho mais curto entre duas palavras dado um grafo de palavras similares. O grafo é representado por uma hash table, onde cada palavra é um nó e cada aresta representa a semelhança entre as palavras. A mesma é chamada com três argumentos: a própria hash table, a palavra origem e palavra goal. Priramente é aplicado o método find_word para encontrar todos os nós correspondentes a cada uma dessas duas palavras dentro da hash table. Em seguida, aplicando o método find_representative são encontrados os representativos dos componentes conexos que contêm as palavras fornecidas. A função passa pela verificação das palavras em relação à sua existência na hash table e caso as mesmas pertençam ao mesmo componente conexo, é garantido que tb estejam conectadas no grafo. Caso contrário é imprimida uma mensgaem de erro. Depois disso, ela aloca memória para a lista de vértices usando a função 'malloc' e armazena o endereço dessa memória na variável list_of_vertices para que a possa usar na função breadth_first_search() com o intuito de encontrar o menor caminho entre as palavras dadas. De seguida são printadas todas as palavras pertencentes a esse mesmo caminho e libertada a memória associada à lista de vertices.

```
static void path_finder(hash_table_t *hash_table,const char *from_word,
const char *to_word)
{
  hash_table_node_t *from_node,*to_node, *fromRep, *toRep, **
list_of_vertices, *node;
  int final_index;
  unsigned int i;
  from_node = find_word(hash_table,from_word,0);
  to_node = find_word(hash_table,to_word,0);
  fromRep = find_representative(from_node);
  toRep = find_representative(to_node);
  if(from_node == NULL || to_node == NULL)
  {
    printf("One of the words does not exist in the hash table.\n");
    return;
  }
  if(fromRep != toRep)
  {
    printf("The words '%s' and '%s' do not belong to the same connected
component.\n",from_word,to_word);
    return;
  }
  list_of_vertices = malloc(fromRep->number_of_vertices * sizeof(
hash_table_node_t *));

  if(list_of_vertices == NULL)
  {
```

```
    fprintf(stderr,"path_finder: unable to allocate memory for the list of
vertices\n");
    exit(1);
  }
  final_index = breadth_first_search(hash_table->number_of_entries,
list_of_vertices,to_node,from_node);
  node = list_of_vertices[final_index - 1];
  printf("\nThe shortest path from '%s' to '%s' is:\n",from_word,to_word);

  while (node != NULL)
  {
    printf("%s \n", node->word);
    node = node ->  previous;
  }
  free(list_of_vertices);
```

Capítulo 11

connected_component_diameter

Esa função tem como objetivo calcular o diâmetro de cada componente conexa do grafo representado pela hash table.

Para isso, usamos duas vezes a função breadth_first_search()explicada acima. A primeira procura começa no nó passado como argumento e é usada para encontrar o vértice mais longínquo na componente conexa, guardando os vértices visitados, por ordem, na *list_of_vertices* e a segunda começa nesse mesmo vértice, para encontrar o caminho mais longo da componente conexa, refazendo a *list_of_vertices* criada anteriormente, pois essa foi apenas necessária para guardar o vértice mais longínquo num nó ao qual chamámos *farthest*. De seguida, através de um loop *for*, são percorridos todos os vértices visitados pela segunda procura efetuada e, enquanto esse vértice for diferente de NULL, a função calcula o diâmetro da componente conexa como o número de nós desde cada vértice até ao nó raíz. Finalmente, é retornado o diâmetro máximo de entre estes e libertada a lista de vértices.

Capítulo 12

graph_info

A função graph_info(), novamente como o nome indica, tem a utilidade de fornecer diversos dados estatísticos sobre o grafo representado pela hash table.

Começa por alocar memória para os vértices representativos das componentes conexas do grafo. São também declaradas as variáveis *largest_diameter* e *smallest_diameter* com valores iniciais bem elevado e muito baixo, respetivamente, neste caso, 1000000 e 0.

De seguida, através de um loop *for* percorremos todos os vértices da hash table para encontrar o representativo da componente conexa à qual o atual vértice pertence, através da função já explicada acima find_representative. Caso o respresentativo ainda não tenha sido visitado, é adicionado ao array de vértices representativos, marcado como visitado e, através da função também já explicada connected_component_diameter() é calculado o diâmetro da componente conexa e neste processo de iteração vão sendo atualizadas as variáveis *largest_diameter* e *smallest_diameter*.
Entretanto, redefinimos o "estado"de visitado para todos os vértices e recorremos novamente a um loop *for* que percorre o array de representativos para encontrar a maior componente conexa.

Finalmente temos todos os dados necessários para esta função ser capaz de nos apresentar todas as seguintes informações: **número de arestas, número de vértices, número de componentes conexas, número de vértices da maior componente conexa, número médio de vértices numa componente conexa, ..., o maior diâmetro do grafo e o menor**.

Capítulo 13

hash_table_info

A função hash_table_info consiste na recolha da informção relativa à hash table, sendo recolhida informação acerca do número de entradas na mesma, o número de listas ligadas vazias, o load factor e por fim o número maximo e mínimo de listas ligadas associadas às mesmas. A descoberta desta informação é provenida por um simples ciclo for que percorre toda a hash table, dentro desse ciclo está um ciclo while que ao aceder todos os nós, verifica se os mesmos estão vazios o tamanho dos mesmos. Guarda a informação nas respetivas variáveis e printa-as mais abaixo.

Capítulo 14

Resultados Obtidos

Ao correr o programa com o Valgrind, verificamos que corre sem *memory leaks* para todos os ficheiros e para todos os comandos.

Nas figuras abaixo podemos ver exemplos da utilização deste software.

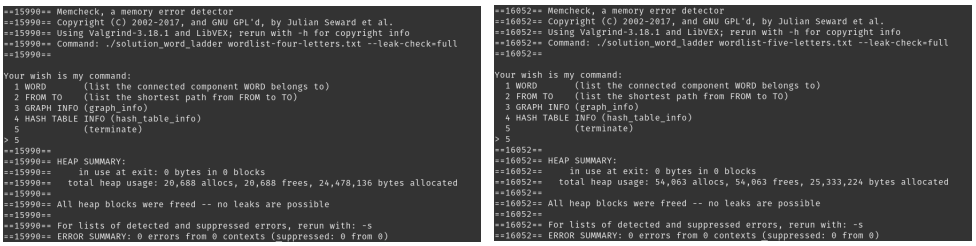

Figura 14.1: Memory Leaks Four & Five Letters

```
==16069== Memcheck, a memory error detector
==16069== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==16069== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==16069== Command: ./solution_word_ladder wordlist-six-letters.txt --leak-check=full
==16069==

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 GRAPH INFO (graph_info)
  4 HASH TABLE INFO (hash_table_info)
  5            (terminate)
> 5
==16069==
==16069== HEAP SUMMARY:
==16069==     in use at exit: 0 bytes in 0 blocks
==16069==   total heap usage: 88,067 allocs, 88,067 frees, 26,420,520 bytes allocated
==16069==
==16069== All heap blocks were freed -- no leaks are possible
==16069==
==16069== For lists of detected and suppressed errors, rerun with: -s
==16069== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figura 14.2: Memory Leaks Six Letters

FROM to TO words

Exemplos do funcionamento do **comando 2** que, neste caso, encontra o menor caminho da palavra "casa"até à palavra "nabo", da palavra "folha"até à palavra "carro", da palavra "exceto"até à palavra "cínico".

1. casa
2. caso
3. cabo
4. nabo

1. folha
2. colha
3. calha
4. calho
5. caldo
6. cardo
7. carro

1. exceto
2. expeto
3. espeto
4. aspeto
5. asseto
6. assedo
7. assado
8. assada
9. ossada
10. ousada
11. ougada
12. sugada
13. segada
14. regada
15. regida
16. retida
17. metida
18. medida
19. medica
20. sódica
21. sónica
22. cónica
23. cínica
24. cínico

Gráficos MatLab

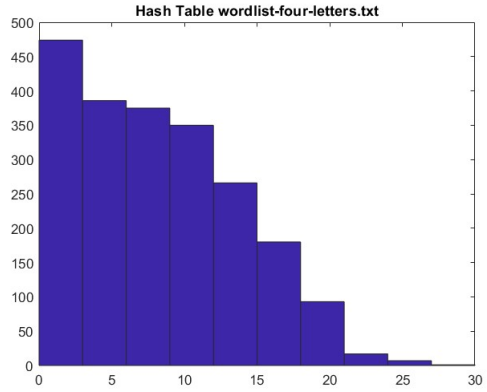

Figura 14.3: Histograma 4 letras

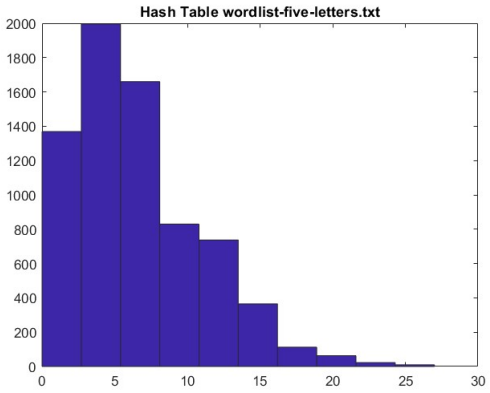

Figura 14.4: Histograma 5 letras

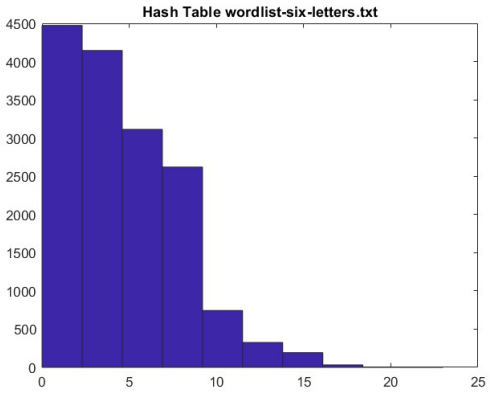

Figura 14.5: Histograma 5 letras

Graphs

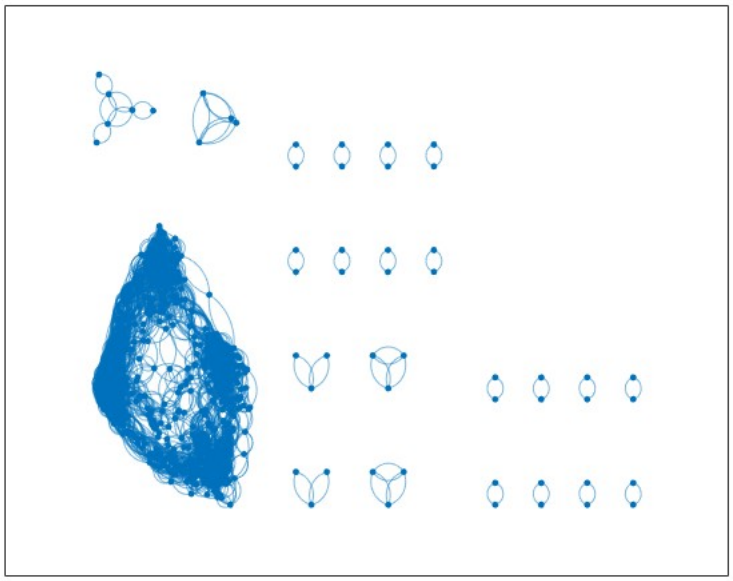

Figura 14.6: Grafo para 4 letras

```
Number of edges: 9267
Number of vertices: 2149
Number of connected components: 2149
Number of vertices in the largest connected component: 1931
Average number of vertices in a connected component: 1.000000
Average number of edges in a connected component: 4.312238
Average degree of a vertex: 4.312238
Largest diameter of the graph: 15
Smallest diameter of the graph: 0
```

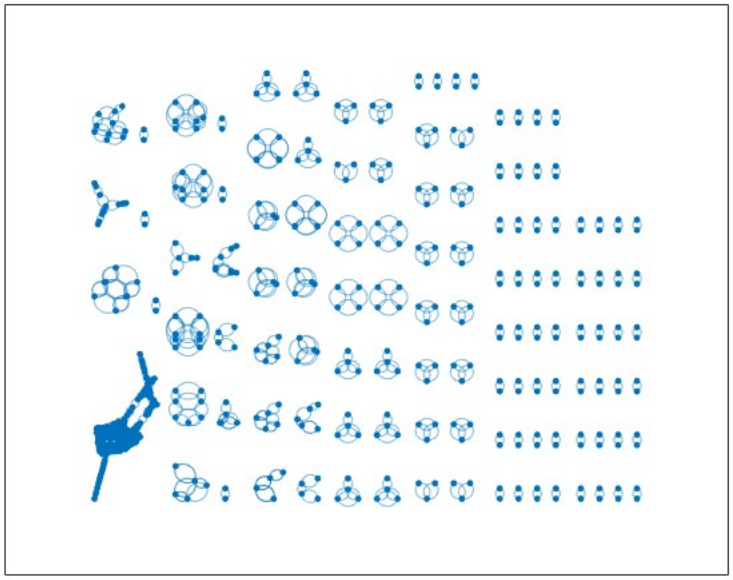

Figura 14.7: Grafo para 5 letras

```
Number of edges: 23446
Number of vertices: 7166
Number of connected components: 7166
Number of vertices in the largest connected component: 6321
Average number of vertices in a connected component: 1.000000
Average number of edges in a connected component: 3.271839
Average degree of a vertex: 3.271839
Largest diameter of the graph: 33
Smallest diameter of the graph: 0
```

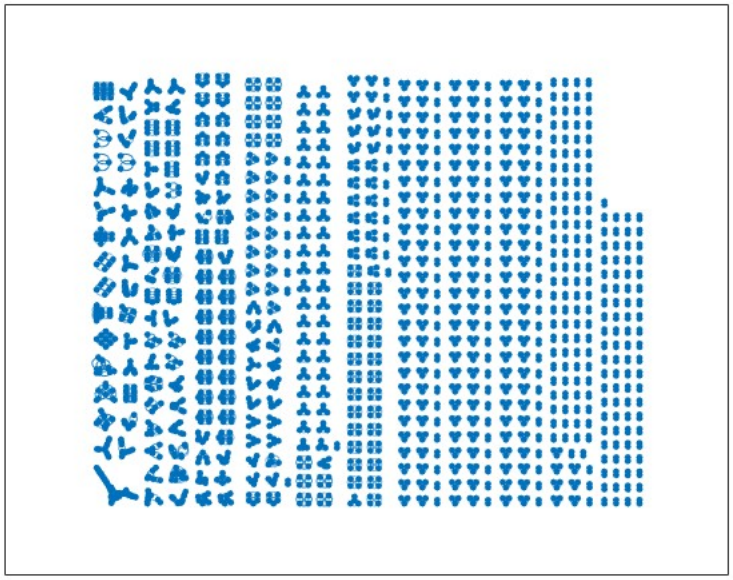

Figura 14.8: Grafo para 6 letras

```
Number of edges: 36204
Number of vertices: 15654
Number of connected components: 15654
Number of vertices in the largest connected component: 11613
Average number of vertices in a connected component: 1.000000
Average number of edges in a connected component: 2.312763
Average degree of a vertex: 2.312763
Largest diameter of the graph: 57
Smallest diameter of the graph: 0
```

Capítulo 15

Conclusão

Para finalizar, achamos que conseguimos cumprir o desafio a que nos propusemos e que descrevemos na secção de contextualização do problema deste relatório, conseguindo mostrar ao utilizador as funcionalidades que pretendíamos. Para além disso, conseguimos representar alguns desses dados em gráficos realizados no matlab, o que achamos ter enriquecido tanto o relatório por ser uma forma de visualizar o problema mais facilmente como a nossa noção visual do problema.

Com este trabalho, aprimorámos significativamente a nossa capacidade de manipulação de hash tables em c, como estas podem funcionar e com que objetivo. Melhorámos também a nossa noção de espaço na memória e como melhor gerir a mesma, pois tivemos de ter em atenção se havia ou não *memory leaks* nalguma parte do nosso programa.

Contribuição (%)

Anderson Lourenço : 33.(3)%

Sara Almeida : 33.(3)%

Hugo Correia : 33.(3)%

Capítulo 16

Código C/Anexo 1

```
1
2  //
3  // AED, November 2022 (Tom s Oliveira e Silva)
4  //
5  // Second practical assignement (speed run)
6  //
7  // Place your student numbers and names here
8  //    N. Mec. 108579 Name : Anderson Lourenco --> PC2
9  //    N. Mec. 108796 Name : Sara Almeida --> PC3
10 //    N. Mec. 108215 Name : Hugo Correia --> PC1
11 //
12 // Do as much as you can
13 //   1) MANDATORY: complete the hash table code
14 //      *) hash_table_create (done)
15 //      *) hash_table_grow
16 //      *) hash_table_free (done)
17 //      *) find_word
18 //      +) add code to get some statistical data about the hash table
19 //   2) HIGHLY RECOMMENDED: build the - (including union-find data) -- use the
       similar_words function...
20 //      *) find_representative
21 //      *) add_edge
22 //   3) RECOMMENDED: implement breadth-first search in the graph
23 //      *) breadh_first_search
24 //   4) RECOMMENDED: list all words belonginh to a connected component
25 //      *) breadh_first_search
26 //      *) list_connected_component
27 //   5) RECOMMENDED: find the shortest path between to words
28 //      *) breadh_first_search
29 //      *) path_finder
30 //      *) test the smallest path from bem to mal
31 //         [ 0] bem
32 //         [ 1] tem
33 //         [ 2] teu
34 //         [ 3] meu
35 //         [ 4] mau
36 //         [ 5] mal
37 //      *) find other interesting word ladders
38 //   6) OPTIONAL: compute the diameter of a connected component and list the
       longest word chain
39 //      *) breadh_first_search
40 //      *) connected_component_diameter
41 //   7) OPTIONAL: print some statistics about the graph
42 //      *) graph_info
43 //   8) OPTIONAL: test for memory leaks
44 //
```

```
45
46 #include <stdio.h>
47
48
49
50
51 #include <stdlib.h>
52 #include <string.h>
53
54
55 //
56 // static configuration
57 //
58
59 #define _max_word_size_  32
60
61
62 //
63 // data structures (SUGGESTION --- you may do it in a different way)
64 //
65
66 typedef struct adjacency_node_s  adjacency_node_t;
67 typedef struct hash_table_node_s hash_table_node_t;
68 typedef struct hash_table_s      hash_table_t;
69
70 struct adjacency_node_s
71 {
72   adjacency_node_t *next;            // link to th enext adjacency list node
73   hash_table_node_t *vertex;         // the other vertex
74 };
75
76 struct hash_table_node_s
77 {
78   // the hash table data
79   char word[_max_word_size_];        // the word
80   hash_table_node_t *next;           // next hash table linked list node
81   // the vertex data
82   adjacency_node_t *head;            // head of the linked list of adjancency
      edges
83   int visited;                       // visited status (while not in use, keep
      it at 0)
84   hash_table_node_t *previous;       // breadth-first search parent
85   // the union find data
86   hash_table_node_t *representative; // the representative of the connected
      component this vertex belongs to
87   int number_of_vertices;            // number of vertices of the conected
      component (only correct for the representative of each connected component)
88   int number_of_edges;               // number of edges of the conected
      component (only correct for the representative of each connected component)
89 };
90
91 struct hash_table_s
92 {
93   unsigned int hash_table_size;      // the size of the hash table array
94   unsigned int number_of_entries;    // the number of entries in the hash table
95   unsigned int number_of_edges;      // number of edges (for information
      purposes only)
96   hash_table_node_t **heads;         // the heads of the linked lists
97 };
98
99
100
101 //
```

```
102 // allocation and deallocation of linked list nodes (done)
103 //
104
105 static adjacency_node_t *allocate_adjacency_node(void)
106 {
107   adjacency_node_t *node;
108
109   node = (adjacency_node_t *)malloc(sizeof(adjacency_node_t));
110   if(node == NULL)
111   {
112     fprintf(stderr,"allocate_adjacency_node: out of memory\n");
113     exit(1);
114   }
115   return node;
116 }
117
118 static void free_adjacency_node(adjacency_node_t *node)
119 {
120   free(node);
121 }
122
123 static hash_table_node_t *allocate_hash_table_node(void)
124 {
125   hash_table_node_t *node;
126
127   node = (hash_table_node_t *)malloc(sizeof(hash_table_node_t));
128   if(node == NULL)
129   {
130     fprintf(stderr,"allocate_hash_table_node: out of memory\n");
131     exit(1);
132   }
133   return node;
134 }
135
136 static void free_hash_table_node(hash_table_node_t *node)
137 {
138   free(node);
139 }
140
141
142 //
143 // hash table stuff (mostly to be done)
144 //
145
146 unsigned int crc32(const char *str)
147 {
148   static unsigned int table[256];
149   unsigned int crc;
150
151   if(table[1] == 0u) // do we need to initialize the table[] array?
152   {
153     unsigned int i,j;
154
155     for(i = 0u;i < 256u;i++)
156       for(table[i] = i,j = 0u;j < 8u;j++)
157         if(table[i] & 1u)
158           table[i] = (table[i] >> 1) ^ 0xAED00022u; // "magic" constant
159         else
160           table[i] >>= 1;
161   }
162   crc = 0xAED02022u; // initial value (chosen arbitrarily)
163   while(*str != '\0')
164     crc = (crc >> 8) ^ table[crc & 0xFFu] ^ ((unsigned int)*str++ << 24);
```

```
165   return crc;
166 }
167
168 //function that prints the hash table
169
170
171
172 static hash_table_t *hash_table_create(void)
173 {
174   hash_table_t *hash_table;
175   unsigned int i;
176
177   hash_table = (hash_table_t *)malloc(sizeof(hash_table_t));
178   if(hash_table == NULL)
179   {
180     fprintf(stderr,"create_hash_table: out of memory\n");
181     exit(1);
182   }
183   //
184   // complete this
185   hash_table->hash_table_size = 3000507;
186   hash_table->number_of_entries = 0;
187   hash_table->number_of_edges = 0;
188   hash_table->heads = (hash_table_node_t **)malloc(hash_table->hash_table_size
      * sizeof(hash_table_node_t *));
189   if (hash_table->heads == NULL) {
190     fprintf(stderr,"create_hash_table: out of memory\n");
191     exit(1);
192   }
193   for (i = 0; i < hash_table->hash_table_size; i++) {
194     hash_table->heads[i] = NULL;
195   }
196   return hash_table;
197 }
198
199
200 static void hash_table_grow(hash_table_t *hash_table)
201 {
202   // Create a new, larger hash table
203   unsigned int new_size = hash_table->hash_table_size * 2;
204   hash_table_node_t **new_heads = (hash_table_node_t **)calloc(new_size, sizeof
      (hash_table_node_t *));
205   if (new_heads == NULL)
206   {
207     // Handle allocation failure
208     fprintf(stderr, "hash_table_grow: out of memory\n");
209     exit(1);
210   }
211
212   // Re-hash all of the entries in the old table into the new one
213   for (unsigned int i = 0; i < hash_table->hash_table_size; i++)
214   {
215     hash_table_node_t *node = hash_table->heads[i];
216     while (node != NULL)
217     {
218       // Calculate the new hash value for the word
219       unsigned int hash = crc32(node->word) % new_size;
220
221       // Add the node to the new hash table
222       node->next = new_heads[hash];
223       new_heads[hash] = node;
224
225       // Move to the next node in the linked list
```

```
226       node = node->next;
227     }
228   }
229
230   // Free the old array of hash table heads
231   free(hash_table->heads);
232
233   // Update the hash table with the new size and heads array
234   hash_table->hash_table_size = new_size;
235   hash_table->heads = new_heads;
236 }
237
238
239 static void hash_table_free(hash_table_t *hash_table)
240 {
241   // Free all nodes in the hash table
242   for (int i = 0; i < hash_table->hash_table_size; i++)
243   {
244     hash_table_node_t *node = hash_table->heads[i];
245     while (node != NULL)
246     {
247       hash_table_node_t *next_node = node->next;
248
249       // Free all adjacency list nodes for this hash table node
250       adjacency_node_t *adj_node = node->head;
251       while (adj_node != NULL)
252       {
253         adjacency_node_t *next_adj_node = adj_node->next;
254         free_adjacency_node(adj_node);
255         adj_node = next_adj_node;
256       }
257
258       free_hash_table_node(node);
259       node = next_node;
260     }
261   }
262
263   // Free the array of hash table heads
264   free(hash_table->heads);
265
266   // Free the hash table itself
267   free(hash_table);
268 }
269
270 // funtion that prints the hash table
271 static void hash_table_print(hash_table_t *hash_table)
272 {
273   int i, count;
274
275   for (int i = 0; i < hash_table->hash_table_size; i++)
276   {
277     hash_table_node_t *node = hash_table->heads[i];
278     adjacency_node_t *adj_node = node->head;
279     while (node != NULL)
280     {
281       printf("%s ", node->word);
282
283       count=0;
284
285       adjacency_node_t *adj_node = node->head;
286       while(adj_node != NULL)
287       {
288         count = count+1;
```

```
289         adj_node = adj_node->next;
290       }
291
292       printf("%d\n",count);
293       node = node->next;
294
295     }
296   }
297 }
298
299 static hash_table_node_t *find_word(hash_table_t *hash_table, const char *word,
      int insert_if_not_found)
300 {
301   unsigned int i = crc32(word) % hash_table->hash_table_size;
302   hash_table_node_t *node = hash_table->heads[i];
303   while (node != NULL)
304   {
305     if (strcmp(node->word, word) == 0)
306     {
307       // Word found, return the node
308       return node;
309     }
310     node = node->next;
311   }
312   if (insert_if_not_found && strlen(word) < _max_word_size_)
313   {
314     // Word not found, insert it
315     node = allocate_hash_table_node();
316     if (node == NULL)
317     {
318       // Memory allocation failed
319       return NULL;
320     }
321     strcpy(node->word, word);
322     node->next = hash_table->heads[i];
323     node->representative = node;
324     node->previous = NULL;
325     node->number_of_vertices = 1;
326     node->number_of_edges = 0;
327     node-> visited = 0;
328     node->head = NULL;
329     hash_table->heads[i] = node;
330     hash_table->number_of_entries++;
331
332     if (hash_table->number_of_entries > hash_table->hash_table_size*0.75)
333     {
334       // Hash table is getting full, resize it
335       hash_table_grow(hash_table);
336     }
337   }
338   else
339   {
340     // Word not found and insert_if_not_found flag is not set
341     return NULL;
342   }
343   return node;
344 }
345
346
347
348
349 //
350 // add edges to the word ladder graph (mostly do be done)
```

```
351 //
352
353 static hash_table_node_t *find_representative(hash_table_node_t *node)
354 {
355   hash_table_node_t *representative, *next_node;
356
357   representative = node;
358   while (representative != representative->representative)
359   {
360     representative = representative->representative;
361   }
362
363   // Path compression optimization
364   next_node = node;
365  for (next_node = node; next_node != representative; next_node = next_node->
       representative)
366   {
367     hash_table_node_t *temp = next_node->representative;
368     next_node->representative = representative;
369     next_node = temp;
370   }
371
372   return representative;
373 }
374
375
376 static void add_edge(hash_table_t *hash_table, hash_table_node_t *from, const
       char *word)
377 {
378     hash_table_node_t *to, *from_representative, *to_representative;
379     adjacency_node_t *link_from, *link_to;
380
381     from_representative = find_representative(from);
382     to = find_word(hash_table, word, 0);
383
384     if (to == NULL || to == from)
385         return;
386
387     to_representative = find_representative(to);
388     if (from_representative == to_representative)
389     {
390         from_representative->number_of_vertices++;
391     }
392     else if (from_representative->number_of_vertices < to_representative->
       number_of_vertices)
393     {
394         from_representative->representative = to_representative;
395         to_representative->number_of_vertices += from_representative->
       number_of_vertices;
396         to_representative->number_of_edges += from_representative->
       number_of_edges;
397     }
398     else
399     {
400         to_representative->representative = from_representative;
401         from_representative->number_of_vertices += to_representative->
       number_of_vertices;
402         from_representative->number_of_edges += to_representative->
       number_of_edges;
403     }
404
405     link_from = allocate_adjacency_node();
406     link_to = allocate_adjacency_node();
```

```
407
408     if (link_from == NULL || link_to == NULL)
409     {
410         fprintf(stderr, "add_edge: out of memory\n");
411         exit(1);
412     }
413
414     link_from->vertex = to;
415     link_from->next = from->head;
416     from->head = link_from;
417
418     link_to->vertex = from;
419     link_to->next = to->head;
420     to->head = link_to;
421
422     from_representative->number_of_edges++;
423     to_representative->number_of_edges++;
424     hash_table->number_of_edges++;
425     return;
426 }
427
428
429
430 //
431 // generates a list of similar words and calls the function add_edge for each
        one (done)
432 //
433 // man utf8 for details on the uft8 encoding
434 //
435
436 static void break_utf8_string(const char *word,int *individual_characters)
437 {
438   int byte0,byte1;
439
440   while(*word != '\0')
441   {
442     byte0 = (int)(*(word++)) & 0xFF;
443     if(byte0 < 0x80)
444       *(individual_characters++) = byte0; // plain ASCII character
445     else
446     {
447       byte1 = (int)(*(word++)) & 0xFF;
448       if((byte0 & 0b11100000) != 0b11000000 || (byte1 & 0b11000000) != 0
      b10000000)
449       {
450         fprintf(stderr,"break_utf8_string: unexpected UFT-8 character\n");
451         exit(1);
452       }
453       *(individual_characters++) = ((byte0 & 0b00011111) << 6) | (byte1 & 0
      b00111111); // utf8 -> unicode
454     }
455   }
456   *individual_characters = 0; // mark the end!
457 }
458
459 static void make_utf8_string(const int *individual_characters,char word[
        _max_word_size_])
460 {
461   int code;
462
463   while(*individual_characters != 0)
464   {
465     code = *(individual_characters++);
```

```
466     if(code < 0x80)
467       *(word++) = (char)code;
468     else if(code < (1 << 11))
469     { // unicode -> utf8
470       *(word++) = 0b11000000 | (code >> 6);
471       *(word++) = 0b10000000 | (code & 0b00111111);
472     }
473     else
474     {
475       fprintf(stderr,"make_utf8_string: unexpected UFT-8 character\n");
476       exit(1);
477     }
478   }
479   *word = '\0';  // mark the end
480 }
481
482 static void similar_words(hash_table_t *hash_table,hash_table_node_t *from)
483 {
484   static const int valid_characters[] =
485   { // unicode!
486     0x2D,
       // -
487     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
488     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
489     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
490     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
491     0xC1,0xC2,0xC9,0xCD,0xD3,0xDA,
       //
492     0xE0,0xE1,0xE2,0xE3,0xE7,0xE8,0xE9,0xEA,0xED,0xEE,0xF3,0xF4,0xF5,0xFA,0xFC,
       //
493     0
494   };
495   int i,j,k,individual_characters[_max_word_size_];
496   char new_word[2 * _max_word_size_];
497
498   break_utf8_string(from->word,individual_characters);
499   for(i = 0;individual_characters[i] != 0;i++)
500   {
501     k = individual_characters[i];
502     for(j = 0;valid_characters[j] != 0;j++)
503     {
504       individual_characters[i] = valid_characters[j];
505       make_utf8_string(individual_characters,new_word);
506       // avoid duplicate cases
507       if(strcmp(new_word,from->word) > 0)
508         add_edge(hash_table,from,new_word);
509     }
510     individual_characters[i] = k;
511   }
512 }
513
514 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) {
515   int r = 0, w = 1;
516   list_of_vertices[0] = origin;
517   origin->previous = NULL;
518   origin->visited = 1;
519   int found = 0;
```

```
520   while (r != w) {
521     adjacency_node_t *node = list_of_vertices[r++]->head;
522     if (found) {
523       break;
524     }
525     while (node != NULL) {
526       if (node->vertex->visited == 0) {
527         node->vertex->visited = 1;
528         node->vertex->previous = list_of_vertices[r - 1];
529         list_of_vertices[w++] = node->vertex;
530         if (node->vertex == goal) {
531           found = 1;
532           break;
533         }
534       }
535       node = node->next;
536     }
537   }
538   for (int i = 0; i < w; i++) {
539     list_of_vertices[i]->visited = 0;
540   }
541   return w;
542 }
543
544
545
546 //
547 // list all vertices belonging to a connected component (complete this)
548 //
549
550 static void list_connected_component(hash_table_t *hash_table, const char *word
      ) {
551   // Find the node for the given word
552   hash_table_node_t *node = find_word(hash_table, word, 0);
553   int v = 0;
554   if (node == NULL) {
555     // The given word does not exist in the hash table
556     printf("The word '%s' does not exist in the hash table.\n", word);
557     return;
558   }
559
560   // Find the representative node for the connected component containing the
       given node
561   hash_table_node_t *representative = find_representative(node);
562   hash_table_node_t **list_of_vertices = malloc(representative->
       number_of_vertices * sizeof(hash_table_node_t *));
563   printf("The connected component containing '%s' has %d vertices.\n", word,
       representative->number_of_vertices);
564
565   int number_of_vertices = breadth_first_search(hash_table->number_of_entries,
       list_of_vertices, node, NULL);
566   // Print all of the words belonging to the connected component
567   for (unsigned int i = 0; i < number_of_vertices; i++) {
568     printf("%s\n", list_of_vertices[i]->word);
569     v++;
570   }
571   printf("\nNumber of vertices: %d\n", v);
572   printf("Number of edges: %d\n", representative->number_of_edges);
573   free(list_of_vertices);
574 }
575
576
577
```

```
578 //
579 // compute the diameter of a connected component (optional)
580 //
581
582 //static int largest_diameter;
583 //static hash_table_node_t **largest_diameter_example;
584
585 // static int connected_component_diameter(hash_table_node_t *node)
586 // {
587 //   int diameter;
588
589 //   int max_distance = 0;
590 //   queue_t *queue = queue_create();
591 //   hash_table_node_t *curr;
592 //   hash_table_node_t *neighbor;
593
594 //   queue_enqueue(queue, node);
595 //   while (!queue_isempty(queue)) {
596 //     curr = queue_dequeue(queue);
597
598 //     for (int i = 0; i < curr->numNeighbors; i++) {
599 //       neighbor = curr->neighbors[i];
600
601 //       if (neighbor->visited == false) {
602 //         neighbor->visited = true;
603
604 //         int distance = curr->distance + 1;
605
606 //         if (distance > max_distance) {
607 //           max_distance = distance;
608 //         }
609
610 //         neighbor->distance = distance;
611
612 //         queue_enqueue(queue, neighbor);
613 //       }
614 //     }
615 //   }
616 // diameter = max_distance;
617
618 //   return diameter;
619 // }
620 static int connected_component_diameter(hash_table_node_t *node)
621 {
622     int diameter = 0;
623     int maximum_number_of_vertices = find_representative(node)->
    number_of_vertices;
624     hash_table_node_t **list_of_vertices = (hash_table_node_t **)malloc(
    maximum_number_of_vertices * sizeof(hash_table_node_t *));
625
626     if (list_of_vertices == NULL) {
627         fprintf(stderr, "connected_component_diameter: out of memory\n");
628         exit(1);
629     }
630
631     int n = breadth_first_search(maximum_number_of_vertices, list_of_vertices,
    node, NULL);
632     hash_table_node_t *farthest = list_of_vertices[n - 1];
633
634     int m = breadth_first_search(maximum_number_of_vertices, list_of_vertices,
    farthest, NULL);
635
636     for (int i = 0; i < m; i++) {
```

```
637         int temp_diameter = 0;
638         hash_table_node_t *p = list_of_vertices[i];
639         while (p != NULL) {
640             temp_diameter++;
641             p = p->previous;
642         }
643         if (temp_diameter > diameter) {
644           diameter = temp_diameter;
645         }
646 
647     }
648     free(list_of_vertices);
649     return diameter;
650 }
651 
652 
653 //
654 // find the shortest path from a given word to another given word (to be done)
655 //
656 
657 static void path_finder(hash_table_t *hash_table,const char *from_word,const
      char *to_word)
658 {
659   hash_table_node_t *from_node,*to_node, *fromRep, *toRep, **list_of_vertices,
       *node;
660   int final_index;
661   unsigned int i = 0;
662 
663   // find the nodes for the given words
664   from_node = find_word(hash_table,from_word,0);
665   to_node = find_word(hash_table,to_word,0);
666 
667   // find the representative nodes for the connected components containing the
       given nodes
668   fromRep = find_representative(from_node);
669   toRep = find_representative(to_node);
670 
671   // check if the words exist in the hash table
672   if(from_node == NULL || to_node == NULL)
673   {
674     printf("One of the words does not exist in the hash table.\n");
675     return;
676   }
677 
678   // check if the words belong to the same connected component
679   if(fromRep != toRep)
680   {
681     printf("The words '%s' and '%s' do not belong to the same connected
       component.\n",from_word,to_word);
682     return;
683   }
684 
685   // allocate memory for the list of vertices
686   list_of_vertices = malloc(fromRep->number_of_vertices * sizeof(
       hash_table_node_t *));
687 
688   if(list_of_vertices == NULL)
689   {
690     fprintf(stderr,"path_finder: unable to allocate memory for the list of
       vertices\n");
691     exit(1);
692   }
693 
```

```
694   // find the shortest path
695   final_index = breadth_first_search(hash_table->number_of_entries,
        list_of_vertices,to_node,from_node);
696   node = list_of_vertices[final_index - 1];
697
698   // print the shortest path
699   printf("\nThe shortest path from '%s' to '%s' is:\n",from_word,to_word);
700
701   while (node != NULL)
702   {
703
704     printf("[ %d] %s \n", node->word);
705     i++;
706     node = node ->  previous;
707   }
708   // free memory
709   free(list_of_vertices);
710 }
711
712
713 //
714 // some graph information (optional)
715 //
716
717 static void graph_info(hash_table_t *hash_table)
718 {
719   int nrRepresentatives = 0;
720   int smallest_diameter = 1000000; //d
721   int largest_diameter = 0; //d
722   hash_table_t **representatives = (hash_table_t **)malloc(hash_table->
        number_of_entries * sizeof(hash_table_t));
723
724   // Find the representatives of each connected component
725   for (int i = 0; i < hash_table->hash_table_size; i++)
726   {
727     for (hash_table_node_t *vertex = hash_table->heads[i]; vertex != NULL;
        vertex = vertex->next)
728     {
729       // Find the representative of the connected component
730       hash_table_node_t *representative = find_representative(vertex);
731
732       // Add the representative to the array if it has not already been added
733       if (!representative->visited)
734       {
735         representatives[nrRepresentatives++] = representative;
736         representative->visited = 1;
737
738         // Find the diameter of the connected component
739         int diam = connected_component_diameter(representative)-1;
740         if (diam > largest_diameter)
741         {
742           largest_diameter = diam;
743         }
744         if (diam < smallest_diameter)
745         {
746           smallest_diameter = diam;
747         }
748       }
749     }
750   }
751
752   // Reset the visited status of all vertices
753   for (int i = 0; i < hash_table->hash_table_size; i++)
```

```
754   {
755     for (hash_table_node_t *vertex = hash_table->heads[i]; vertex != NULL;
      vertex = vertex->next)
756     {
757       vertex->visited = 0;
758     }
759   }
760
761   // Find the largest connected component
762   int largestComponent = 0;
763   for (int i = 0; i < nrRepresentatives; i++)
764   {
765     int componentSize = 0;
766     for (int j = 0; j < hash_table->hash_table_size; j++)
767     {
768       for (hash_table_node_t *vertex = hash_table->heads[j]; vertex != NULL;
      vertex = vertex->next)
769       {
770         if (find_representative(vertex) == representatives[i])
771         {
772           componentSize++;
773         }
774       }
775     }
776     //largest_diameter=connected_component_diameter(hash_table);
777
778     if (componentSize > largestComponent)
779     {
780       largestComponent = componentSize;
781     }
782   }
783
784   //display the number of edges
785   printf("\nNumber of edges: %d \n", hash_table->number_of_edges);
786   //display the numver of vertices
787   printf("Number of vertices: %d \n", hash_table->number_of_entries);
788   //display the number of connected components
789   printf("Number of connected components: %d \n", nrRepresentatives);
790   //display the number of vertices in the largest connected component
791   printf("Number of vertices in the largest connected component: %d \n",
      largestComponent);
792   //display the average number of vertices in a connected component
793   printf("Average number of vertices in a connected component: %f \n", (float)
      hash_table->number_of_entries / nrRepresentatives);
794   //display the average number of edges in a connected component
795   printf("Average number of edges in a connected component: %f \n", (float)
      hash_table->number_of_edges / nrRepresentatives);
796   //display the average degree of a vertex
797   printf("Average degree of a vertex: %f \n", (float)hash_table->
      number_of_edges / hash_table->number_of_entries);
798   //display largest diameter of the graph
799   printf("Largest diameter of the graph: %d \n", largest_diameter);
800   //display the smallest diameter of the graph
801   printf("Smallest diameter of the graph: %d \n", smallest_diameter);
802
803
804   free(representatives);
805 }
806
807 static void hash_table_info(hash_table_t *hash_table){
808
809   int number_of_collisions = 0, number_of_empty_lists = 0, max = 0, min =
      100000, number_of_lists = 0;
```

```
  for (int i = 0; i < hash_table->hash_table_size; i++){
    int length = 0;
    hash_table_node_t *node = hash_table->heads[i];
    while (node != NULL){
      length++;
      node = node->next;
    }
    if (length > max){
      max = length;
    }
    if (length < min && length != 0){
      min = length;
    }
    if (length == 0){
      number_of_empty_lists++;
    }
  }

  printf("\nNumber of entries: %d\n", hash_table->number_of_entries);
  printf("Number of empty list: %d\n", number_of_empty_lists);
  printf("Load factor: %f\n", (float)hash_table->number_of_entries / (float)
    hash_table->hash_table_size);
  printf("Max length of linked lists: %d\n", max);
  printf("Min length of linked lists: %d\n", min);

}

//
// 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-four-letters.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);
  //hash_table_print(hash_table);
  // ask what to do

  //code to create graph
  // for (unsigned int i = 0; i < hash_table->hash_table_size; i++) // loop
    through the hash table
```

```
871   // {
872   //   hash_table_node_t *node = hash_table->heads[i]; // set node to the first
        element of the hash table
873   //   while (node != NULL)                            // while the node has a
       next node
874   //   {
875   //     hash_table_node_t *temp = node;          // set temp to the node
876   //     node = node->next;                       // set node to the next node
                          // free the temp node
877   //     adjacency_node_t *adj_node = temp->head; // set adj_node to the first
       element of the adjacency list
878   //     while (adj_node != NULL)                 // while the adj_node has a
       next node
879   //     {
880   //       adjacency_node_t *temp_adj = adj_node;                 // set
       temp_adj to the adj_node
881   //       adj_node = adj_node->next;                             // set
       adj_node to the next node
882   //       printf("%s %s\n", temp->word, temp_adj->vertex->word); // print the
       word and the word in the adjacency list")
883   //     }
884   //   }
885   // }
886   // return 0;
887
888   for(;;)
889   {
890     fprintf(stderr,"\nYour wish is my command:\n");
891     fprintf(stderr,"  1 WORD       (list the connected component WORD belongs
      to)\n");
892     fprintf(stderr,"  2 FROM TO    (list the shortest path from FROM to TO)\n")
      ;
893     fprintf(stderr,"  3 GRAPH INFO (graph_info)\n");
894     fprintf(stderr,"  4 HASH INFO  (hash_table_info)\n");
895     fprintf(stderr,"  5 HASH PRINT (hash_table_print)\n");
896     fprintf(stderr,"  6            (terminate)\n");
897     fprintf(stderr,"> ");
898     if(scanf("%99s",word) != 1)
899       break;
900     command = atoi(word);
901     if(command == 1)
902     {
903       if(scanf("%99s",word) != 1)
904         break;
905       list_connected_component(hash_table,word);
906     }
907     else if(command == 2)
908     {
909       if(scanf("%99s",from) != 1)
910         break;
911       if(scanf("%99s",to) != 1)
912         break;
913       path_finder(hash_table,from,to);
914     }
915     else if(command == 3){
916       graph_info(hash_table);
917     }
918
919     else if (command == 4)
920     {
921       hash_table_info(hash_table);
922
923     } else if (command == 5){
```

```
924         hash_table_print(hash_table);
925 
926       } else if (command == 6){
927         break;
928       }
929     }
930     // clean up
931     hash_table_free(hash_table);
932     return 0;
933 }
```

Capítulo 17

Código MatLab/Anexo 2

```
1
2  four_letters = readcell("graph6.txt");
3  %five_letters = readcell("graph5.txt");
4  %six_letters = readcell("graph6.txt");
5
6  connections_four = four_letters(:,2);
7  %connections_five = five_letters(:,2);
8  %connections_six = six_letters(:,2);
9
10 dados_four_letters = [];
11 dados_five_letters = [];
12 dados_six_letters = [];
13
14 for i=1:length(four_letters(:,2))
15     dados_four_letters(i,1)=connections_four{i};
16 end
17
18 %for i=1:length(five_letters(:,2))
19 %     dados_five_letters(i,1)=connections_five{i};
20 %end
21
22 %for i=1:length(six_letters(:,2))
23 %     dados_six_letters(i,1)=connections_six{i};
24 %end
25
26 figure('name', 'Hash_Table wordlist_six_letters.txt')
27 hist(dados_four_letters)
28 title('Hash Table wordlist-six-letters.txt')
```

Figure Graphs

```
1
2  data=readtable("graph.txt");
3  G=graph;
4  words=unique(data(:,1));
5  for i=1:height(words)
6      G = addnode(G,words{i,1});
7  end
8  for i=1:height(data)
9      G=addedge(G,data{i,1},data{i,2});
10 end
11
12 plot(G)
```