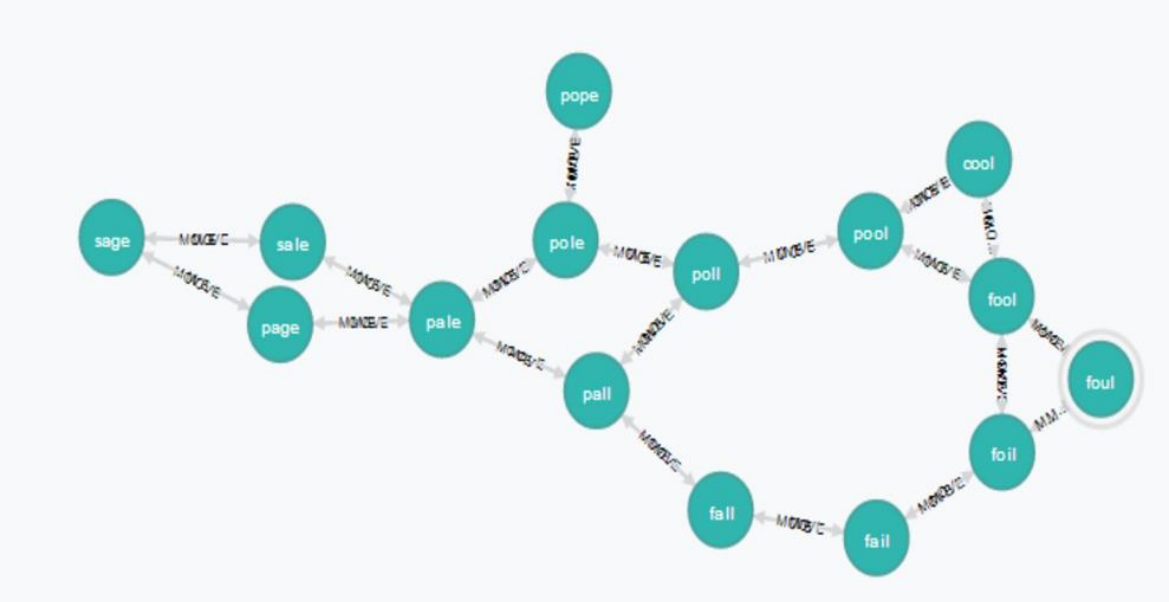
universidade de aveiro

# WORD LADDDER

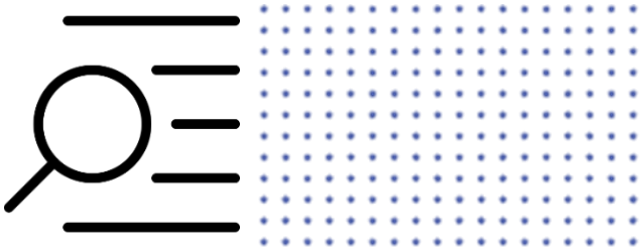
## ALGORITMOS E ESTRUTURAS DE DADOS 2022

Prof. Tomás Oliveira e Silva


André Oliveira / 107637 33.(3)%
Duarte Cruz / 107359 33.(3)%
Rodrigo Graça / 107634 33.(3)%

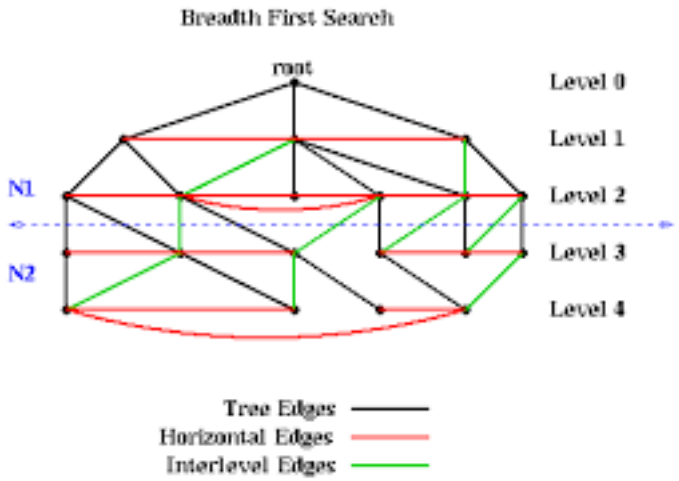
# Índice



# INTRODUÇÃO

Um *Word Ladder*, também conhecido como Escada de Palavras, é um jogo em que o objetivo é transformar uma palavra em outra usando apenas uma única alteração de letra em cada etapa.

Neste relatório, será descrito como foi usada uma *Hash Table* para implementar um programa em C que resolve o jogo da Escada de Palavras. Isso inclui a descrição da sua estrutura de dados, dos algoritmos utilizados e do funcionamento do programa. Além disso, são apresentados os resultados obtidos ao testar o programa com diferentes entradas e discutidas as suas conclusões.

# FUNÇÕES

## Função *hash_table_create()*

```c
static hash_table_t *hash_table_create(void)
{
  // create a new hash table
  hash_table_t *hash_table;

  // allocate the hash table
  hash_table = (hash_table_t *)malloc(sizeof(hash_table_t));
  // check for allocation errors
  if (hash_table == NULL)
  {
    return NULL;
  }

  // allocate the array of linked list heads
  hash_table->heads = (hash_table_node_t **)malloc(sizeof(hash_table_node_t *));
  // check for allocation errors
  if (hash_table->heads == NULL)
  {
    free(hash_table);
    return NULL;
  }

  // initialize the hash table
  hash_table->hash_table_size = 250;
  hash_table->number_of_entries = 0;
  hash_table->number_of_edges = 0;

  // Fill the array of linked list heads with NULL
  for (int i = 0; i < hash_table->hash_table_size; i++)
  {
    hash_table->heads[i] = NULL;
  }

  return hash_table;
}
```

*Figura 1 | Função hash_table_create()*

A função tem como objetivo criar uma *Hash Table* e inicializá-la. Começa declarando uma variável do tipo *hash_table_t*, que é a estrutura que representa uma *Hash Table*. Em seguida, aloca memória para essa estrutura usando a função malloc, que aloca um bloco de memória do tamanho especificado em bytes. Se a alocação de memória falhar, a função retorna *NULL*.

Em seguida, aloca memória para um *array* de ponteiros para nós da *Hash Table*, que será usado para armazenar as entradas na tabela. Novamente, verificaremos se a alocação de memória falhou e, caso tenha falhado, libera a memória alocada para a estrutura da *Hash Table* e também retorna *NULL*.

É definido ainda o tamanho inicial da *Hash Table* como 250 e inicializa as variáveis *number_of_entries* e *number_of_edges* como 0. Estas variáveis são usadas para armazenar o número de entradas e o número de arestas, respetivamente, na *Hash Table*.

Por fim, percorremos o *array* de cabeças da *Hash Table* e inicializa cada posição como *NULL*, o que significa que não há nenhuma entrada na *Hash Table* ainda.

A função retorna um ponteiro para a nova *Hash Table* criada, ou *NULL* se a alocação de memória falhar.

## Função *hash_table_grow()*

```c
static void hash_table_grow(hash_table_t *hash_table)
{
  // the old and new array of linked list heads
  hash_table_node_t **old_heads, **new_heads, *node, *next;
  unsigned int old_size, i;

  // save a pointer to the old array of linked list heads and its size
  old_heads = hash_table->heads;
  old_size = hash_table->hash_table_size;

  // create a new hash table with a larger size
  hash_table->hash_table_size *= 2;
  new_heads = (hash_table_node_t **)malloc(hash_table->hash_table_size * sizeof(hash_table_node_t *));

  // Fill the array of linked list heads with NULL
  for (i = 0; i < hash_table->hash_table_size; i++)
    new_heads[i] = NULL;

  if (new_heads == NULL)
  {
    fprintf(stderr, "Error: out of memory");
    exit(1);
  }

  // rehash the entries from the old array to the new one
  for (i = 0; i < old_size; i++)
  {
    node = old_heads[i];
    while (node != NULL)
    {

      next = node->next;

      size_t value = crc32(node->word) % hash_table->hash_table_size;
      node->next = new_heads[value];
      new_heads[value] = node;

      node = next;
    }
  }
  free(old_heads);
  hash_table->heads = new_heads;
}
```

*Figura 2 | Função hash_table_grow()*

Esta função tem como objetivo redimensionar (aumentar o tamanho) uma *Hash Table* existente. Começa guardando um ponteiro para o antigo *array* de cabeças da *Hash Table* e seu tamanho. Em seguida, aumenta o tamanho da *Hash Table* para o dobro do tamanho anterior e aloca memória para um novo *array* de cabeças da *Hash Table*.

Posteriormente, preenche o novo *array* de cabeças com NULL. Se a alocação de memória falhar, exibe uma mensagem de erro e finaliza o programa.

Por fim, percorre o antigo *array* de cabeças da *Hash Table* e, para cada entrada, "rehasha" o valor da palavra (usando a função crc32) e insere o nó na nova *Hash Table* usando o novo tamanho. Liberta ainda a memória alocada para o antigo array de cabeças e atualiza o ponteiro da *Hash Table* para apontar para o novo *array* de cabeças.

A função é chamada quando a *Hash Table* atinge um determinado limite de carga (número de entradas dividido pelo tamanho da *Hash Table*), para evitar que a performance da tabela se degrade muito. Ao redimensionar a *Hash Table*, a carga é distribuída de maneira mais equilibrada e o tempo de acesso às entradas é mantido em um nível aceitável.

## Função *hash_table_free()*

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

  for (i = 0; i < hash_table->hash_table_size; i++)
  {
    node = hash_table->heads[i];
    while (node != NULL)
    {
      temp = node;
      adj_node = node->head;
      while (adj_node != NULL)
      {
        temp_adj = adj_node;
        adj_node = adj_node->next;
        free(temp_adj);
      }
      node = node->next;
      free(temp);
    }
  }

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

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

*Figura 3 | Função hash_table_free()*

A função *hash_table_free* tem como objetivo liberar toda a memória alocada para uma *Hash Table*. Começa por declarar quatro variáveis: dois ponteiros para nós da *Hash Table* (*node* e *temp*) e dois ponteiros para nós de adjacência (*adj_node* e *temp_adj*).

Em seguida, a percorre o *array* de cabeças da *Hash Table* e, para cada entrada, atribui a variável node o ponteiro para o nó atual e, em seguida, entra em um *loop* para percorrer a lista de adjacência do nó. Dentro desse *loop*, à variável *adj_node* é atribuída o ponteiro para o nó de adjacência atual e à variável *temp_adj* é atribuída o ponteiro para o próximo nó de adjacência. Em seguida, libera a memória alocada para o nó de adjacência atual (*temp_adj*) e atualiza o ponteiro *adj_node* para o próximo nó de adjacência.

Quando o *loop* que percorrer a lista de adjacência termina, a função atribui à variável *temp* o ponteiro para o nó atual e atualiza o ponteiro node para o próximo nó da *Hash Table*. Em seguida, libera a memória alocada para o nó atual (*temp*).

Quando o próximo *loop* que percorre o *array* de cabeças da *Hash Table* termina, a função libera a memória alocada para o *array* de cabeças da *Hash Table*. Por fim, a função libera a memória alocada para a estrutura da *Hash Table*.

Esta função é importante para garantir que não haja vazamento de memória quando a *Hash Table* não é mais necessária. Deve ser chamada após o uso da *Hash Table* para liberar a memória alocada e evitar problemas com o gerenciamento de memória do sistema.

## Função *find_word()*

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

  // find the word in the hash table
  i = crc32(word) % hash_table->hash_table_size;
  node = hash_table->heads[i];
  while (node != NULL)
  {
    if (strcmp(node->word, word) == 0)
      return node;
    node = node->next;
  }

  // if the word is not found, insert it into the hash table
  if (insert_if_not_found && strlen(word) < _max_word_size_)
  {
    node = allocate_hash_table_node();          // allocate a new node
    strncpy(node->word, word, _max_word_size_); // copy the word into the node
    node->representative = node;                // the representative of a node is itself
    node->next = hash_table->heads[i];          // insert the node at the head of the linked list
    node->previous = NULL;
    node->number_of_edges = 0;
    node->number_of_vertices = 1;
    node->visited = 0;
    node->head = NULL;
    hash_table->heads[i] = node;
    hash_table->number_of_entries++;
    // if the number of entries exceeds the size of the hash table, grow the hash table
    if (hash_table->number_of_entries > hash_table->hash_table_size)
      hash_table_grow(hash_table);
    return node;
  }

  return NULL;
}
```

*Figura 4 | Função find_word()*

Esta função tem como objetivo encontrar uma determinada palavra em uma *Hash Table* ou, se a opção *insert_if_not_found* for verdadeira, inserir a palavra na *Hash Table* caso ela não esteja presente.

A função começa por fazer o "rehash" da palavra usando a função crc32 e usando o resultado para encontrar a posição correspondente no *array* de cabeças da *Hash Table*. Em seguida, atribui a uma variável node o ponteiro para o nó na cabeça da *linked list* na posição encontrada e entra em um *loop* para percorrer a mesma. Dentro desse loop, a função compara a palavra no nó atual com a palavra procurada e, se elas forem iguais, retorna o nó. Caso contrário, o ponteiro node é atualizado para o próximo nó da *linked list* e o *loop* continua.

Se o *loop* terminar sem encontrar a palavra procurada, a opção *insert_if_not_found* é verdadeira e o tamanho da palavra é menor que o tamanho máximo permitido, a função aloca um novo nó, copia a palavra para o nó, insere o nó no início da *linked this* e atualiza o contador de entradas da *Hash Table*. Se o número de entradas ultrapassar o tamanho da Hash Table, a função *hash_table_grow* é chamada para redimensionar a tabela. Por fim, o nó é retornado.

Se a palavra não foi encontrada e a opção *insert_if_not_found* é falso ou o tamanho da palavra é maior que o tamanho máximo permitido, a função retorna NULL.

## Função *find_representative()*

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

  // Find the representative element by following the chain of representatives until we reach an element that points to itself
  for (representative = node; representative != representative->representative; representative = representative->representative)
    ;

  // Do a second pass to apply path compression, which flattens the tree representation of the disjoint-set and improves the efficiency of future find operations
  for (next_node = node; next_node != representative; next_node = next_node->representative)
  {
    actual_node = next_node->representative;
    next_node->representative = representative;
  }

  return representative;
}
```

*Figura 5 | Função find_representative()*

Nesta implementação, cada elemento é representado por uma estrutura *hash_table_node_t*, que possui um campo chamado *representative*, que aponta para o elemento representativo de seu subconjunto. O elemento representativo é o representante do componente conexo ao qual pertence. Nesta implementação, para obtê-lo consideramos que seja o elemento que aponta para si mesmo.

Primeiro, através de um *for loop,* percorre os vértices representantes começando do elemento dado e terminando no elemento representativo já explicado acima. Em seguida, através de outro *for loop*, volta a percorrer os mesmos representantes, fazendo com que o campo *representative* de cada um aponte para o representativo calculado inicialmente, em vez de indiretamente através de uma cadeia de outros elementos.

Por fim, a função retorna um ponteiro para o elemento representativo.

## Função *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 *linkfrom, *linkto;

  // Find the word in the hash table
  from_representative = find_representative(from);
  to = find_word(hash_table, word, 0);

  // If the word is not in the hash table, or if the two vertices are the same, return without adding an edge
  if (to == NULL || to == from)
    return;

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

  // If the vertices are already in the same connected component, increment the number of vertices in the component
  if (from_representative == to_representative)
  {
    from_representative->number_of_vertices++;
  }

  // If the vertices are not in the same connected component, merge the two connected components
  if (from_representative != to_representative)
  {
    if (from_representative->number_of_vertices < to_representative->number_of_vertices)
    {
      from_representative->representative = to_representative;
      to_representative->number_of_vertices += from_representative->number_of_vertices;
      to_representative->number_of_edges += from_representative->number_of_edges;
    }
    else
    {
      to_representative->representative = from_representative;
      from_representative->number_of_vertices += to_representative->number_of_vertices;
      from_representative->number_of_edges += to_representative->number_of_edges;
    }
  }
  // Allocate two adjacency nodes to represent the edge between the vertices
  linkfrom = allocate_adjacency_node();
  linkto = allocate_adjacency_node();

  // If allocation fails, print an error message and exit
  if (linkfrom == NULL || linkto == NULL)
  {
    fprintf(stderr, "add_edge: out of memory\n");
    exit(1);
  }

  // Update the linked lists of adjacency nodes for each vertex to include the new nodes
  linkfrom->vertex = to;
  linkfrom->next = from->head;
  from->head = linkfrom;

  linkto->vertex = from;
  linkto->next = to->head;
  to->head = linkto;

  from_representative->number_of_edges++;
  to_representative->number_of_edges++;
  hash_table->number_of_edges++;
  return;
}
```

*Figura 6 | Função add_edge()*

Esta função adiciona uma aresta entre dois vértices em uma estrutura de dados que representa um grafo. A aresta liga o vértice *from* ao vértice com a palavra (*word*) especificada.

Os vértices são representados por elementos da estrutura *hash_table_node_t* e as arestas são representadas por elementos da estrutura *adjacency_node_t*, que contêm um ponteiro para um vértice e um ponteiro para o próximo elemento na lista de adjacências do vértice.

Começaremos por chamar a função *find_representative()* para encontrar os representantes dos vértices *from* e *to*. O vértice to é encontrado chamando a função *find_word()* e passando a palavra (*word*) como parâmetro. Se o vértice to não for encontrado ou se o vértice *from* for igual ao vértice to, a função retorna sem adicionar a aresta.

De seguida, verificaremos se os vértices *from* e to já pertencem à mesma componente conexa. Se o mesmo acontecer, o número de vértices da componente é incrementado. Cajo os vértices não pertencerem à mesma componente, faremos merge das duas componentes conexa, atribuindo o representante de uma componente como o representante da outra. O número de vértices e o número de arestas da componente que recebe o novo representante são atualizados para incluir os valores da que perde o representante.

Alocamos ainda duas estruturas *adjacency_node_t* para representar a aresta entre os vértices. Se a alocação falhar, a função imprime uma mensagem de erro e encerra o programa.

Em seguida, atualizamos as *linked lists* de nós de adjacência de cada vértice para incluir os novos nós. O vértice *from* passa a apontar para o novo nó *linkfrom* e o vértice to passa a apontar para o novo nó *linkto*.

Por fim, é atualizado o número de arestas dos representantes dos vértices *from* e *to* e o número de arestas da *hash table*.

## Função *breadh_first_search()*

```c
static int breadh_first_search(int maximum_number_of_vertices, hash_table_node_t **list_of_vertices, hash_table_node_t *origin, hash_table_node_t *goal)
{
  // Indices for the read and write positions of the queue
  int read = 0, write = 1;
  // Initialize the first position of the queue with the origin vertex [origin, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL]
  list_of_vertices[0] = origin;
  // Initialize the previous vertex of the origin as NULL
  origin->previous = NULL;
  // Mark the origin vertex as visited
  origin->visited = 1;
  int found = 0;
  // While the queue is not empty
  while (read != write)
  {
    // node is set to the head of the origin vertex to begin traversing the linked list
    adjacency_node_t *node = list_of_vertices[read++]->head;
    if (found == 1)
    {
      break;
    }
    while (node != NULL) // Traverse the linked list
    {
      // Check if the vertex has already been visited
      if (node->vertex->visited == 0)
      {
        node->vertex->visited = 1;
        // The previous vertex of this vertex is the vertex before it in the queue
        node->vertex->previous = list_of_vertices[read - 1];
        // Add the vertex to the queue using the write position
        list_of_vertices[write++] = node->vertex;
        if (node->vertex == goal)
        {
          found = 1;
          break;
        }
      }
      node = node->next;
    }
  }
  for (int i = 0; i < write; i++)
  {
    list_of_vertices[i]->visited = 0;
  }
  return write;
}
```

*Figura 7 | Função breadh_first_search()*

Esta função realiza uma busca em largura (*breadth-first search*, BFS) em um grafo a partir de um vértice de origem até um vértice de destino (*goal*). A busca em largura é um algoritmo de busca que percorre todos os vértices de um grafo em uma ordem específica, explorando todos os mesmo de um certo nível antes de passar para o próximo nível. Isso garante que a primeira vez que um vértice é visitado, todos os seus vizinhos mais próximos já foram visitados.

A função começa inicializando alguns valores e declarando algumas variáveis. Em seguida, entra num *loop* que continuará enquanto houver vértices na fila (*queue*) de vértices a serem visitados. O *loop* começa por percorrer a lista de adjacências do vértice atual, marcando cada vértice não visitado como visitado e adicionando-o à fila. Se o vértice atual é o vértice de destino, a função marca a variável *found* como 1 e sai do *loop*. No final do *loop*, a função “reseta” o campo *visited* de todos os vértices para 0 e retorna o número de vértices visitados.

## Função *list_connected_component()*

```c
static int list_connected_component(hash_table_t *hash_table, const char *word, int option)
{
  // Find the given word in the hash table
  hash_table_node_t *node = find_word(hash_table, word, 0);
  int vertices = 0;
  if (node == NULL)
  {
    fprintf(stderr, "Word not found\n");
    return -1;
  }

  // Get the representative node of the connected component containing the word
  hash_table_node_t *representative = find_representative(node);
  // Allocate memory for a list of vertices in the connected component
  hash_table_node_t **list_of_vertices = (hash_table_node_t **)malloc(representative->number_of_vertices * sizeof(hash_table_node_t *));

  // Perform a breadth-first search to find all the vertices in the connected component
  int num_vertices = breadh_first_search(hash_table->number_of_entries, list_of_vertices, node, NULL);
  // Print all the vertices in the connected component
  if (option == 1)
  {
    printf("Connected component:\n");
  }

  for (int i = 0; i < num_vertices; i++)
  {
    vertices++;
    if (option == 1)
    {
      printf("[%d] %s\n", vertices, list_of_vertices[i]->word);
    }
  }

  free(list_of_vertices);

  return vertices;
}
```

*Figura 8 | Função list_connected_component()*

Esta função tem o intuito de apresentar todos os vértices de um componente conexo, ou seja, todas as palavras às quais poderemos chegar a partir de uma originária.

Inicialmente, procuramos o vértice de origem na *hash table* através da função *find_word.* Se o vértice não for encontrado, saímos da função e é impressa uma mensagem de erro. Se mesmo for encontrado, calcularemos o seu representativo usando a função *find_representative* e alocaremos uma lista de vértices suficientemente grande para armazenar todos os vértices do componente conexo.

Através da função *bread_first_serch*, percorreremos o grafo em largura a partir do vértice origem e armazenaremos os vértices visitados na lista alocada já referida.

Por fim, depois de percorrer o grafo, será impressa a lista dos vértices dessa componente conexa assim como o seu número de vértices e arestas.

Nota: O parâmetro de entrada *option* serve apenas como indicar se é para imprimiar ou não as palavras do componente conexo.

## Função *connected_component_diameter()*

```c
static int largestDiameter, smallestDiameter, diametersSum, numDiameters;
static hash_table_node_t **largestDiameterList;
static int maxNumVertices;

static connected_component_diameter(hash_table_node_t *node)
{
  int diameter = -1;
  int numVertices;
  hash_table_node_t **vertices, **temporaryList;

  maxNumVertices = find_representative(node)->number_of_vertices;
  vertices = (hash_table_node_t **)malloc(maxNumVertices * sizeof(hash_table_node_t *));
  temporaryList = (hash_table_node_t **)malloc(maxNumVertices * sizeof(hash_table_node_t *));

  if (vertices == NULL || temporaryList == NULL)
  {
    fprintf(stderr, "connected_component_diameter: out of memory\n");
    exit(1);
  }

  numVertices = breadh_first_search(maxNumVertices, vertices, node, NULL);

  for (int i = 0; i < numVertices; i++)
  {
    int temporaryNumVertices = breadh_first_search(maxNumVertices, temporaryList, vertices[i], vertices[i]);
    int temporaryDiameter = 0;
    hash_table_node_t *palavras = temporaryList[temporaryNumVertices - 1];

    while (palavras != NULL)
    {
      temporaryDiameter++;
      palavras = palavras->previous;
    }

    if (temporaryDiameter > diameter)
    {
      diameter = temporaryDiameter;
    }
  }
  if (diameter > largestDiameter)
  {
    largestDiameter = diameter;

    largestDiameterList = (hash_table_node_t **)malloc(numVertices * sizeof(hash_table_node_t *));

    if (largestDiameterList == NULL)
    {
      fprintf(stderr, "connected_component_diameter: out of memory\n");
      exit(1);
    }

    largestDiameterList = vertices;
  }

  if (diameter < smallestDiameter)
  {
    smallestDiameter = diameter;
  }

  if (diameter == -1)
  {
    printf("connected_component_diameter: diameter not found\n");
    return -1;
  }

  numDiameters++;
  diametersSum += diameter;

  free(temporaryList);
  free(vertices);
}
```

*Figura 9 | Função connected_component_diameter()*

Esta função tem como objetivo calcular o diâmetro de uma componente conexa, ou seja, a distância máxima entre dois vértices dessa componente.

A função possui várias variáveis estáticas, que são compartilhadas por todas as chamadas da função e mantêm seu valor entre essas chamadas. Essas variáveis incluem o diâmetro máximo encontrado até o momento, o diâmetro mínimo encontrado até o momento, a soma de todos os diâmetros encontrados, o número de diâmetros encontrados e uma lista com os vértices que formam o diâmetro máximo encontrado até o momento. Além disso, a função possui uma variável estática chamada *maxNumVertices*, que armazena o número máximo de vértices que a função pode processar.

Em seguida, alocaremos dois *arrays* de ponteiros para nós (*vertices* e *temporaryList*) e chamamos a função *breadh_first_search()* passando o *array* vértices e o nó node como parâmetros. A função *breadh_first_search()* percorre a componente conexa em largura a partir do nó node e coloca todos os nós visitados no *array* vértices. O tamanho do *array* vértices é passado como parâmetro para a função *breadh_first_search()* e é igual a *maxNumVertices*. O valor de retorno da função *breadh_first_search()* é o número de nós colocados no *array vertices*.

De seguida, entraremos num ciclo for que irá percorrer todos os nós do *array* vértices. Para cada nó do *array*, a função *breadh_first_search()* é chamada novamente, desta vez passando *maxNumVertices*, o *array temporaryList* e o nó vertices[i] como parâmetros. O *array temporaryList* irá conter a lista dos nós do caminho mais longo a partir do nó vertices[i].

A variável temporaryDiameter é inicializada 0 e será criado um ponteiro para nós chamado palavras que aponta para o último nó do array temporaryList. Enquanto o ponteiro palavras não for nulo, o código incrementa o valor de *temporaryDiameter* e atualiza o ponteiro palavras para apontar para o nó anterior (*previous*). Se o valor de *temporaryDiameter* for maior do que o valor de *diameter*, o mesmo é atualizado com o valor de *temporaryDiameter*.

Após o ciclo for, para efeitos de apresentação de estatísticas do grafo posteriormente, verificamos se o valor de *diameter* é maior do que o valor de *largestDiameter*. Se for, o valor de *largestDiameter* é atualizado com o valor de *diameter* e o *array* largestDiameterList é atualizado com o array vertices. Caso o valor de *diameter* seja menos que *smallestDiameter*, o anteriormente será alterado. Incrementamos ainda o valor de *numDiameters* e adiciona o valor de *diameter* ao valor de *diametersSum* para calcularmos posteriormente médias.

Por fim, o código liberta os *arrays temporaryList* e vértices da memória e retorna o valor de *diameter*.

## Função *path_finder()*

```c
static void path_finder(hash_table_t *hash_table, const char *from_word, const char *to_word)
{
  // Find the hash table nodes corresponding to the given words.
  hash_table_node_t *from, *fromRepresentative, *to, *toRepresentative;
  from = find_word(hash_table, from_word, 0);
  to = find_word(hash_table, to_word, 0);

  // Find the representatives of the connected components containing the words.
  fromRepresentative = find_representative(from);
  toRepresentative = find_representative(to);

  // If one of the words doesn't exist in the hash table, print an error message and return.
  if (from == NULL || to == NULL)
  {
    printf("One of the words doesn't exist\n");
    return;
  }

  // If the words aren't in the same connected component, print an error message and return.
  if (fromRepresentative != toRepresentative)
  {
    printf("The words aren't in the same connected component\n");
    return;
  }

  // Allocate an array of pointers to hash table nodes to store the vertices of the shortest path.
  hash_table_node_t **vertices = malloc(sizeof(hash_table_node_t *) * fromRepresentative->number_of_vertices);

  // Find the shortest path between the two words using a breadth-first search.
  int path = breadh_first_search(fromRepresentative->number_of_vertices, vertices, to, from);

  // Print the words on the shortest path in order, starting from the destination word and ending at the source word.
  hash_table_node_t *palavras = vertices[path - 1];
  int ordem = 0;
  while (palavras != NULL)
  {
    printf("[%d] %s \n", ordem, palavras->word);
    ordem++;
    palavras = palavras->previous;
  }

  // Free the array of vertices.
  free(vertices);
}
```

*Figura 10 | Função path_finder()*

Esta função encontra o caminho mais curto entre dois vértices em um grafo.

Primeiramente, procuraremos os vértices de origem e destino na *hash table* através da função *find_word().* Se algum dos vértices não for encontrado, sairemos da função e será impressa uma mensagem de erro.

De seguida, encontraremos os elementos representativos dos vértices em questão usando a função *find_representative(),* verificando se estes estão no mesmo componente conexo. Caso isto não se verifique, sairemos da função e é impressa uma mensagem de erro. Caso contrário, alocaremos uma lista de vértices suficientemente grande para armazenar todos os vértices do componente conexo e chamaremos a função *bread_first_search* para percorrer o grafo em largura a partir do vértice destino até ao vértice origem, armazenando os vértices visitados na lista.

Por fim, percorreremos a lista de trás para a frente imprimindo as palavras respetivas ao caminho do vértice origem ao vértice origem.

## Função *find_connected_component_representatives()*

```c
static int find_connected_component_representatives(hash_table_t *hash_table, hash_table_node_t **representatives)
{
  int nrRepresentatives = 0;

  // Find the representatives of each connected component
  for (int i = 0; i < hash_table->hash_table_size; i++)
  {
    for (hash_table_node_t *vertex = hash_table->heads[i]; vertex != NULL; vertex = vertex->next)
    {
      // Find the representative of the connected component
      hash_table_node_t *representative = find_representative(vertex);

      // Add the representative to the array if it has not already been added
      if (!representative->visited)
      {
        representatives[nrRepresentatives++] = representative->word;
        representative->visited = 1;
      }
    }
  }

  // Reset the visited status of all vertices
  for (int i = 0; i < hash_table->hash_table_size; i++)
  {
    for (hash_table_node_t *vertex = hash_table->heads[i]; vertex != NULL; vertex = vertex->next)
    {
      vertex->visited = 0;
    }
  }

  return nrRepresentatives;
}
```

*Figura 11 | Função find_connected_component_representative()*

Criámos ainda esta função com o objetivo de encontrar os representantes de cada componente conectada numa *hash table* e retornar o número de representantes diferentes. Esta será nos útil na descoberta dos diferentes diâmetros (posteriormente explicada na função *graph_info()*)

Inicia por correr todas as posições da *hash table* e, para cada nó encontrado, chama a função *find_representative(vertex)* para encontrar o representante da componente conexo ao qual o nó pertence. O representante é o nó que representa a componente conexa e é o ponto de partida para percorrer todos os nós da mesma componente.

De seguida, verifica se o representante já foi adicionado ao *array* de representantes. Se o mesmo não tiver sido adicionado, é adicionado ao *array* e a sua propriedade *visited* é definida como 1.

Por fim, percorre novamente toda a *hash table* e "reseta" a propriedade *visited* de todos os nós para 0.

O número de representantes encontrados é retornado como resultado da função.

## Função *graph_info()*

```c
static void graph_info(hash_table_t *hash_table)
{
  // Allocate an array to store representatives (one representative per connected component)
  hash_table_node_t **representatives = malloc(sizeof(hash_table_node_t) * hash_table->hash_table_size);

  // Find the representatives for each connected component in the hash table
  int nrRepresentatives = find_connected_component_representatives(hash_table, representatives);

  // Reset the global variables that track the largest and smallest diameters
  largestDiameter = 0;
  smallestDiameter = hash_table->number_of_entries;

  // For each representative node, find the diameter of its connected component
  for (int i = 0; i < nrRepresentatives; i++)
  {
    connected_component_diameter(representatives[i]);
  }

  //calcula  tamanho maximo de componente conexa
  int min = hash_table->number_of_entries;
  int max = 0;
  int total = 0;
  for (int i = 0; i < nrRepresentatives; i++)
  {
    int size = list_connected_component(hash_table, representatives[i]->word,2);
    if (size > max)
    {
      max = size;
    }
    if (size < min)
    {
      min = size;
    }

    total += size;
  }

  // Print out various statistics about the graph
  printf("\nNumber of edges: %u\n", hash_table->number_of_edges);
  printf("Number of vertices: %u\n", hash_table->number_of_entries);
  printf("Number of different representatives: %d\n", numDiameters);
  printf("Largest connected component: %u\n", max);
  printf("Smallest connected component: %u\n", min);
  printf("Average connected component: %.2f\n", (float)total / nrRepresentatives);
  printf("Largest diameter: %d\n", largestDiameter);
  printf("Smallest diameter: %d\n", smallestDiameter);
  printf("Average of diameters: %.2f\n\n", (float)diametersSum / numDiameters);

  // Print an example of a word chain in the largest connected component
  printf("Largest diameter example: \n");

  for (int i = 0; i < largestDiameter+1; i++)
  {
    printf("[%d] %s \n", i, largestDiameterList[i]->word);
  }
  printf("\n");

  free(representatives);
}
```

*Figura 12 | Função graph_info()*

Este código é uma função que imprime algumas informações sobre um grafo armazenado numa *hash table*. Começa por alocar um *array* que armazena os nós representativos de cada componente conexo no grafo. Em seguida, chama uma função chamada *find_connected_component_representatives*, que retorna o número de diferentes nós representativos de cada componente conexo do grafo.

Depois, inicializa as variáveis globais *largestDiameter* e *smallestDiameter* que armazenam o diâmetro maior e o diâmetro mais pequeno dos componentes conexos, respetivamente. Em seguida, itera sobre cada um dos nós representativos e chama a função *connected_component_diameter* para encontrar o diâmetro de cada componente conectado a partir daquele nó.

Adicionalmente, a função *list_connected_component* é chamada para encontrar o tamanho das componentes conexas, encontrando o tamanho máximo, mínimo e médio das mesmas.

Imprime, posteriormente, várias estatísticas sobre o grafo, como o número de arestas, o diâmetro maior e mais pequeno, a soma de todos os diâmetros, o número de diferentes representativos, a média dos diâmetros e tamanho mínimo, máximo e médio das componentes conexas. Por fim, a função imprime um exemplo de uma cadeia de palavras no componente conexo de diâmetro maior.

## Função *hash_table_info()*

```c
static void hash_table_info(hash_table_t *hash_table)
{
  unsigned int total_list_length = 0;
  unsigned int list_length;
  hash_table_node_t *node;
  unsigned int i;

  // Initialize the statistical data
  hash_table->average_list_length = 0.0;
  hash_table->max_list_length = 0;
  hash_table->min_list_length = _max_word_size_;
  hash_table->empty_lists = 0;

  // Iterate through the array of linked list heads and calculate the statistical data
  for (i = 0; i < hash_table->hash_table_size; i++)
  {
    list_length = 0;
    node = hash_table->heads[i];
    while (node != NULL)
    {
      list_length++;
      node = node->next;
    }
    if (list_length > hash_table->max_list_length)
    {
      hash_table->max_list_length = list_length;
    }
    if (list_length < hash_table->min_list_length && list_length != 0)
    {
      hash_table->min_list_length = list_length;
    }
    if (list_length == 0)
    {
      hash_table->empty_lists++;
    }
    total_list_length += list_length;
  }

  // Calculate the average list length
  hash_table->average_list_length = (float)total_list_length / hash_table->hash_table_size;

  // Print the statistical data
  printf("\nNumber of entries: %u\n", hash_table->number_of_entries);
  printf("Hash table size: %u\n", hash_table->hash_table_size);
  printf("Average list length: %.2f\n", hash_table->average_list_length);
  printf("Max list length: %u\n", hash_table->max_list_length);
  printf("Min list length: %u\n", hash_table->min_list_length);
  printf("Number of empty lists: %u\n\n", hash_table->empty_lists);
}
```

*Figura 13 | Função hash_table_info()*

Críamos esta função para apresentar algumas estatísticas sobre a *hash table.* É uma função relativamente simples, começando pelos dados da estrutura *hash_table_t* que armazenam as estatísticas a serem calculadas.

Em seguida, iremos percorrer todos os nós das *linked lists* presentes na *hash table* e iremos contar o número de elementos em cada lista atualizando os respetivos campos da estrutura referida inicialmente.

Por fim, serão impressos todos os dados calculados, sendo estes:

- Tamanho médio das *linked lists;*
- Tamanho máximo de uma *linked list;*
- Tamanho mínimo de uma *linked list;*
- Número de *linked lists* vazias.

# ESTATÍSTICAS DA *HASH TABLE*

## Crescimento

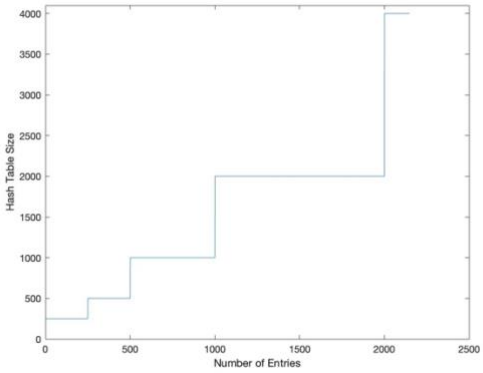



*Figura 15 | Crescimento com ficheiro de 4 letras*

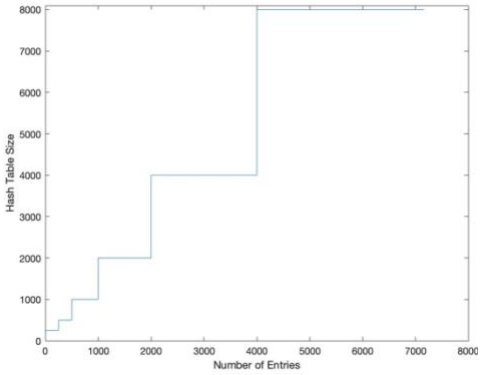



*Figura 16 | Crescimento com ficheiro de 5 letras*

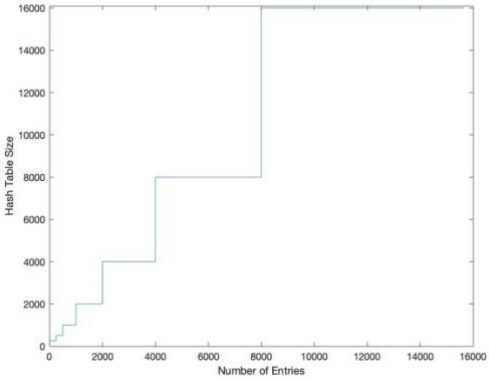



*Figura 14 | Crescimento com ficheiro de 6 letras*

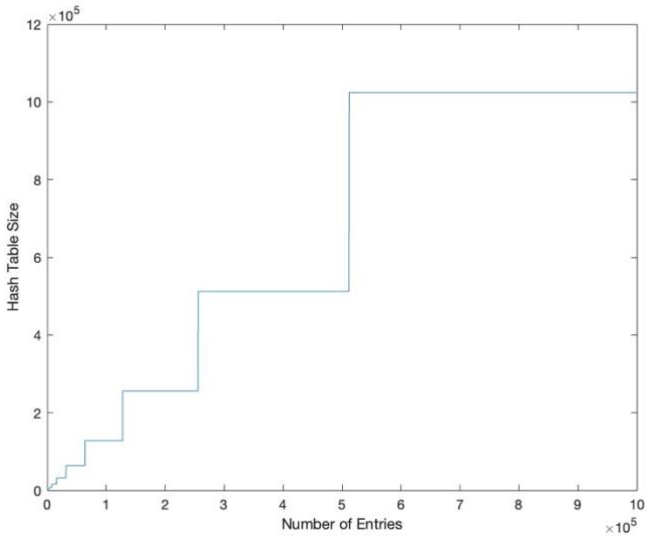



*Figura 17 | Crescimento com ficheiro default*

Estes gráficos têm o intuito de mostrar o crescimento da *hash table* à medida que vão sendo adicionadas palavras. Assim, podemos verificar que, quando o número de palavras colocadas iguala o tamanho da *hash table*, esta aumenta o seu tamanho em dobro.

Estes dados foram passados para um ficheiro no formato ".txt" sempre que era adicionada uma palavra, sendo os três primeiros gráficos correspondentes aos ficheiros mais pequenos e o último correspondente ao ficheiro *default*.

# Atribuições de “lugares”

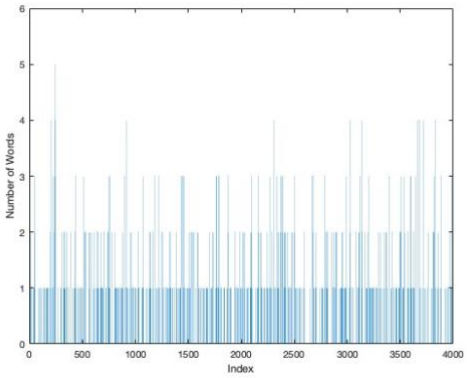



*Figura 20 | Atribuição de "lugares" com ficheiro de 4 letras*

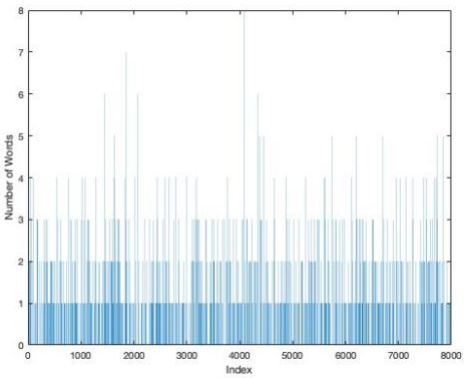



*Figura 19 | Atribuição de "lugares" com ficheiro de 5 letras*

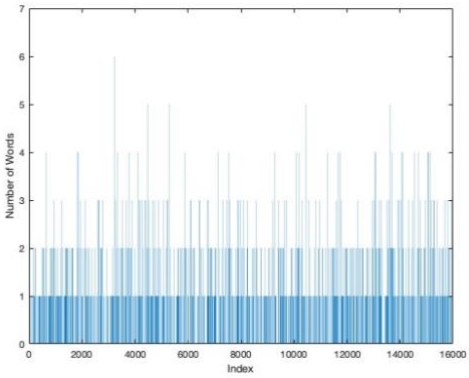



*Figura 18 | Atribuição de "lugares" com ficheiro de 6 letras*

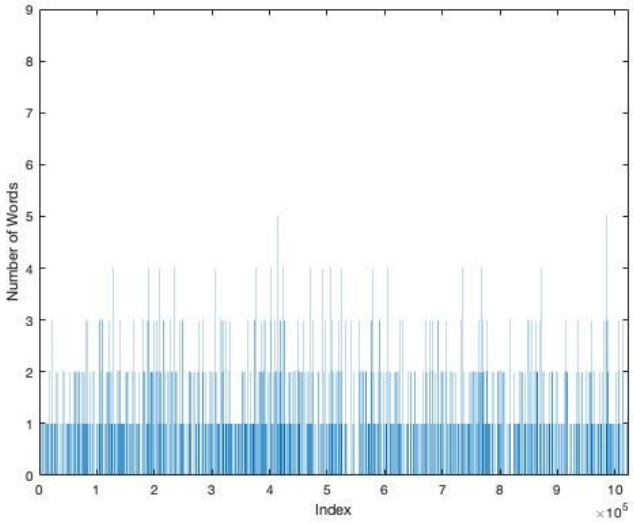



*Figura 21 | Atribuição de "lugares" com ficheiro default*

Estes gráficos têm o intuito de mostrar a distribuição das palavras do ficheiro pela *hash table.*

Conseguimos perceber que há uma distribuição bastante uniforme das palavras, sendo os três primeiros gráficos correspondentes aos ficheiros mais pequenos e o último correspondente ao ficheiro *default*.

# CURIOSIDADES

## Grafos

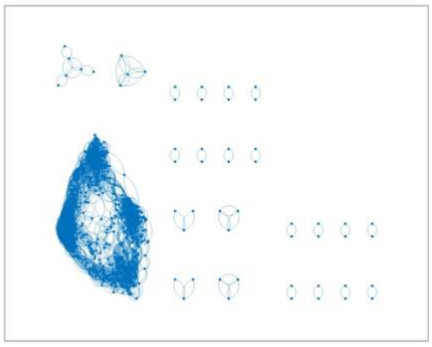


*Figura 22 | Grafo do ficheiro de 4 letras*

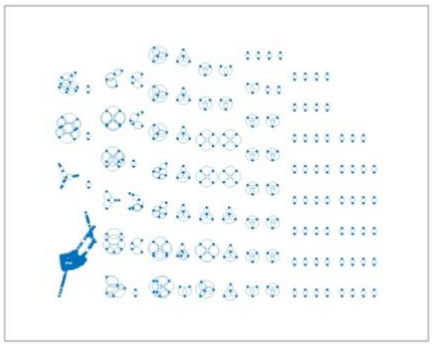


*Figura 22 | Grafo do ficheiro de 5 letras*

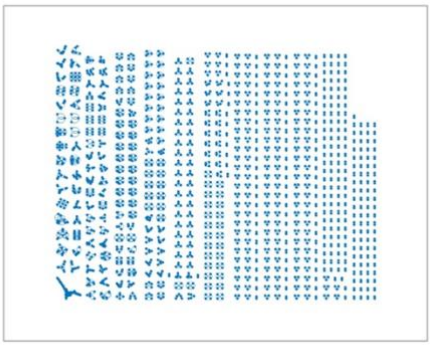


*Figura 24 | Grafo do ficheiro de 6 letras*

Acima estão representados os componentes conexos dos grafos respetivos aos ficheiros mais pequenos.

Não nos foi possível apresentar os referidos componentes respetivos ao ficheiro *default*, pois devido à grande dimensão deste, levará imenso tempo na criação da imagem.

## Informações dos grafos

**Ficheiro de 4 letras:**

- Number of edges: 9267
- Number of vertices: 2149
- Number of different representatives: 187
- Largest connected component: 1931
- Smallest connected component: 1
- Average connected component: 11.49
- Largest diameter: 15
- Smallest diameter: 0
- Average of diameters: 0.22

**Exemplo de maior diâmetro:**

[0] coai
[1] ceai
[2] coas
[3] coar
[4] coam
[5] cozi
[6] cosi
[7] comi
[8] coei
[9] zoai
[10] voai
[11] toai
[12] soai
[13] doai
[14] cear
[15] ceei

**Ficheiro de 5 letras:**

- Number of edges: 23446
- Number of vertices: 7166
- Number of different representatives: 575
- Largest connected component: 6321
- Smallest connected component: 1
- Average connected component: 12.46
- Largest diameter: 33
- Smallest diameter: 0
- Average of diameters: 0.37

**Exemplo de maior diâmetro:**

[0] matar
[1] datar
[2] manar
[3] mamar
[4] malar
[5] matam
[6] Catar
[7] matai
[8] catar
[9] matas
[10] maçar
[11] ratar
[12] datas
[13] dotar
[14] ditar
[15] datam
[16] danar
[17] datai
[18] datal
[19] manas
[20] minar
[21] sanar
[22] panar
[23] nanar
[24] lanar
[25] manam
[26] fanar
[27] manai
[28] mamas
[29] mimar
[30] mamal
[31] gamar
[32] mamam
[33] mamai

**Ficheiro de 6 letras:**

- Number of edges: 36204
- Number of vertices: 15654
- Number of different representatives: 1929
- Largest connected component: 11613
- Smallest connected component: 1
- Average connected component: 8.12
- Largest diameter: 57
- Smallest diameter: 0
- Average of diameters: 0.67

**Exemplo de maior diâmetro:**

[0] voltam
[1] soltam
[2] voltai
[3] voltas
[4] voltar
[5] voltem
[6] volvam
[7] soltai
[8] soltas
[9] soltar
[10] soltem
[11] solvam
[12] saltam
[13] soldam
[14] voltei
[15] voltes
[16] volvas
[17] volvem
[18] voavam
[19] soltei
[20] saltai
[21] soldai
[22] solhas
[23] soldas
[24] saltas
[25] soltos
[26] soltes
[27] solvas
[28] saltar
[29] soldar
[30] solvem
[31] soldem
[32] saltem
[33] soavam
[34] sorvam
[35] salvam
[36] silvam
[37] faltam
[38] salgam
[39] saldam
[40] sondam
[41] toldam
[42] moldam
[43] volvei
[44] volves
[45] valvas
[46] voavas
[47] vulvas
[48] volveu
[49] volver
[50] voaram
[51] coavam
[52] toavam
[53] doavam
[54] zoavam
[55] solvei
[56] soldei
[57] saltei

**Ficheiro *default*:**

- Number of edges: 1060534
- Number of vertices: 999282
- Number of different representatives: 377234
- Largest connected component: 16698
- Smallest connected component: 1
- Average connected component: 2.65
- Largest diameter: 92
- Smallest diameter: 0
- Average of diameters: 0.80

**Exemplo de maior diâmetro:**

[0] barradas
[1] barbadas
[2] barracas
[3] barrados
[4] barravas
[5] barraras
[6] borradas
[7] birradas
[8] berradas
[9] narradas
[10] marradas
[11] carradas
[12] barbados
[13] barbavas
[14] barbaras
[15] barbudas
[16] barracos
[17] barrocas
[18] barricas
[19] barramos
[20] borrados
[21] birrados
[22] berrados
[23] narrados
[24] marrados
[25] barravam
[26] borravas
[27] birravas
[28] berravas
[29] narravas
[30] marravas
[31] barraram
[32] barrarás
[33] barrares
[34] borraras
[35] birraras
[36] berraras
[37] narraras
[38] marraras
[39] borratas
[40] torradas
[41] porradas
[42] jorradas
[43] forradas
[44] bordadas
[45] mirradas
[46] beiradas
[47] serradas
[48] ferradas
[49] cerradas
[50] marcadas
[51] mareadas
[52] carraças
[53] cardadas
[54] cariadas
[55] barbamos
[56] barbudos
[57] barbavam
[58] barbaram
[59] barbarás
[60] barbares
[61] bárbaras
[62] barrocos
[63] barricai
[64] barricar
[65] barricam
[66] barrigas
[67] barrámos
[68] barremos
[69] borramos
[70] birramos
[71] berramos
[72] varramos
[73] narramos
[74] marramos
[75] garramos
[76] torrados
[77] jorrados
[78] forrados
[79] bordados
[80] borrador
[81] mirrados
[82] berrador
[83] beirados
[84] serrados
[85] ferrados
[86] cerrados
[87] narrador
[88] mareados
[89] marcados
90] borravam
[91] birravam
[92] berravam

# Informações das *hash tables*

**Ficheiro de 4 letras:**

- Number of entries: 2149
- Hash table size: 4000
- Average list length: 0.54
- Max list length: 6
- Min list length: 1
- Number of empty lists: 2544

**Ficheiro de 5 letras:**

- Number of entries: 7166
- Hash table size: 8000
- Average list length: 0.90
- Max list length: 8
- Min list length: 1
- Number of empty lists: 3811

**Ficheiro de 6 letras:**

- Number of entries: 15654
- Hash table size: 16000
- Average list length: 0.98
- Max list length: 7
- Min list length: 1
- Number of empty lists: 6203

**Ficheiro *default*:**

- Number of entries: 999282
- Hash table size: 1024000
- Average list length: 0.98
- Max list length: 9
- Min list length: 1
- Number of empty lists: 386313

# Cadeias interessantes de palavras

[0] cadeira
[1] caleira
[2] raleira
[3] raleara
[4] raleada
[5] ralhada
[6] rolhada
[7] rolhado
[8] folhado
[9] folhedo
[10] folheio
[11] folheis
[12] falheis
[13] falíeis
[14] faríeis
[15] feríeis
[16] geríeis
[17] gemíeis
[18] remíeis
[19] regíeis
[20] regreis
[21] regrais
[22] retrais
[23] retraia
[24] retrava
[25] recrava
[26] recuava
[27] recurva
[28] recurvo
[29] recurso

[0] rabo
[1] cabo
[2] caio
[3] cais
[4] caos
[5] cios
[6] aios
[7] aços
[8] açor
[9] apor
[10] opor
[11] odor

[0] palcos
[1] parcos
[2] partos
[3] pastos
[4] castos
[5] cantos
[6] cantor

[0] bem
[1] tem
[2] teu
[3] meu
[4] mau
[5] mal

[0] mãe
[1] mie
[2] fie
[3] fiz
[4] faz
[5] paz
[6] pai

[0] risada
[1] pisada
[2] pirada
[3] parada
[4] para-a
[5] pare-a
[6] pareia
[7] mareia
[8] moreia
[9] morria
[10] sorria
[11] sorrir

[1] chumbar
[2] chumbam
[3] chumbas
[4] chumbai
[5] chumbem
[6] chumbos
[7] chumbes
[8] chumbei
[9] chumbou

[0] cebola
[1] rebola
[2] rebela
[3] rebeca
[4] rabeca
[5] rabeia
[6] rareia
[7] pareia
[8] pareis
[9] partis
[10] partas
[11] cartas
[12] cortas
[13] coutas
[14] chutas
[15] chuvas
[16] chovas
[17] choras
[18] chorar

[0] bebe
[1] bibe
[2] bise
[3] vise
[4] vive
[5] vivi
[6] viii
[7] xiii
[8] xixi

[0] pintor
[1] pintar
[2] pintas
[3] tintas

[0] triste
[1] traste
[2] araste
[3] ataste
[4] ateste
[5] atente
[6] atende
[7] acende
[8] acendi
[9] acenai
[10] acenas
[11] arenas
[12] areias
[13] creias
[14] cheias
[15] checas
[16] chocas
[17] chocar
[18] chorar

[0] bolo
[1] bola
[2] boca
[3] doca
[4] doce

[0] subir
[1] subia
[2] sabia
[3] cabia
[4] caria
[5] carta
[6] casta
[7] cesta
[8] desta
[9] deste
[10] desce

[0] musica
[1] musico
[2] mésico
[3] médico
[4] medico
[5] medi-o
[6] mede-o
[7] mede-a
[8] medeia
[9] meneia
[10] mentia
[11] sentia
[12] sentis
[13] sentas
[14] santas
[15] cantas
[16] cantos
[17] cantor

# Confirmações de *memory leaks*

Após uma confirmação da possível existência de *memory leaks,* concluímos que o programa foi executado com êxito e sem os mesmos, como podemos verificar na imagem abaixo.


*Figura 26 | Execução do programa através do valgrind*

# CONCLUSÃO

Este trabalho, permitiu aumentar os nossos conhecimentos relativamente ao uso de *hash tables* e grafos, que até então nos eram desconhecidos.

Ficou evidente a utilidade das *hash tables* para armazenamento e busca de dados, assim como, a utilidade de grafos para a interligação de diferentes dados com um certo critério de comparação aumentando significativamente a eficiência das funções pretendidas.

# APÊNDICE

## Código c

```
//
// AED, November 2022 (Tomás Oliveira e Silva)
//
// Second practical assignement (word ladder)
//
// Place your student numbers and names here
//   N.Mec. 107637  Name: André Oliveira
//   N.Mec. 107634  Name: Duarte Cruz
//   N.Mec. 107359  Name: Rodrigo Graça
//
// Do as much as you can
//   1) MANDATORY: complete the hash table code
//      *) hash_table_create
//      *) hash_table_grow
//      *) hash_table_free
//      *) find_word
//      +) add code to get some statistical data about the hash table
//   2) HIGHLY RECOMMENDED: build the graph (including union-find data) -- use the similar_words function...
//      *) find_representative
//      *) add_edge
//   3) RECOMMENDED: implement breadth-first search in the graph
//      *) breadh_first_search
//   4) RECOMMENDED: list all words belonginh to a connected component
//      *) breadh_first_search
//      *) list_connected_component
//   5) RECOMMENDED: find the shortest path between to words
//      *) breadh_first_search
//      *) path_finder
//      *) test the smallest path from bem to mal
//         [ 0] bem
//         [ 1] tem
//         [ 2] teu
//         [ 3] meu
//         [ 4] mau
//         [ 5] mal
//      *) find other interesting word ladders
//   6) OPTIONAL: compute the diameter of a connected component and list the longest word chain
//      *) breadh_first_search
//      *) connected_component_diameter
//   7) OPTIONAL: print some statistics about the graph
//      *) graph_info
//   8) OPTIONAL: test for memory leaks
//

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

//
// static configuration
//

#define _max_word_size_ 32

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

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

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

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

struct hash_table_s
{
  unsigned int hash_table_size;   // the size of the hash table array
  unsigned int number_of_entries; // the number of entries in the hash table
  unsigned int number_of_edges;   // number of edges (for information purposes only)
  hash_table_node_t **heads;      // the heads of the linked lists
  float average_list_length;      // the average length of the linked lists
  unsigned int max_list_length;   // the maximum length of a linked list
  unsigned int min_list_length;   // the minimum length of a linked list
  unsigned int empty_lists;       // the number of empty 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 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;
}

//
// 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)
{
  // create a new hash table
  hash_table_t *hash_table;

  // allocate the hash table
  hash_table = (hash_table_t *)malloc(sizeof(hash_table_t));

  // check for allocation errors
  if (hash_table == NULL)
  {
    fprintf(stderr, "create_hash_table: out of memory\n");
    return -1;
  }

  // initialize the hash table
  hash_table->hash_table_size = 250;
  hash_table->number_of_entries = 0;
  hash_table->number_of_edges = 0;

  // allocate the array of linked list heads
  hash_table->heads = (hash_table_node_t **)malloc(hash_table->hash_table_size * sizeof(hash_table_node_t *));

  // check for allocation errors
  if (hash_table->heads == NULL)
  {
    fprintf(stderr, "create_hash_table: out of memory\n");
    return -1;
  }

  // Fill the array of linked list heads with NULL
  for (int i = 0; i < hash_table->hash_table_size; i++)
  {
    hash_table->heads[i] = NULL;
  }

  return hash_table;
}

static void hash_table_grow(hash_table_t *hash_table)
{
  // the old and new array of linked list heads
  hash_table_node_t **old_heads, **new_heads, *node, *next;
  unsigned int old_size, i;

  // save a pointer to the old array of linked list heads and its size
  old_heads = hash_table->heads;
  old_size = hash_table->hash_table_size;

  // create a new hash table with a larger size
  hash_table->hash_table_size *= 2;
  new_heads = (hash_table_node_t **)malloc(hash_table->hash_table_size * sizeof(hash_table_node_t *));

  // Fill the array of linked list heads with NULL
  for (i = 0; i < hash_table->hash_table_size; i++)
    new_heads[i] = NULL;

  if (new_heads == NULL)
  {
    fprintf(stderr, "Error: out of memory");
    exit(1);
  }

  // rehash the entries from the old array to the new one
  for (i = 0; i < old_size; i++)
  {
    node = old_heads[i];
    while (node != NULL)
    {

      next = node->next;

      size_t value = crc32(node->word) % hash_table->hash_table_size;
      node->next = new_heads[value];
      new_heads[value] = node;

      node = next;
    }
  }
  free(old_heads);
  hash_table->heads = new_heads;
}

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

  for (i = 0; i < hash_table->hash_table_size; i++)
  {
    node = hash_table->heads[i];
    while (node != NULL)
    {
      temp = node;
      adj_node = node->head;
      while (adj_node != NULL)
      {
        temp_adj = adj_node;
        adj_node = adj_node->next;
        free(temp_adj);
      }
      node = node->next;
      free(temp);
    }
  }

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

  // Free the memory used by the hash table
  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;

  // find the word in the hash table
  i = crc32(word) % hash_table->hash_table_size;
  node = hash_table->heads[i];
  while (node != NULL)
  {
    if (strcmp(node->word, word) == 0)
      return node;
    node = node->next;
  }

  // if the word is not found, insert it into the hash table
  if (insert_if_not_found && strlen(word) < _max_word_size_)
  {
    node = allocate_hash_table_node();          // allocate a new node
    strncpy(node->word, word, _max_word_size_); // copy the word into the node
    node->representative = node;                // the representative of a node is itself
    node->next = hash_table->heads[i];          // insert the node at the head of the linked list
    node->previous = NULL;
    node->number_of_edges = 0;
    node->number_of_vertices = 1;
    node->visited = 0;
    node->head = NULL;
    hash_table->heads[i] = node;
```

```
    hash_table->number_of_entries++;
    // if the number of entries exceeds the size of the hash table, grow the hash table
    if (hash_table->number_of_entries > hash_table->hash_table_size)
      hash_table_grow(hash_table);
    return node;
  }

  return NULL;
}

//
// add edges to the word ladder graph
//

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

  // Find the representative element by following the chain of representatives until we reach an element that points to itself
  for (representative = node; representative != representative->representative; representative = representative->representative)
    ;

  // Do a second pass to apply path compression, which flattens the tree representation of the disjoint-set and improves the efficiency of future find operations
  for (next_node = node; next_node != representative; next_node = next_node->representative)
  {
    actual_node = next_node->representative;
    next_node->representative = representative;
  }

  return representative;
}

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

  // Find the word in the hash table
  from_representative = find_representative(from);
  to = find_word(hash_table, word, 0);

  // If the word is not in the hash table, or if the two vertices are the same, return without adding an edge
  if (to == NULL || to == from)
    return;

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

  // If the vertices are already in the same connected component, increment the number of vertices in the component
  if (from_representative == to_representative)
  {
    from_representative->number_of_vertices++;
  }

  // If the vertices are not in the same connected component, merge the two connected components
  if (from_representative != to_representative)
  {
    if (from_representative->number_of_vertices < to_representative->number_of_vertices)
    {
      from_representative->representative = to_representative;
      to_representative->number_of_vertices += from_representative->number_of_vertices;
      to_representative->number_of_edges += from_representative->number_of_edges;
    }
    else
    {
      to_representative->representative = from_representative;
      from_representative->number_of_vertices += to_representative->number_of_vertices;
      from_representative->number_of_edges += to_representative->number_of_edges;
    }
  }

  // Allocate two adjacency nodes to represent the edge between the vertices
  linkfrom = allocate_adjacency_node();
  linkto = allocate_adjacency_node();

  // If allocation fails, print an error message and exit
  if (linkfrom == NULL || linkto == NULL)
  {
    fprintf(stderr, "add_edge: out of memory\n");
    exit(1);
  }

  // Update the linked lists of adjacency nodes for each vertex to include the new nodes
  linkfrom->vertex = to;
  linkfrom->next = from->head;
  from->head = linkfrom;

  linkto->vertex = from;
  linkto->next = to->head;
  to->head = linkto;

  from_representative->number_of_edges++;
  to_representative->number_of_edges++;
  hash_table->number_of_edges++;
  return;
}

//
// generates a list of similar words and calls the function add_edge for each one
//
// 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[] =
      {
       0x2D,                                                                                                  // unicode!
       0x41, 0x42, 0x43, 0x44, 0x45, 0x46, 0x47, 0x48, 0x49, 0x4A, 0x4B, 0x4C, 0x4D,                         // - 
       0x4E, 0x4F, 0x50, 0x51, 0x52, 0x53, 0x54, 0x55, 0x56, 0x57, 0x58, 0x59, 0x5A,                         // A B C D E F G H I J K L M
       0x61, 0x62, 0x63, 0x64, 0x65, 0x66, 0x67, 0x68, 0x69, 0x6A, 0x6B, 0x6C, 0x6D,                         // N O P Q R S T U V W X Y Z
       0x6E, 0x6F, 0x70, 0x71, 0x72, 0x73, 0x74, 0x75, 0x76, 0x77, 0x78, 0x79, 0x7A,                         // a b c d e f g h i j k l m
       0xC1, 0xC2, 0xC9, 0xCD, 0xD3, 0xDA,                                                                   // n o p q r s t u v w x y z
       0xE0, 0xE1, 0xE2, 0xE3, 0xE7, 0xE8, 0xE9, 0xEA, 0xED, 0xEE, 0xF3, 0xF4, 0xF5, 0xFA, 0xFC,             // Á Â É Í Ó Ú
       0};                                                                                                   // à á â ã ç è é ê í î ó ô õ ú ü
  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
//
// 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 breadh_first_search(int maximum_number_of_vertices, hash_table_node_t **list_of_vertices, hash_table_node_t *origin, hash_table_node_t *goal)
{
  // Indices for the read and write positions of the queue
  int read = 0, write = 1;
  // Initialize the first position of the queue with the origin vertex [origin, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL]
  list_of_vertices[0] = origin;
  // Initialize the previous vertex of the origin as NULL
  origin->previous = NULL;
  // Mark the origin vertex as visited
  origin->visited = 1;
  int found = 0;
  // While the queue is not empty
  while (read != write)
  {
    // node is set to the head of the origin vertex to begin traversing the linked list
    adjacency_node_t *node = list_of_vertices[read++]->head;
    if (found == 1)
    {
      break;
    }
    while (node != NULL) // Traverse the linked list
    {
      // Check if the vertex has already been visited
      if (node->vertex->visited == 0)
      {
        node->vertex->visited = 1;
        // The previous vertex of this vertex is the vertex before it in the queue
        node->vertex->previous = list_of_vertices[read - 1];
        // Add the vertex to the queue using the write position
        list_of_vertices[write++] = node->vertex;
        if (node->vertex == goal)
        {
          found = 1;
          break;
        }
      }
      node = node->next;
    }
  }
  for (int i = 0; i < write; i++)
  {
    list_of_vertices[i]->visited = 0;
  }
  return write;
}

//
// list all vertices belonging to a connected component
//

static int list_connected_component(hash_table_t *hash_table, const char *word, int option)
{
  // Find the given word in the hash table
  hash_table_node_t *node = find_word(hash_table, word, 0);
  int vertices = 0;
  if (node == NULL)
  {
    fprintf(stderr, "Word not found\n");
    return -1;
  }

  // Get the representative node of the connected component containing the word
  hash_table_node_t *representative = find_representative(node);
  // Allocate memory for a list of vertices in the connected component
  hash_table_node_t **list_of_vertices = (hash_table_node_t **)malloc(representative->number_of_vertices * sizeof(hash_table_node_t *));

  // Perform a breadth-first search to find all the vertices in the connected component
  int num_vertices = breadh_first_search(hash_table->number_of_entries, list_of_vertices, node, NULL);
  // Print all the vertices in the connected component
  if (option == 1)
  {
    printf("Connected component:\n");
  }

  for (int i = 0; i < num_vertices; i++)
  {
    vertices++;
    if (option == 1)
    {
      printf("[%d] %s\n", vertices, list_of_vertices[i]->word);
    }
  }

  free(list_of_vertices);

  return vertices;
}

//
// compute the diameter of a connected component
//

static int largestDiameter, smallestDiameter, diametersSum, numDiameters;
static hash_table_node_t **largestDiameterList;
static int maxNumVertices;

static connected_component_diameter(hash_table_node_t *node)
{
  int diameter = -1;
  int numVertices;
  hash_table_node_t **vertices, **temporaryList;

  maxNumVertices = find_representative(node)->number_of_vertices;
  vertices = (hash_table_node_t **)malloc(maxNumVertices * sizeof(hash_table_node_t *));
  temporaryList = (hash_table_node_t **)malloc(maxNumVertices * sizeof(hash_table_node_t *));

  if (vertices == NULL || temporaryList == NULL)
  {
    fprintf(stderr, "connected_component_diameter: out of memory\n");
    exit(1);
  }

  numVertices = breadh_first_search(maxNumVertices, vertices, node, NULL);

  for (int i = 0; i < numVertices; i++)
  {
    int temporaryNumVertices = breadh_first_search(maxNumVertices, temporaryList, vertices[i], vertices[i]);
    int temporaryDiameter = 0;
    hash_table_node_t *palavras = temporaryList[temporaryNumVertices - 1];

    while (palavras != NULL)
    {
      temporaryDiameter++;
      palavras = palavras->previous;
    }

    if (temporaryDiameter > diameter)
    {
      diameter = temporaryDiameter - 1;
    }
  }

  if (diameter > largestDiameter)
  {
    largestDiameter = diameter;

    largestDiameterList = (hash_table_node_t **)malloc(numVertices * sizeof(hash_table_node_t *));

    if (largestDiameterList == NULL)
    {
      fprintf(stderr, "connected_component_diameter: out of memory\n");
      exit(1);
    }
```

```
    largestDiameterList = vertices;
  }

  if (diameter < smallestDiameter)
  {
    smallestDiameter = diameter;
  }

  if (diameter == -1)
  {
    printf("connected_component_diameter: diameter not found\n");
    return -1;
  }

  numDiameters++;
  diametersSum += diameter;

  free(temporaryList);
  free(vertices);
}

//
// find the shortest path from a given word to another given word
//

static void path_finder(hash_table_t *hash_table, const char *from_word, const char *to_word)
{
  // Find the hash table nodes corresponding to the given words.
  hash_table_node_t *from, *fromRepresentative, *to, *toRepresentative;
  from = find_word(hash_table, from_word, 0);
  to = find_word(hash_table, to_word, 0);

  // Find the representatives of the connected components containing the words.
  fromRepresentative = find_representative(from);
  toRepresentative = find_representative(to);

  // If one of the words doesn't exist in the hash table, print an error message and return.
  if (from == NULL || to == NULL)
  {
    printf("One of the words doesn't exist\n");
    return;
  }

  // If the words aren't in the same connected component, print an error message and return.
  if (fromRepresentative != toRepresentative)
  {
    printf("The words aren't in the same connected component\n");
    return;
  }

  // Allocate an array of pointers to hash table nodes to store the vertices of the shortest path.
  hash_table_node_t **vertices = malloc(sizeof(hash_table_node_t *) * fromRepresentative->number_of_vertices);

  // Find the shortest path between the two words using a breadth-first search.
  int path = breadh_first_search(fromRepresentative->number_of_vertices, vertices, to, from);

  // Print the words on the shortest path in order, starting from the destination word and ending at the source word.
  hash_table_node_t *palavras = vertices[path - 1];
  int ordem = 0;
  while (palavras != NULL)
  {
    printf("[%d] %s \n", ordem, palavras->word);
    ordem++;
    palavras = palavras->previous;
  }

  // Free the array of vertices.
  free(vertices);
}

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

static int find_connected_component_representatives(hash_table_t *hash_table, hash_table_node_t **representatives)
{
  int nrRepresentatives = 0;

  // Find the representatives of each connected component
  for (int i = 0; i < hash_table->hash_table_size; i++)
  {
    for (hash_table_node_t *vertex = hash_table->heads[i]; vertex != NULL; vertex = vertex->next)
    {
      // Find the representative of the connected component
      hash_table_node_t *representative = find_representative(vertex);

      // Add the representative to the array if it has not already been added
      if (!representative->visited)
      {
        representatives[nrRepresentatives++] = representative->word;
        representative->visited = 1;
      }
    }
  }

  // Reset the visited status of all vertices
  for (int i = 0; i < hash_table->hash_table_size; i++)
  {
    for (hash_table_node_t *vertex = hash_table->heads[i]; vertex != NULL; vertex = vertex->next)
    {
      vertex->visited = 0;
    }
  }

  return nrRepresentatives;
}

static void graph_info(hash_table_t *hash_table)
{
  // Allocate an array to store representatives (one representative per connected component)
  hash_table_node_t **representatives = malloc(sizeof(hash_table_node_t) * hash_table->hash_table_size);

  // Find the representatives for each connected component in the hash table
  int nrRepresentatives = find_connected_component_representatives(hash_table, representatives);

  // Reset the global variables that track the largest and smallest diameters
  largestDiameter = 0;
  smallestDiameter = hash_table->number_of_entries;

  // For each representative node, find the diameter of its connected component
  for (int i = 0; i < nrRepresentatives; i++)
  {
    connected_component_diameter(representatives[i]);
  }

  // calcula  tamanho maximo de componente conexa
  int min = hash_table->number_of_entries;
  int max = 0;
  int total = 0;
  for (int i = 0; i < nrRepresentatives; i++)
  {
    int size = list_connected_component(hash_table, representatives[i]->word, 2);
    if (size > max)
    {
      max = size;
    }
    if (size < min)
    {
      min = size;
    }

    total += size;
  }

  // Print out various statistics about the graph
  printf("\nNumber of edges: %u\n", hash_table->number_of_edges);
  printf("Number of vertices: %u\n", hash_table->number_of_entries);
  printf("Number of different representatives: %d\n", numDiameters);
  printf("Largest connected component: %u\n", max);
  printf("Smallest connected component: %u\n", min);
  printf("Average connected component: %.2f\n", (float)total / nrRepresentatives);
  printf("Largest diameter: %d\n", largestDiameter);
  printf("Smallest diameter: %d\n", smallestDiameter);
  printf("Average of diameters: %.2f\n\n", (float)diametersSum / numDiameters);

  // Print an example of a word chain in the largest connected component
  printf("Largest diameter example: \n");

  for (int i = 0; i < largestDiameter + 1; i++)
  {
    printf("[%d] %s \n", i, largestDiameterList[i]->word);
  }
  printf("\n");

  free(representatives);
```

```
}

static void hash_table_info(hash_table_t *hash_table)
{
  unsigned int total_list_length = 0;
  unsigned int list_length;
  hash_table_node_t *node;
  unsigned int i;

  // Initialize the statistical data
  hash_table->average_list_length = 0.0;
  hash_table->max_list_length = 0;
  hash_table->min_list_length = _max_word_size_;
  hash_table->empty_lists = 0;

  // Iterate through the array of linked list heads and calculate the statistical data
  for (i = 0; i < hash_table->hash_table_size; i++)
  {
    list_length = 0;
    node = hash_table->heads[i];
    while (node != NULL)
    {
      list_length++;
      node = node->next;
    }
    if (list_length > hash_table->max_list_length)
    {
      hash_table->max_list_length = list_length;
    }
    if (list_length < hash_table->min_list_length && list_length != 0)
    {
      hash_table->min_list_length = list_length;
    }
    if (list_length == 0)
    {
      hash_table->empty_lists++;
    }
    total_list_length += list_length;
  }

  // Calculate the average list length
  hash_table->average_list_length = (float)total_list_length / hash_table->hash_table_size;

  // Print the statistical data
  printf("\nNumber of entries: %u\n", hash_table->number_of_entries);
  printf("Hash table size: %u\n", hash_table->hash_table_size);
  printf("Average list length: %.2f\n", hash_table->average_list_length);
  printf("Max list length: %u\n", hash_table->max_list_length);
  printf("Min list length: %u\n", hash_table->min_list_length);
  printf("Number of empty lists: %u\n\n", hash_table->empty_lists);
}

//
// 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;
  printf("Inicialização do programa\n");
  // 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);
    // printf("%d %d\n", hash_table->hash_table_size, hash_table->number_of_entries);
  }
  fclose(fp);
  // return 0;

  //   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);
    }
  }

  /* for (unsigned int i = 0; i < hash_table->hash_table_size; i++) // loop through the hash table
  {
    hash_table_node_t *node = hash_table->heads[i]; // set node to the first element of the hash table
    while (node != NULL)                            // while the node has a next node
    {
      hash_table_node_t *temp = node;          // set temp to the node
      node = node->next;                       // set node to the next node                    // free the temp node
      adjacency_node_t *adj_node = temp->head; // set adj_node to the first element of the adjacency list
      while (adj_node != NULL)                 // while the adj_node has a next node
      {
        adjacency_node_t *temp_adj = adj_node;                // set temp_adj to the adj_node
        adj_node = adj_node->next;                            // set adj_node to the next node
        printf("%s %s\n", temp->word, temp_adj->vertex->word); // print the word and the word in the adjacency list")
      }
    }
  }
  return 0; */

  // ask what to do
  for (;;)
  {
    fprintf(stderr, "Your wish is my command:\n");
    fprintf(stderr, "  1 WORD      (list the connected component WORD belongs to)\n");
    fprintf(stderr, "  2 FROM TO   (list the shortest path from FROM to TO)\n");
    fprintf(stderr, "  3           (hash info)\n");
    fprintf(stderr, "  4           (graph info)\n");
    fprintf(stderr, "  5           (print words)\n");
    fprintf(stderr, "  6           (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, 1);
    }
    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)
    {
      hash_table_info(hash_table);
    }
    else if (command == 4)
    {
      graph_info(hash_table);
    }
    else if (command == 5)
    {
      int numPalavras = 0;
      for (i = 0; i < hash_table->hash_table_size; i++)
      {
        for (node = hash_table->heads[i]; node != NULL; node = node->next)
        {
          printf("indice = %u -> %s\n", hash_table->heads[i], node->word);
          numPalavras++;
        }
      }

      printf("Número de palavras = %d\n", numPalavras);
    }
    else if (command == 6)
      break;
    else
    {
      fprintf(stderr, "Invalid command\n");
```

```
    }
  }
  // clean up
  hash_table_free(hash_table);
  return 0;
}
```

## Código Matlab

```
%% Hash Table Size Big

valores = load("hashTableSizeBig.txt");
hashTableSize = valores(:,1);
numEntries = valores(:,2);

plot(numEntries,hashTableSize);
axis([0,1000000,0,1200000]);
ylabel("Hash Table Size");
xlabel("Number of Entries");

%% Gráfico Barras Big

valores = load("histogramaBig.txt");
indexs = valores(:,1);
numWords = valores(:,2);

bar(indexs,numWords);
ylabel("Number of Words");
xlabel("Index");

%% Grafo Big

data = readtable("graphBig.txt");
G = graph;
words = unique(data(:,1));
for i= 1:height(words)
    G = addnode(G,words{i,1});
end
for i= 1:height(data)
    G = addedge(G,data{i,1},data{i,2});
end

plot(G)

%% Hash Table Size Four

valores = load("hashTableSizeFour.txt");
hashTableSize = valores(:,1);
numEntries = valores(:,2);

plot(numEntries,hashTableSize);
axis([0,2500,0,4100]);
ylabel("Hash Table Size");
xlabel("Number of Entries");

%% Gráfico Barras Four

valores = load("histogramaFour.txt");
indexs = valores(:,1);
numWords = valores(:,2);

bar(indexs,numWords);
ylabel("Number of Words");
xlabel("Index");

%% Grafo Four

data = readtable("graphFour.txt");
G = graph;
words = unique(data(:,1));
for i= 1:height(words)
```

```
    G = addnode(G,words{i,1});
end
for i= 1:height(data)
    G = addedge(G,data{i,1},data{i,2});
end

plot(G)

%% Hash Table Size Five

valores = load("hashTableSizeFive.txt");
hashTableSize = valores(:,1);
numEntries = valores(:,2);

plot(numEntries,hashTableSize);
axis([0,8000,0,8100]);
ylabel("Hash Table Size");
xlabel("Number of Entries");

%% Gráfico Barras Five

valores = load("histogramaFive.txt");
indexs = valores(:,1);
numWords = valores(:,2);

bar(indexs,numWords);
ylabel("Number of Words");
xlabel("Index");

%% Grafo Five

data = readtable("graphFive.txt");
G = graph;
words = unique(data(:,1));
for i= 1:height(words)
    G = addnode(G,words{i,1});
end
for i= 1:height(data)
    G = addedge(G,data{i,1},data{i,2});
end

plot(G)

%% Hash Table Size Six

valores = load("hashTableSizeSix.txt");
hashTableSize = valores(:,1);
numEntries = valores(:,2);

plot(numEntries,hashTableSize);
axis([0,16000,0,16100]);
ylabel("Hash Table Size");
xlabel("Number of Entries");

%% Gráfico Barras Six

valores = load("histogramaSix.txt");
indexs = valores(:,1);
numWords = valores(:,2);

bar(indexs,numWords);
ylabel("Number of Words");
xlabel("Index");

%% Grafo Six
```

```
data = readtable("graphSix.txt");
G = graph;
words = unique(data(:,1));
for i= 1:height(words)
    G = addnode(G,words{i,1});
end
for i= 1:height(data)
    G = addedge(G,data{i,1},data{i,2});
end

plot(G)
```

# *Printscreens* resultados mostrados anteriormente

### Ficheiro 4 letras

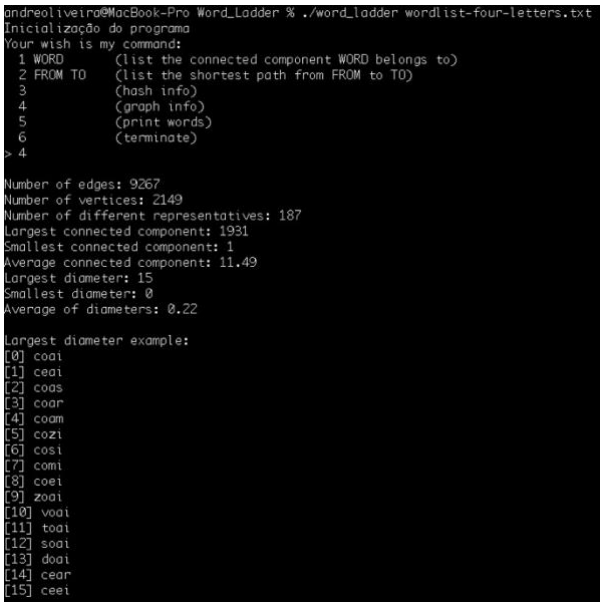



*Figura 27 | Graph Info 4 letras*

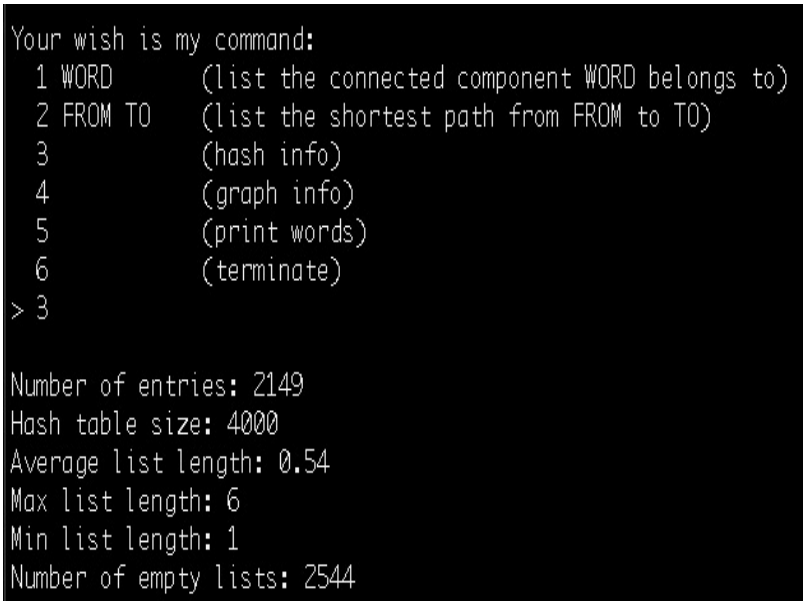



*Figura 28 | Hash Info 4 letras*

### Ficheiro 5 letras

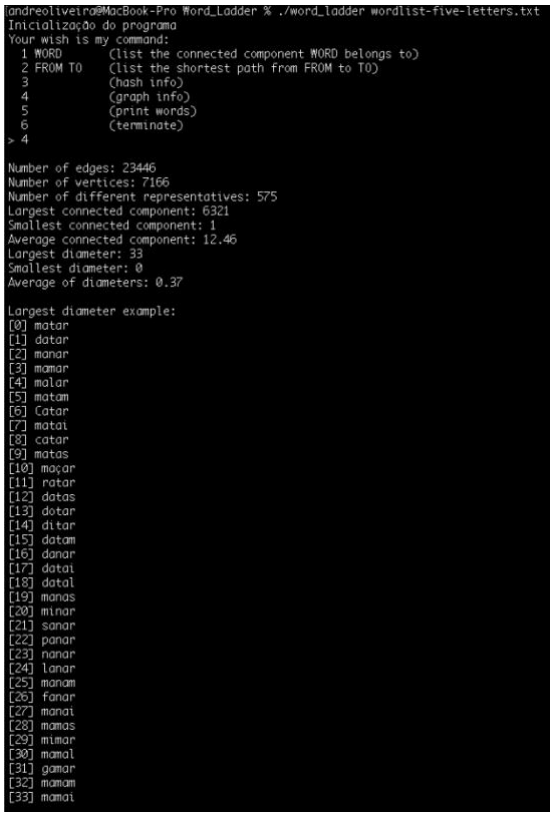



*Figura 29 | Graph Info 5 letras*

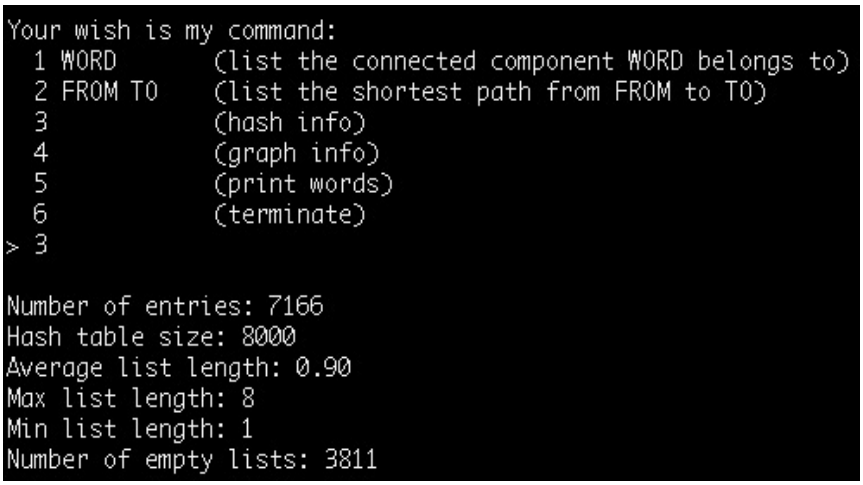



*Figura 30 | Hash Info 5 letras*

## Ficheiro 6 letras

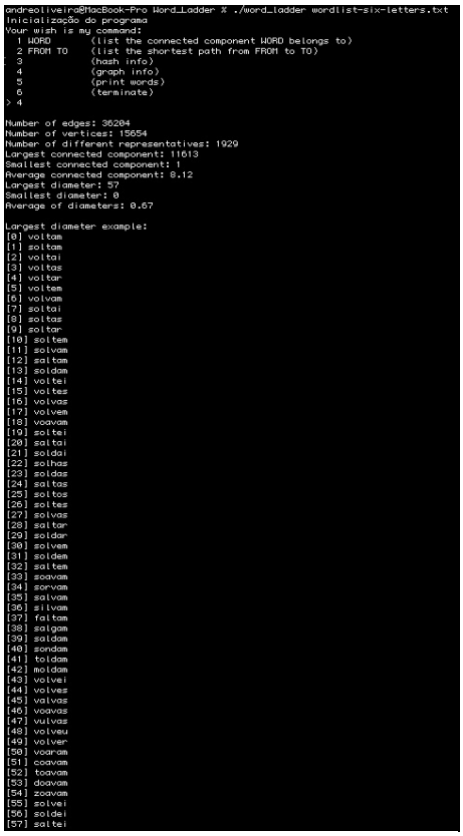



*Figura 31 | Graph Info 6 letras*

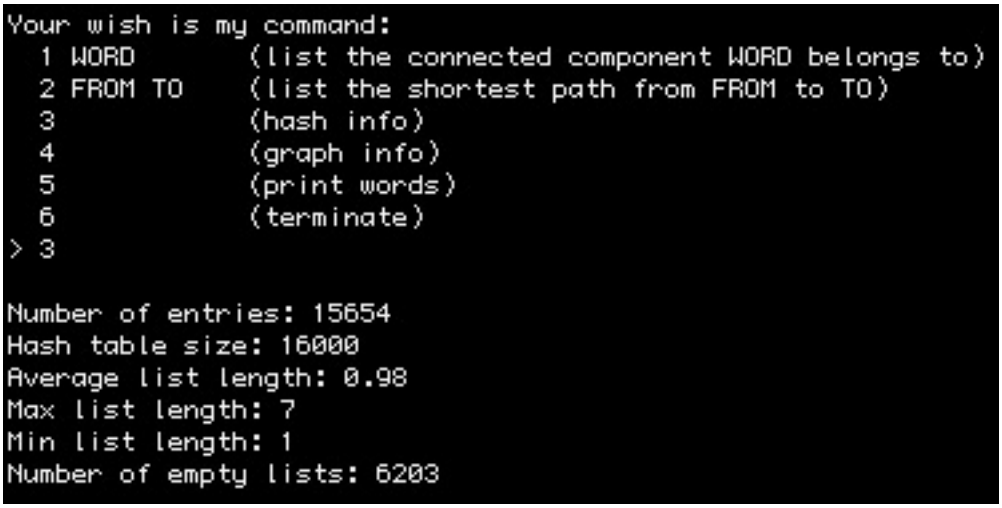



*Figura 32 | Hash Info 6 letras*

## Ficheiro *default*

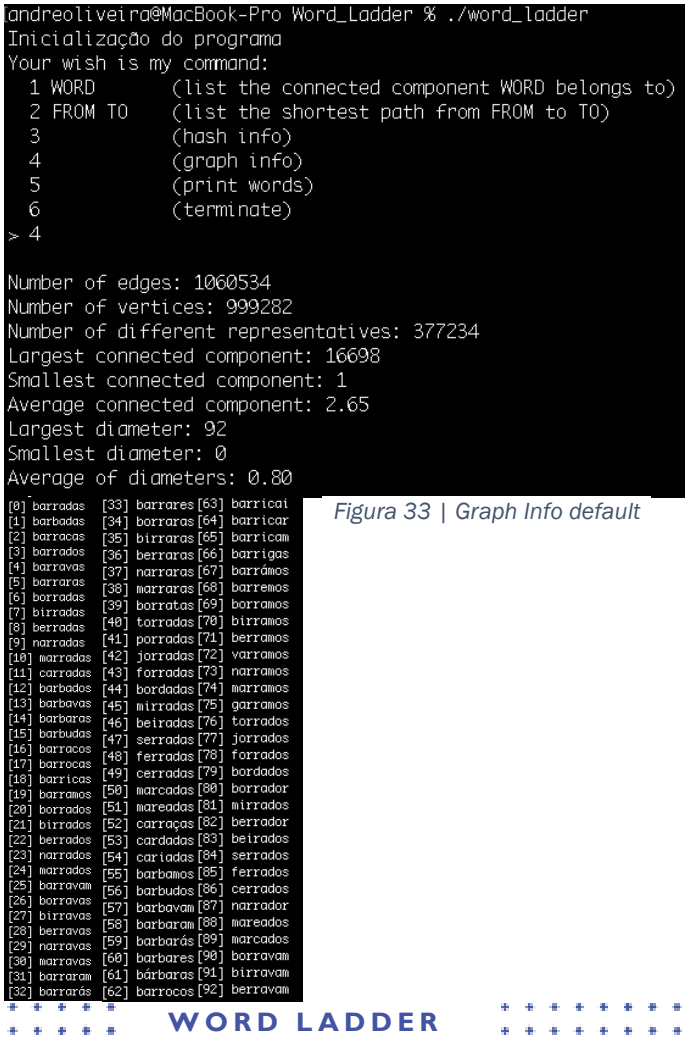



*Figura 33 | Graph Info default*

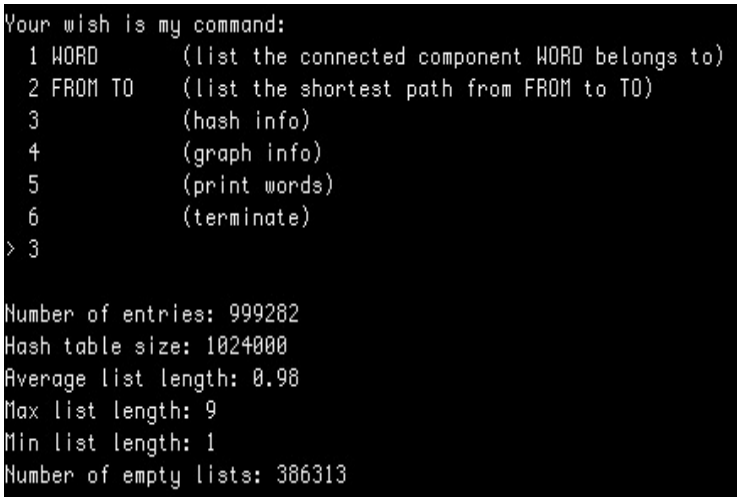



*Figura 34 | Hash Info default*