Word Leader

ALGORITMOS E ESTRUTURAS DE DADOS

Trabalho realizado por:

Alexandre Cotorobai – 107849 (55%)

Vitalie Bologa – 107854 (45%)

universidade de aveiro

2022/2023

Índice

Introdução

No âmbito da unidade curricular de AED (Algoritmo e Estruturas de Dados), foi-nos proposto um trabalho prático que consiste num *script* que irá processar uma lista de palavras.

Esta proposta está dividida em 2 tarefas principais, a primeira baseia-se em implementar uma *hash table* para as palavras recebidas, a segunda envolve criar grafos juntando as palavras que apenas diferem em uma letra.

Neste relatório será demonstrado o funcionamento de cada função desenvolvida, bem como a estrutura de pensamento dos algoritmos usados.

No final iremos mostrar vários dados e informações importantes que conseguimos obter através das funções criadas, tanto sobre a *hash table* como do grafo. Alguns testes também foram executados para mostrar na prática o funcionamento do programa.

Métodos

hash_table_create()

Começando pela primeira função deste programa, inicialmente é alocado memória para o campo *heads*. Caso não haja memória suficiente, será impresso uma mensagem de erro no terminal e sairá do programa

Definimos o campo *hash_table_size* com valor 50, escolheu-se esse número pois é um valor não muito grande (o que não tomaria partido da função *hash_table_grow*), nem muito pequeno (para não usar essa mesma função logo de início desnecessariamente).

O campo *number_of_entries* e *number_of_edges* são definidos como 0, são variáveis de iniciação, que posteriormente quando forem adicionados elementos à *hash_table*, sofrerão alterações.

Por fim, temos de iniciar todos os *heads* da *hash table* com ponteiros NULL.

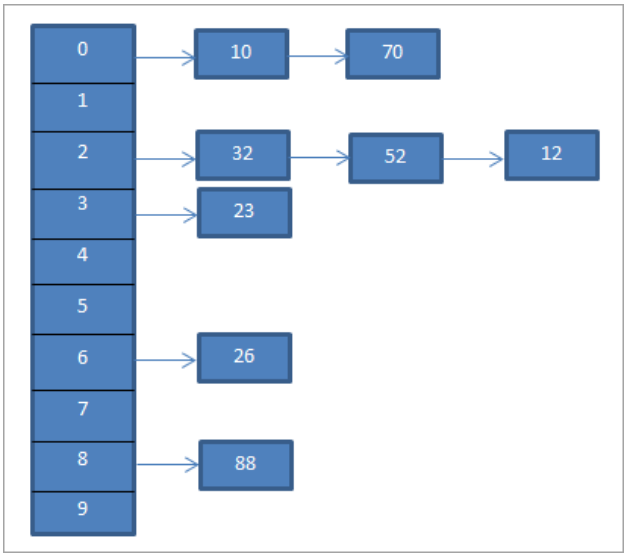

Figura 1 – Exemplo de hash_table

hash_table_grow()

Esta função é responsável por aumentar o tamanho de um *hash_table*, redimensionando a sua matriz para um tamanho maior.

Primeiramente, declaramos dois ponteiros *hash_table_node_t* denominados por *old_heads* e *new_heads*. O *old_heads* é usado para guardar os ponteiros direcionados para o *heads* atual, durante o redimensionamento.

Posteriormente, é alocado um novo *array* de ponteiros (inicializados a NULL) atribuído ao *new_heads* que passará a ser o *heads* da nova *hash table* após o redimensionamento.

Decidimos duplicar o tamanho da tabela sempre que a função é usada, para controlar o número de vezes que será chamada (aumentos menores iriam exigir mais redimensionamentos, aumentos maiores iriam reduzir o número de redimensionamentos).

Por fim, libertamos o *old_heads*, pois não será mais necessário usar esse conteúdo guardado na memória.

hash_table_free()

Esta função tem a responsabilidade de libertar toda a memória alocada pela *hash_table* e os seus nós.

A libertação da memória tem de ser feita de forma hierárquica na *struct* de modo a não eliminar campos que posteriormente não poderemos acessar e consequentemente não libertar da memória.

Por isso começamos, por percorrer todos os elementos das *linked list* e libertamos da memória cada nó da *adjacency list* para esse mesmo nó. Depois serão libertados cada nó da *linked list* e por fim é libertada a *head* de cada *linked_list* e a própria *hash_table*.

find_word()

Esta função procura uma palavra na *hash table*.

Assim que recebemos a palavra é aplicada a função de *hash*, descobrimos o seu índice na *hash table* e vamos procurar na *linked list* desse índice se o nó lá existe.

Para isso, vamos percorrer a *linked_list* do índice encontrado da *hash_table* à procura da palavra. Se a palavra for encontrada, a função retorna o nó que a contém, se não for encontrada e o terceiro argumento da função (*insert_if_not_found*) for 1, é criado um novo nó usando a função *allocate_hash_table_node*, inicializado os seus campos e inserido no início da *linked_list*.

Tendo conta que foi adicionado um novo elemento à *hash_table* o campo *number_of_entries* é incrementado, define o *representative* da *connected component* para o próprio nó (pois neste momento ainda não unimos os diversos nós de modo a formar as componentes conexas, será útil em funções mais à frente descritas), define o número de vértices da *connected component* como 1 e define o número de arestas a 0.

Por fim é verificado se a *hash_table* atingiu o fator carga de 0.75,ou seja, se o número de elementos introduzidos ultrapassa 75% da *hash_table->size*. Caso sim, é chamado a função *hash_table_grow* para aumentar o tamanho da *hash_table*, com intuito de diminuir o número de colisões e assim, melhorar a velocidade da navegação na *hash_table*.

A função, dependendo das circunstâncias, retorna o nó que contém a palavra, o nó recém criado ou NULL.

find_representative()

O *find_representative* é usado para encontrar o representante do *connected component* do vértice.

Tendo em conta os dados da *union-find*, cada nó do grafo está associado a um nó "representativo", que representa a *connected component* à qual o nó pertence.

A função usa um *hash_table_node_t* como argumento e retorna um ponteiro para o nó representativo do nó (de entrada), seguindo a cadeia de ponteiros "representativos" do nó de entrada até a raiz (ou seja, o nó que aponta para si mesmo como representante). Em seguida, ele atualiza o representativo de todos os nós no caminho do nó de entrada até a raiz.

add_edge()

Esta função tem como objetivo de adicionar uma aresta entre dois vértices do grafo. Os vértices são representados pelos ponteiros *from* e *to*.

Em primeiro lugar a função chama a função *find_word* para obter um ponteiro para o vértice *to*. Se o vértice *to* não for encontrado ou for igual ao vértice *from*, a função retorna sem adicionar uma aresta.

Caso a função encontre os representantes dos vértices *from* e *to* e os mesmos forem diferentes, é atualizado o representativo da componente menor, passando a apontar para o representativo da componente maior e atualizado o número de vértices e de arestas da componente maior (este passa a incluir a componente menor que reseta as suas informações de vértices e arestas).

Se os representantes forem iguais, os dois vértices serão ligados por uma aresta (*edges++*).

Por fim, cada vértice adiciona o outro à sua adjacency_list para representar a nova aresta, pois os mesmos são vizinhos.

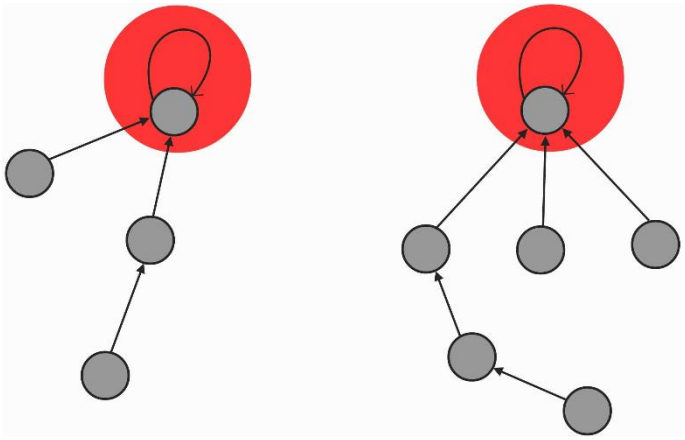

Figura 2 – Formação de uma edge (Parte I)

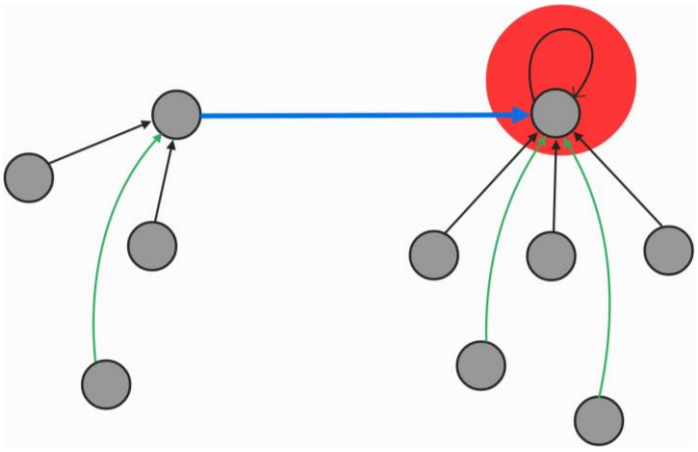

Figura 3 – Formação de uma edge (Parte II)

breadh_first_search()

Esta função é um algoritmo que tem como objetivo retornar o número de vértices visitados até ao nó destino.

Atendendo ao algoritmo, a função irá percorrer todos os vértices alcançáveis a partir do vértice origem com uma certa distância (raio), e seguirá para o próximo nível de distância, consequentemente até alcançar o nó destino.

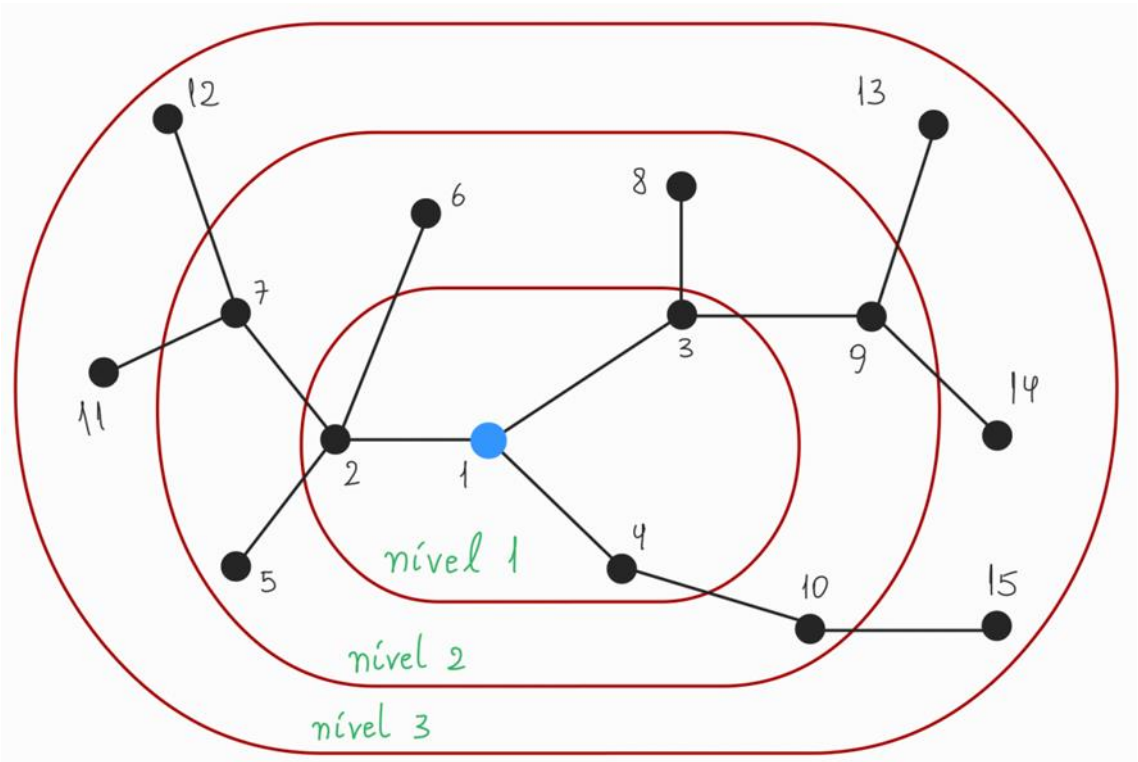

Figura 4 – breadh_first_search

Com isso em mente, (atribuindo a origem ao primeiro elemento da *list_of_vertices*) criamos um *loop* que usa duas variáveis, *r* e *w* para explorar os vértices. O *r* lê consecutivamente os vértices da *list_of_vertices* e o *w* por cada vértice lido são adicionados os vértices adjacentes, que não foram visitados.

A procura termina caso a variável *w* encontre o vértice destino, sendo acionado o operador *stop* ou quando os controladores *r* e *w* chegam à mesma casa (índice da *list_of_vertices*).

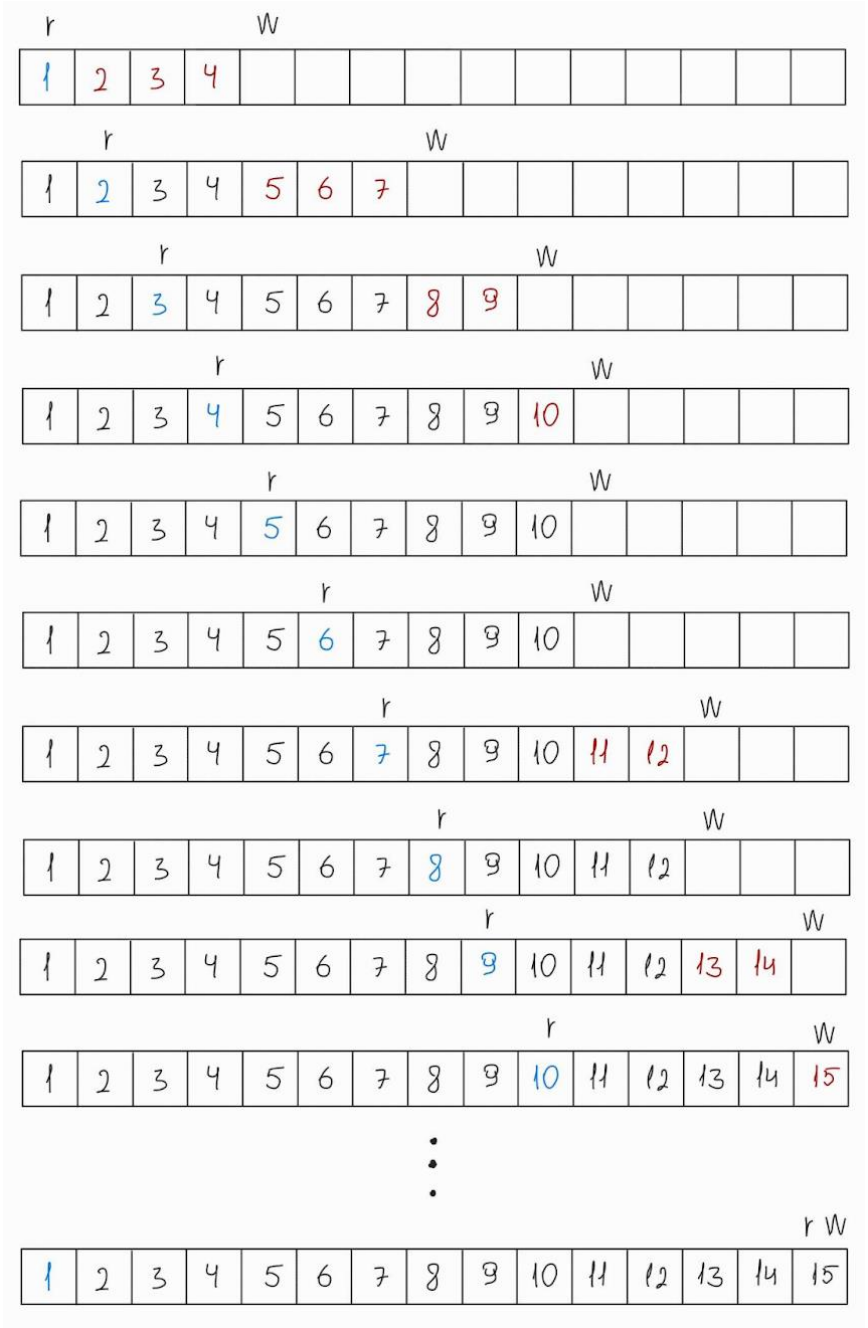

Figura 5 – Vértices visitados

No final a *flag visited* é definida como 0, para todos os vértices, de forma a limpar as alterações efetuadas, permitindo executar novas BFS.

Quando o destino é NULL o BFS vai percorrer todos os vértices da componente conexa, devolvendo assim todos os elementos da componente e o tamanho da mesma.

path_finder()

A função tem o objetivo de encontrar o caminho mais curto entre dois vértices de uma *connected component*.

Em primeiro lugar é chamado a função *find_word* para obter ponteiros para os vértices *from* e *to*.

Com o uso do BFS, a mesma vai criar uma lista com vértices visitados até encontrar o vértice destino, assim que o encontrar começamos a contar desse vértice para trás (*previous*) até à origem, formando o caminho mais curto inverso ao mesmo tempo que conta o número de elementos desse caminho.

Em seguida, armazena os vértices do caminho num *array* e imprime o caminho e o seu comprimento.

Por fim, caso o comprimento do caminho mais curto seja maior que o *largest_diameter* o valor de *largest_diameter* e do *largest_diameter_example* são atualizados.

list_connected_componente()

A função lista todos os vértices pertencentes a uma *connected component*.

Inicialmente é obtido o número de vértices da componente conexa e é alocado espaço para um *array*, com espaço suficiente para conter todos os vértices na componente conexa.

Através do BFS conseguimos obter todos os vértices dessa componente e retornado o número de elementos da mesma.

Finalmente é impresso a lista desses vértices e o seu tamanho.

connected_component_diameter()

Esta função calcula o diâmetro de uma *connected component*.

O diâmetro é definido como o comprimento do caminho mais curto entre os dois vértices mais afastados.

Para fazer isso, a função executa a BFS para gerar uma lista de todos os vértices da componente conexa (lista inicial). Após isso, para cada um desses vértices iremos formar uma nova lista de todos os elementos da componente (lista temporária), como o BFS pesquisa por níveis, o último elemento da lista (temporária) será o vértice mais afastado da origem. Descobrimos assim o vértice mais afastado de cada um dos vértices origem (da lista inicial) e o comprimento entre os dois.

O comprimento desse caminho é comparado ao diâmetro máximo atual e, se for maior, passa a ser o novo máximo.

Também é guardado o *largest path* caso o diâmetro seja maior que o *largest_diameter*, isto será utilizado no *graph_info*.

Por fim, retorna o diâmetro da *connected component*.

graph_info()

Nesta função iremos obter dados relativos aos grafos e às suas componentes.

Provavelmente a informação mais complexa de se obter foi o número de componentes conexas, onde tivemos de percorrer todos os nós para descobrir os representativos que existiam no grafo (os nós cujos representativos já são conhecidos são ignorados de modo a agilizar o processo). Os restantes dados foram obtidos quase diretamente.

hash_table_info()

Esta função será responsável pela coleta de dados sobre a *hash table*, entre esses dados temos o *size*, *number of entries*, *load factor* e também a *longest*, *shortest* e *average chain sizes* que são calculados percorrendo todos os nós da *hash table*.

Esta informação será bastante útil para fazer alguns dos gráficos apresentados abaixo.

Testes e Resultados

Hash table distribution

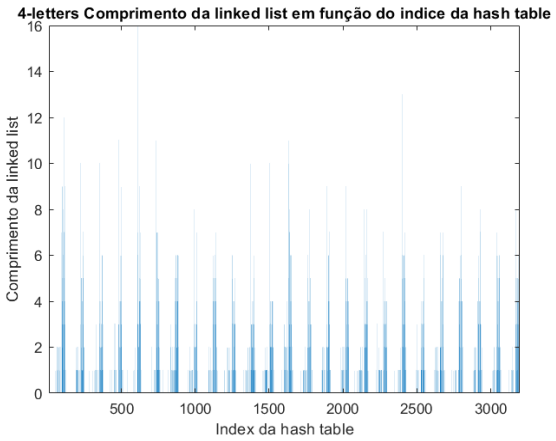

Figura 6 – hash table distribution 4 letters

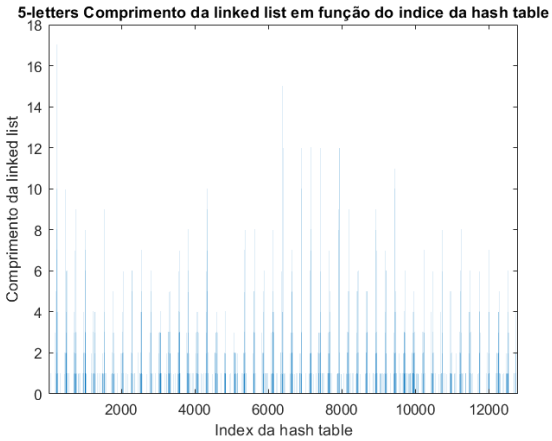

Figura 7 – hash table distribution 5 letters

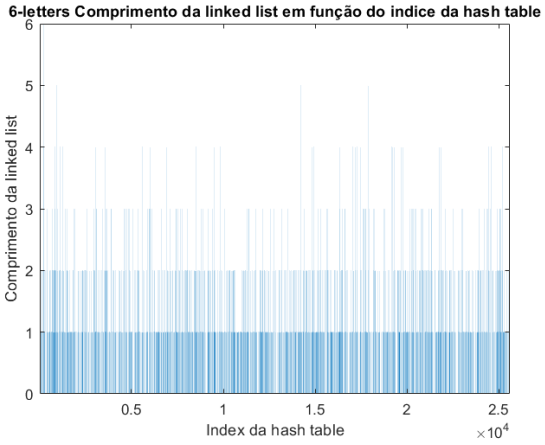

Figura 8 - hash table distribution 6 letters

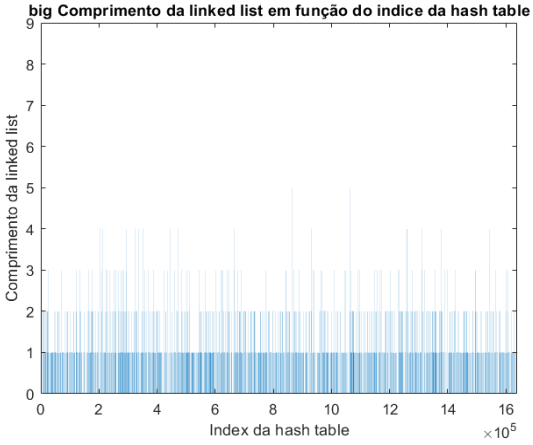

Figura 9 – hash table distribution big

Com estes gráficos conseguimos perceber a distribuição das palavras pelos *heads* da *hash table*. É de se notar que existem listas vazias, mas que as existentes se encontram bastante bem distribuídas por toda a *hash table*.

Hash table linked list sizes

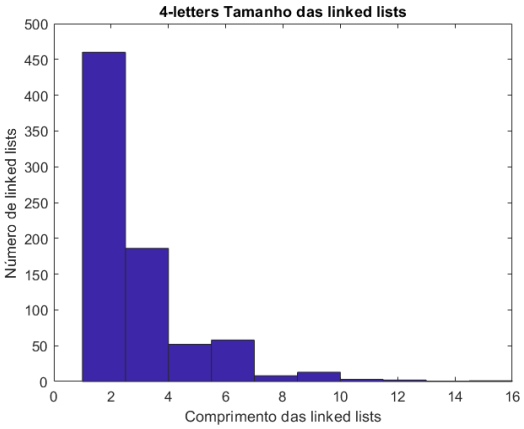

Figura 10 - linked list sizes 4 letters

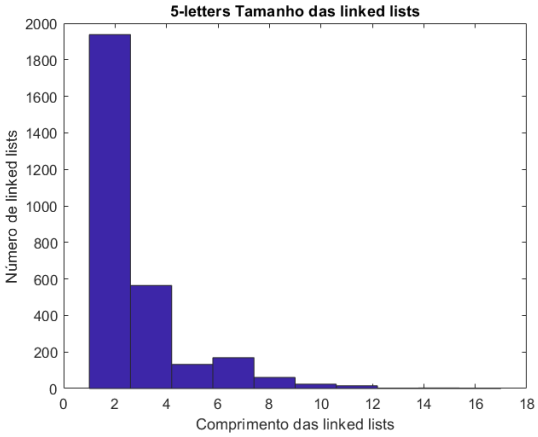

Figura 11 - linked list sizes 5 letters

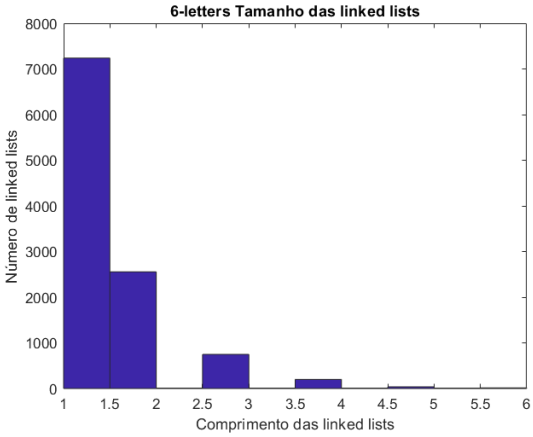

Figura 12 - linked list sizes 6 letters

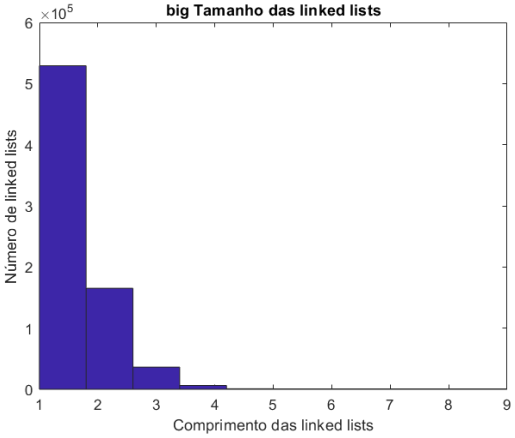

Figura 13 - linked list sizes big

Nestes quatro histogramas é possível verificar a distribuição da informação da *hash table*, aqui conseguimos ver que a maioria das *linked lists* possui apenas um ou dois nós, ou seja, a sua navegação será eficiente. Provamos assim uma excelente dispersão sem sobrecarga em nenhuma zona da *hash table*.

Load factor

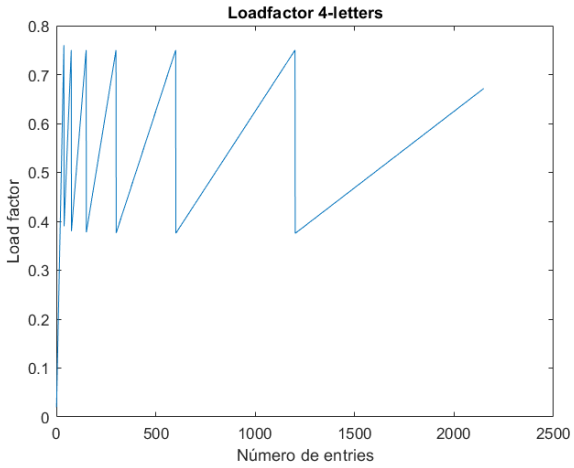

Figura 15 - load factor 4 letters

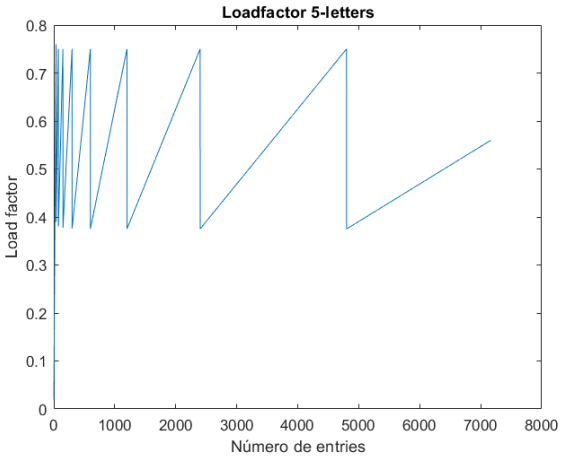

Figura 14 - load factor 5 letters

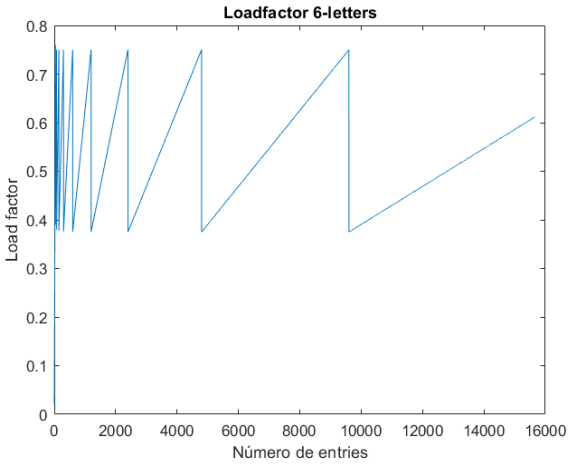

Figura 16 - load factor 6 letters

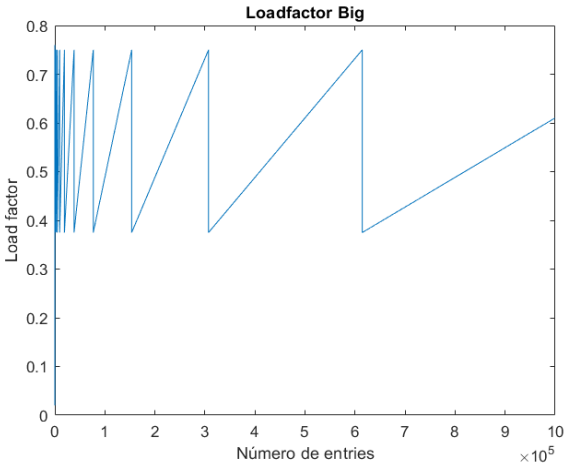

Figura 17 - load factor big

Através destes gráficos conseguimos perceber o efeito da função *grow* conforme vão-se adicionando as palavras à *hash table*. De modo a não sobrecarregar a mesma, sempre que o rácio *number_of_entries*/*hash_table_size* chega a 0.75 a *hash table* sofre *resize* e aumenta de tamanho, assim conseguimos diminuir ao máximo o número de colisões e aumentar a eficácia.

Graphs

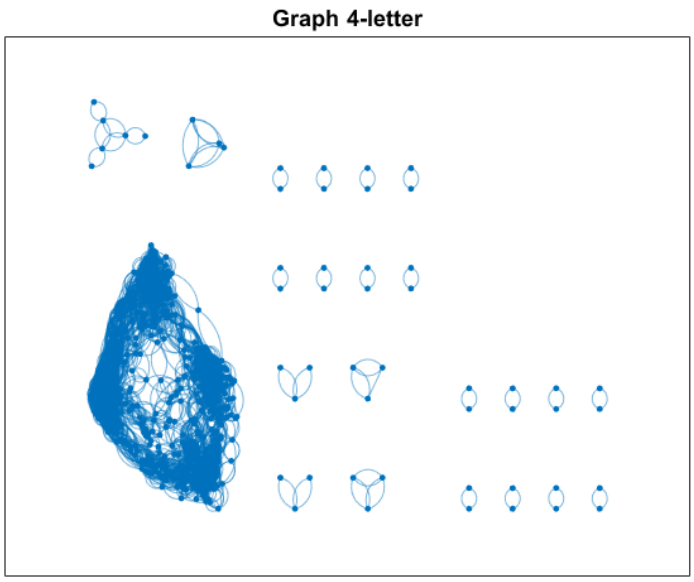

Figura 18 - graph 4 letters

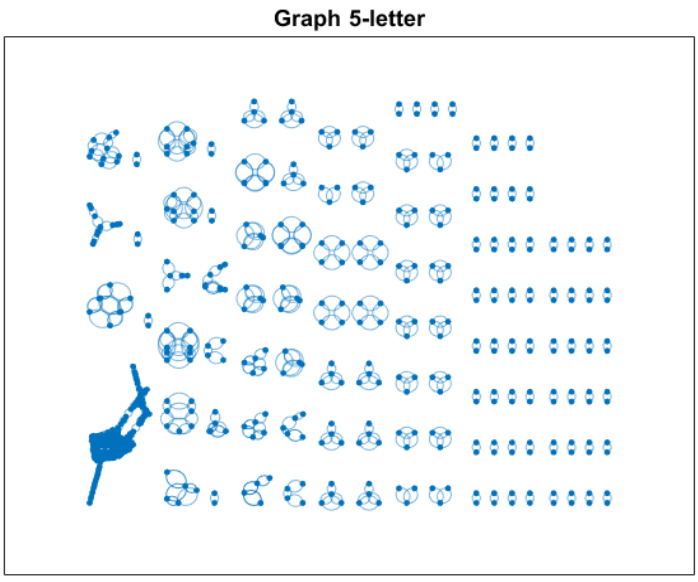

Figura 19 - graph 5 letters

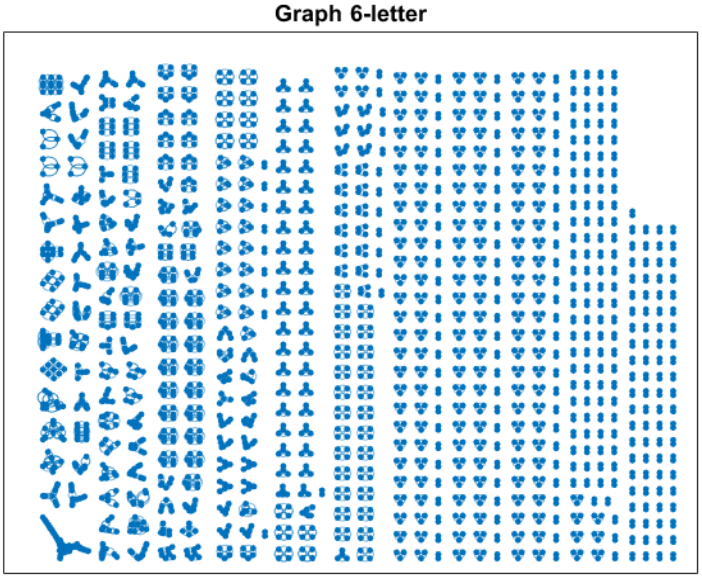

Figura 20 - graph 6 letters

Com estes grafos pretendemos demonstrar visualmente a distribuição e tamanho das *connected components* de cada conjunto de palavras.

Podemos reparar que em cada um existe sempre uma componente bastante maior que as outras, sendo as componentes menores mais comuns.

Há que notar que apesar de termos os dados para fazer o grafo da *wordlist-big*, escolhemos não o fazer pois o tempo que demoraria para obter algum resultado seria exorbitantemente grande.

Hash_table info

De seguida iremos apresentar alguns dados relativos à *hash table* de cada conjunto de palavras.

Para o conjunto de 4 letras
Hash table size: 3200
Hash table number of entries: 2149
Hash table load factor: 0.672
Hash table longest chain size: 16
Hash table shortest chain size: 1
Hash table average chain size: 2.745

Para o conjunto de 5 letras
Hash table size: 12800
Hash table number of entries: 7166
Hash table load factor: 0.560
Hash table longest chain size: 17
Hash table shortest chain size: 1
Hash table average chain size: 2.464

Para o conjunto de 6 letras
Hash table size: 25600
Hash table number of entries: 15654
Hash table load factor: 0.611
Hash table longest chain size: 6
Hash table shortest chain size: 1
Hash table average chain size: 1.451

Para o conjunto big
Hash table size: 1638400
Hash table number of entries: 999282
Hash table load factor: 0.610
Hash table longest chain size: 9
Hash table shortest chain size: 1
Hash table average chain size: 1.354

Hash table	4 letras	5 letras	6 letras	Big
Size	3200	12800	25600	1638400
Number of entries	2149	7166	15654	999282
Load factor	0.672	0.560	0.611	0.610
Longest chain size	16	17	6	9
Shortest chain size	1	1	1	1
Average chain size	2.745	2.464	1.451	1.354

Graphs info

Assim como foi feito para a *hash table*, também retiramos bastantes dados sobre cada grafo.

Graph Info		4 letras	5 letras	6 letras	Big
Graph Data	Number of nodes	2149	7166	15654	999282
	Number of edges	9267	23446	36204	1060534
	Número de componentes conexas	187	575	1929	377234
Component Data	Tamanho médio das componentes conexas	11.4920	12.4626	8.1151	2.6490
	Tamanho máximo das componentes conexas	1931	6321	11613	16698
	Tamanho mínimo das componentes conexas	1	1	1	1
Diameter Data	Diametro medio	0.2193	0.3652	0.6667	0.7984
	Diametro máximo	15	33	57	92
	Diametro mínimo	0	0	0	0

----------*Largest Path 4-letters* ----------
0 – xixi
1 – xiii
2 – viii
3 – vivi
4 – vive
5 – tive
6 – tine
7 – fine
8 – fins
9 – fias
10 – aias
11 – alas
12 – elas
13 – elos
14 – ecos
15 – ecoa

Memory leaks

Durante toda a implementação das funcionalidades tivemos extrema atenção ao espaço que era alocado durante a execução. Certificamo-nos que, sempre que já não fosse necessária determinada informação, libertávamos o espaço da memória onde essa informação estava alocada.

```
==8444== HEAP SUMMARY:
==8444==     in use at exit: 0 bytes in 0 blocks
==8444==   total heap usage: 25,565 allocs, 25,565 frees, 10,425,168 bytes alloc
ated
==8444==
==8444== All heap blocks were freed -- no leaks are possible
==8444==
==8444== For lists of detected and suppressed errors, rerun with: -s
==8444== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figura 21 - memory leaks

Alguns testes

Caminho mais curto de bem a mal (usando *big*)

0 – bem
1 – bom
2 – som
3 – sol
4 – sal
5 - mal

Tamanho do caminho: 6

Caminho mais curto de tudo a nada (usando *4 letters*)
0 – tudo
1 – todo
2 – nodo
3 – nado
4 - nada

Tamanho do caminho: 5

Lista de todos os nós pertencentes à componente de é-te

0 - é-te
1 - é-me
2 - é-se

Tamanho da componente: 3

Conclusões

No fim de muita pesquisa e tempo investido em desenvolvimento conseguimos obter um programa funcional, com todas as *features* pedidas e sem qualquer erro.

Este tema é de extrema importância dada a sua aplicabilidade prática no futuro e com este projeto conseguimos aprofundar o nosso conhecimento sobre a matéria abordada.

Os dados retirados comprovam o bom funcionamento dos métodos aplicados e por esse mesmo motivo estamos bastante satisfeitos com o trabalho aqui apresentado.

Código em MatLab

histogram_hashtable.m

```
clear;
load Dados\hash_table_distribution_big.txt

index = hash_table_distribution_big(:,1);
lenght = hash_table_distribution_big(:,2);

figure(1)
hist(lenght);
xlabel("Comprimento das linked lists");
ylabel("Número de linked lists");
title("big Tamanho das linked lists");

figure(2)
bar(index,lenght);
xlabel("Index da hash table");
ylabel("Comprimento da linked list");
title("big Comprimento da linked list em função do indice da hash
table")
```

load_factor.m

```
clear;
load Dados\hash_table_loadfactor_4letters.txt

n_entries = hash_table_loadfactor_4letters(:,1);
load_fac = hash_table_loadfactor_4letters(:,2);

figure("Name","4 Letters");
plot(n_entries,load_fac);
xlabel("Número de entries");
ylabel("Load factor");
title("Loadfactor 4-letters");

load Dados\hash_table_loadfactor_5letters.txt

n_entries = hash_table_loadfactor_5letters(:,1);
load_fac = hash_table_loadfactor_5letters(:,2);

figure("Name","5 Letters");
plot(n_entries,load_fac);
xlabel("Número de entries");
ylabel("Load factor");
title("Loadfactor 5-letters");

load Dados\hash_table_loadfactor_6letters.txt

n_entries = hash_table_loadfactor_6letters(:,1);
load_fac = hash_table_loadfactor_6letters(:,2);

figure("Name","6 Letters");
plot(n_entries,load_fac);
```

```
xlabel("Número de entries");
ylabel("Load factor");
title("Loadfactor 6-letters");

load Dados\hash_table_loadfactor_big.txt

n_entries = hash_table_loadfactor_big(:,1);
load_fac = hash_table_loadfactor_big(:,2);

figure("Name","Big");
plot(n_entries,load_fac);
xlabel("Número de entries");
ylabel("Load factor");
title("Loadfactor Big");
```

grafo.m

```
clear;
clc;

data = readtable("Dados\graph_6letters.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)
title("Graph 6-letter")
```

Código em C

```
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 = 50;
  hash_table->number_of_entries = 0;
  hash_table->number_of_edges = 0;
  hash_table->heads = (hash_table_node_t
**)malloc(sizeof(hash_table_node_t *) * hash_table->hash_table_size);

  if(hash_table->heads == NULL)
  {
    fprintf(stderr,"create_hash_table heads: out of memory\n");
    exit(1);
  }
  for(i = 0;i < hash_table->hash_table_size;i++)
    hash_table->heads[i] = NULL;

  return hash_table;
}


static void hash_table_grow(hash_table_t *hash_table)
{
  //
  // complete this
  //

  hash_table_node_t **old_heads, **new_heads;

  hash_table->hash_table_size = hash_table->hash_table_size * 2;

  new_heads = (hash_table_node_t **)malloc(sizeof(hash_table_node_t *)
* hash_table->hash_table_size);

  if(hash_table->heads == NULL)
  {
    fprintf(stderr,"create_hash_table heads: out of memory\n");
    exit(1);
  }

  for(unsigned int i = 0;i < hash_table->hash_table_size;i++)
    new_heads[i] = NULL;
```

```
  old_heads = hash_table->heads;

  hash_table->heads = new_heads;

  for(unsigned int i = 0;i < hash_table->hash_table_size/2;i++)
  {
    hash_table_node_t *node = old_heads[i];
    while(node != NULL)
    {
      hash_table_node_t *next = node->next;
      unsigned int j = crc32(node->word) % hash_table-
>hash_table_size;
      node->next = hash_table->heads[j];
      hash_table->heads[j] = node;
      node = next;
    }
  }
  free(old_heads);
}


static void hash_table_free(hash_table_t *hash_table)
{
  //
  // complete this
  //
  for (unsigned int i=0; i<hash_table->hash_table_size; i++)
  {
    hash_table_node_t *node = hash_table->heads[i];
    while(node != NULL)
    {
      hash_table_node_t *temp = node;
      node = node->next;
      adjacency_node_t *adj_node = temp->head;
      while (adj_node != NULL)
      {
        adjacency_node_t *temp_adj = adj_node;
        adj_node = adj_node->next;
        free_adjacency_node(temp_adj
      }
      free_hash_table_node(temp);
    }
  }
  free(hash_table->heads);
  free(hash_table);
}


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

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

```
  //
  // complete this
  //

  node = hash_table->heads[i];
  while(node != NULL)
  {
    if(strcmp(node->word,word) == 0){
      return node;
    }
    node = node->next;
  }

  if(insert_if_not_found == 1)
  {
    node = allocate_hash_table_node();
    node->previous = NULL;
    node->visited = 0;
    strcpy(node->word, word);
    node->head = NULL;
    node->next = hash_table->heads[i];
    hash_table->heads[i] = node;
    hash_table->number_of_entries++;
    node->representative = node;
    node->number_of_vertices = 1;
    node->number_of_edges = 0;


    FILE *fp = fopen("hash_table_loadfactor.txt", "a");

    if (fp == NULL)
    {
      printf("Error!");
      exit(1);
    }

    fprintf(fp, "%i,%.5f,%i\n", hash_table->number_of_entries,
(float)hash_table->number_of_entries/hash_table->hash_table_size, i);

    fclose(fp);

    if(hash_table->number_of_entries >= 0.75 * hash_table-
>hash_table_size)
    {
      hash_table_grow(hash_table);
    }

    return node;
  }
  else
  {
    return NULL;
  }

  return node;
}
```

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

  //
  // complete this
  //
  for(representative = node; representative->representative !=
representative; representative = representative->representative);

  for(next_node = node; next_node != representative; next_node = temp)
  {
    temp = next_node->representative;
    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 *edge_origin , *edge_to;

  to = find_word(hash_table,word,0);
  //
  // complete this
  //

  if(to == NULL || to == from)
  {
    return;
  }

  from_representative = find_representative(from);
  to_representative = find_representative(to);

  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 + 1;
      from_representative->number_of_edges = 0;
      from_representative->number_of_vertices = 0;
    }
    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 + 1;
      to_representative->number_of_edges = 0;
      to_representative->number_of_vertices = 0;
    }
  }else{
    from_representative->number_of_edges++;
  }

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

  edge_to = allocate_adjacency_node();
  edge_to->vertex = from;
  edge_to->next = to->head;
  to->head = edge_to;
}


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)
{
  //
  // complete this
  //
  hash_table_node_t *parent = NULL;

  if(origin == NULL)
  {
    fprintf(stderr,"breadh_first_search: origin is NULL\n");
    return -1;
  }

  int r = 0; int w = 1;
  list_of_vertices[0] = origin;
  origin->previous = NULL;
  origin->visited = 1;
  int stop = 0;

  while (r != w && stop == 0) {

    hash_table_node_t *current = list_of_vertices[r];

    parent = current;
    r++;

    for (adjacency_node_t *adj_node = current->head; adj_node != NULL;
adj_node = adj_node->next) {
      if(adj_node->vertex->visited == 0) {

        list_of_vertices[w++] = adj_node->vertex;
        adj_node->vertex->visited = 1;
        adj_node->vertex->previous = parent;
```

```
        if (adj_node->vertex == goal){
          stop = 1;
          break;
        }
      }
    }
  }

  for (int i = 0; i < w; i++) {
    list_of_vertices[i]->visited = 0;
  }

  return w;
}


static void list_connected_component(hash_table_t *hash_table,const
char *word)
{
  //
  // complete this
  //

  hash_table_node_t *node = find_word(hash_table,word,0);

  if (node == NULL){

    printf("list_connected_component: Word inserida não existe\n");
    return;
  }

  int vertex_max = find_representative(node)->number_of_vertices;
  hash_table_node_t **list_of_vertices = (hash_table_node_t **)
malloc(vertex_max * sizeof(hash_table_node_t*));

  int n_elements = breadh_first_search(vertex_max, list_of_vertices,
node, NULL);

  printf("-----------------------------------------\n");
  printf("Lista de todos os nós pertencentes à componente de %s\n",
word);
  for (int i = 0; i < n_elements; i++) {
    printf("%i - %s\n",i, list_of_vertices[i]->word);
  }
  printf("Tamanho da componente: %i\n", n_elements);
  printf("-----------------------------------------\n");

}


static int largest_diameter = -1, shortest_diameter = -1;
static hash_table_node_t **largest_diameter_example;

static int connected_component_diameter(hash_table_node_t *node)
{
  int diameter;
```

```
  //
  // complete this
  //
  hash_table_node_t **path, **list_of_vertices, **list_of_vertices2;

  int vertex_max = find_representative(node)->number_of_vertices;

  list_of_vertices = (hash_table_node_t **) malloc(vertex_max *
sizeof(hash_table_node_t*));
  list_of_vertices2 = (hash_table_node_t **) malloc(vertex_max *
sizeof(hash_table_node_t*));
  path = (hash_table_node_t **) malloc(vertex_max *
sizeof(hash_table_node_t *));

  if(list_of_vertices == NULL || list_of_vertices2 == NULL)
  {
    fprintf(stderr,"connected_component_diameter: out of memory\n");
    exit(1);
  }

  breadh_first_search(vertex_max, list_of_vertices, node, NULL);

  diameter = 0;
  int count = 0;
  for (int i = 0; i < vertex_max; i++) {

    int x = breadh_first_search(vertex_max, list_of_vertices2,
list_of_vertices[i], NULL);

    hash_table_node_t *temp = list_of_vertices2[x-1];

    count = 0;

    while(temp != NULL){
      temp = temp->previous;
      count++;
    }

    if(count > diameter){
      diameter = count;
      temp = list_of_vertices2[x-1];

      while(temp != NULL){
        path[--count] = temp;
        temp = temp->previous;
      }

    }
  }

  if(largest_diameter == -1 || diameter > largest_diameter) {
    if(largest_diameter_example != NULL){
      free(largest_diameter_example);
    }

    largest_diameter_example = (hash_table_node_t **) malloc(count *
sizeof(hash_table_node_t *));
```

```
    for(int i = 0; i < count; i++){
      largest_diameter_example[i] = path[i];
    }
  }
  if (diameter == -1)
  {
    printf("connected_component_diameter: diameter not found\n");
    return -1;
  }

  free(list_of_vertices);
  free(list_of_vertices2);
  free(path);
  return diameter;
}


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

  hash_table_node_t *from_node = find_word(hash_table,from_word,0);
  hash_table_node_t *to_node = find_word(hash_table,to_word,0);

  if (from_node == NULL || to_node == NULL){

    printf("path_finder: Word inserida não existe\n");
    return;
  }

  int vertex_max = find_representative(from_node)->number_of_vertices;

  hash_table_node_t **list_of_vertices = (hash_table_node_t **)
malloc(vertex_max * sizeof(hash_table_node_t*));;

  int n_elements = breadh_first_search(vertex_max, list_of_vertices,
from_node, to_node);

  hash_table_node_t *node = list_of_vertices[n_elements-1];
  int count = 0;

  while(node != NULL){
    node = node->previous;
    count++;
  }

  hash_table_node_t *solArr[count];
  node = to_node;
  solArr[count] = node;

  while(node != NULL){
    solArr[--count] = node;
    node = node->previous;
```

```
  }

  printf("-----------------------------------------\n");
  printf("Caminho mais curto de %s a %s\n", from_word, to_word);

  for (long unsigned int i = 0; i < sizeof(solArr)/sizeof(solArr[0]);
i++) {
    printf("%li - %s\n",i, solArr[i]->word);
  }

  int temp = sizeof(solArr)/sizeof(solArr[0]);
  printf("Tamanho do caminho: %i\n", temp);
  printf("-----------------------------------------\n");

  if(temp > largest_diameter){
    largest_diameter = temp;
    largest_diameter_example = solArr;
  }
  free(list_of_vertices);
}


static void graph_info(hash_table_t *hash_table)
{
  //
  // complete this
  //

  hash_table_node_t **representatives = (hash_table_node_t **)
malloc(hash_table->number_of_entries * sizeof(hash_table_node_t*));
  int size_smallest_component=-1, size_largest_component=-1,
sum_size_component = 0;


  if(representatives == NULL){
    printf("graph_info: Erro ao alocar memoria\n");
    return;
  }

  int index = 0, number_of_nodes = 0, diameter_sum = 0;

  for (unsigned int i=0; i<hash_table->hash_table_size; i++)
  {

    hash_table_node_t *node = hash_table->heads[i];

    while(node != NULL)
    {
      hash_table_node_t *temp = find_representative(node);

      int flag = 0;

      for (int j = 0; j < index; j++) {
        if (temp == representatives[j]) {
          flag = 1;
          break;
        }
```

```
      }

      if (flag == 0){
        representatives[index++] = temp;
        number_of_nodes += temp->number_of_vertices;
        hash_table->number_of_edges += temp->number_of_edges;

      }

      node = node->next;
    }

  }
  FILE *fp2 = fopen("graph_component_distribution.txt", "w");

  if (fp2 == NULL)
  {
    printf("Error!");
    exit(1);
  }

  for (int i = 0; i < index; i++) {
    hash_table_node_t *temp = representatives[i];
    int diameter = connected_component_diameter(temp) - 1;
    int comp_size = temp->number_of_vertices;

    fprintf(fp2, "%d,%d\n", comp_size, diameter);


    if(largest_diameter == -1 || diameter > largest_diameter) {
      largest_diameter = diameter;
    }

    if(shortest_diameter == -1 || diameter < shortest_diameter) {
      shortest_diameter = diameter;
    }
    diameter_sum += diameter;

    if(size_largest_component == -1 || comp_size >
size_largest_component) {
      size_largest_component = comp_size;
    }
    if(size_smallest_component == -1 || comp_size <
size_smallest_component) {
      size_smallest_component = comp_size;
    }
    sum_size_component += comp_size;
  }

  fclose(fp2);

  printf("-------------Graph data-------------\n");
  printf("Number of nodes: %i\n", number_of_nodes);
  printf("Number of edges: %i\n", hash_table->number_of_edges);
  printf("Numero de componentes conexas: %i\n", index);

  printf("---------------Component data--------------------\n");
```

```
  printf("Tamanho medio das componentes conexas: %.4f\n",
(float)sum_size_component/(float)index);
  printf("Tamanho maximo das componentes conexas: %i\n",
size_largest_component);
  printf("Tamanho minimo das componentes conexas: %i\n",
size_smallest_component);

  printf("--------------Diameter data------------------\n");
  printf("Diametro medio: %.4f\n", (float)diameter_sum/(float)index);
  printf("Diametro maximo: %i\n", largest_diameter);
  printf("Diametro minimo: %i\n", shortest_diameter);

  printf("-----------Largest Path--------------\n");
  for( int i = 0; i <= largest_diameter; i++ ){
    printf("%i - %s\n", i, largest_diameter_example[i]->word);
  }

  // FAZER GRAFO
  FILE *fp3 = fopen("graph.txt", "w");

  if (fp3 == NULL)
  {
    printf("Error!");
    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)
    {
      hash_table_node_t *temp = node;
      node = node->next;
      adjacency_node_t *adj_node = temp->head;
      while (adj_node != NULL)
      {
        adjacency_node_t *temp_adj = adj_node;
        adj_node = adj_node->next;
        fprintf(fp3, "%s %s\n", temp->word, temp_adj->vertex->word);
      }
    }
  }
  fclose(fp3);

  free(representatives);
  free(largest_diameter_example);
}


static void hash_table_info(hash_table_t *hash_table)
{
  printf("Hash table size : %i\n", hash_table->hash_table_size);
  printf("Hash table number of entries: %i\n", hash_table-
>number_of_entries);
  printf("Hash table load factor %.3f\n", (float)hash_table-
>number_of_entries/(float)hash_table->hash_table_size);

  int max = 0, min = -1, sum = 0, number_of_lists = 0;
```

```
  FILE *fp = fopen("hash_table_distribution.txt", "w");

  if (fp == NULL)
  {
    printf("Error!");
    exit(1);
  }

  for(unsigned int i = 0; i < hash_table->hash_table_size; i++){
    hash_table_node_t *node = hash_table->heads[i];
    int count = 0;

    if(node == NULL){
      continue;
    }

    number_of_lists++;

    while(node != NULL){
      count++;
      node = node->next;
    }
    if(count > max){
      max = count;
    }
    if(min == -1 || count < min){
      min = count;
    }
    sum += count;
    fprintf(fp, "%d,%d\n", i, count);

  }
  fclose(fp);

  printf("Hash table longest chain size: %i\n", max);
  printf("Hash table shortest chain size: %i\n", min);
  printf("Hash table average chain size: %.3f\n",
(float)sum/(float)number_of_lists);

}
```