

Algoritmos e Estruturas de Dados

Word Ladder

Professor Tomás Silva

10 de Janeiro de 2023

José Mendes – 107188 – 50%

Bernardo Pinto – 105926 – 50%



universidade
de aveiro

Índice

Introdução	3
Conhecendo o problema:	4
Implementação da Solução:	5
Hash Table:.....	5
Hash_table_create	5
Hash_table_grow.....	6
Hash_table_free.....	7
Find_word.....	8
Construção do grafo.....	10
Find_representative	10
Add_edge	11
Componente conexo:.....	12
Breadh_first_search.....	12
List_connected_component	15
Connected_component_diameter	16
Caminho mais curto entre duas palavras:	17
Path_finder.....	17
Representação de informação sobre o grafo	19
Graph_info	19
Resultados obtidos:	21
Wordlist-four-letters	21
Wordlist-five-letters	24
Wordlist-big-latest/ficheiro base	26
Memory leaks	28
Algumas Word ladders:	28
Conclusão	30
Apêndice do código	31
Código C:	31
Código Matlab:.....	55

Introdução

Este segundo trabalho prático da unidade curricular de Algoritmos e Estruturas de Dados tem como objetivo a implementação de uma **word ladder**, isto é, obter uma sequência de palavras que apenas diferem numa única letra entre palavras adjacentes. Esta sequência tem como início e fim duas palavras escolhidas pelo utilizador.

Por sua vez, a implementação deste problema utiliza a estrutura de dados *hash table*, com o intuito de armazenar todas as palavras que serão mais tarde utilizadas para elaborar as sequências de palavras.

Conhecendo o problema:

Com o intuito de entender o problema e a solução apresentada é necessário, primeiramente, entender o que é e como funciona uma “*hash table*”.

Hash Table, ou tabela de dispersão, é uma estrutura de dados que relaciona chaves de pesquisa a valores. Pelo que, ao passar estas chaves a uma função de dispersão, é gerado um índice correspondente no *array* onde a informação (i.e., o valor) será guardada. Então, por meio da *hashtable* e da função de dispersão, a pesquisa de informação torna-se muito eficaz, $O(1)$ (Big Oh). No entanto, vale ressaltar que a função de dispersão pode gerar “colisões”, ou seja, elementos distintos com um mesmo índice, sendo, portanto, importante a escolha de um bom tamanho para a *hash table*.

O problema proposto pede que seja desenvolvido uma “word ladder”, que, resumidamente, será uma sequência de palavras em que duas palavras adjacente diferem apenas numa letra. Além disso, foram traçados objetivos e metas a serem seguidas para que a solução final fosse encontrada de forma mais eficiente, entre elas:

- Construção correta da *hash table*.
- Construção de um grafo.
- Possibilitar uma busca e listagem completa do grafo.
- Possibilitar a pesquisa e listagem de um componente conexo.
- Possibilitar a busca do caminho mais curto entre duas palavras
- Possibilitar a descoberta do diâmetro de um componente conexo

Entre outras.

Então, entrando no problema abordado por este relatório, as palavras a serem armazenadas (i.e., array de caracteres) serão as chaves, que após passarem pela função de dispersão, irão receber o seu índice correspondente na tabela. Já as palavras que possuem o mesmo índice (colisão), não resultam em nenhuma complicação, uma vez que caso isto aconteça é criada uma lista ligada com essas palavras.

Implementação da Solução:

Hash Table:

Hash_table_create

A função responsável por “criar” a hash table começa por instanciar o objeto **hash_table* do tipo ponteiro para *hash_table_t* e por instanciar uma variável do tipo inteiro sem sinal, que será utilizada em ciclos mais a frente nesta função.

Em seguida, é realizada a alocação da memória para a *hash_table* e é implementado um bloco condicional que irá testar se ocorreu um erro durante esta criação. Em caso positivo, irá apresentar uma mensagem de erro seguida da interrupção do programa.



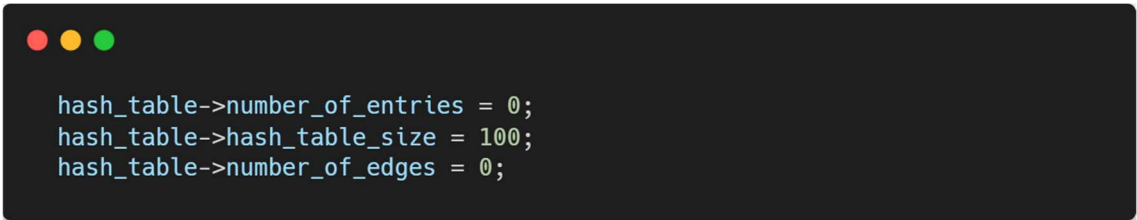
```
hash_table_t *hash_table;
unsigned int i;

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);
}
```

Figura 1 - Alocação da memória para a hash table

No primeiro campo designado para inserção de código foram instanciadas algumas características iniciais da estrutura hash table, por exemplo:

- Número de entradas igual a zero
- Tamanho da tabela igual a cem
- Número de arestas igual a zero



```
hash_table->number_of_entries = 0;
hash_table->hash_table_size = 100;
hash_table->number_of_edges = 0;
```

Figura 2 - Características iniciais da estrutura

Posteriormente, foi implementado o código responsável por realizar a alocação dinâmica da memória para cada posição da *hash table* (i.e. *heads*) e, mais uma vez, seguido do bloco condicional que testa se ocorreu um erro nesta alocação.

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

Figura 3 - Alocação de memória para os heads

Por fim, foi implementado um ciclo *for* com objetivo de percorrer a *hash table* e atribuir a cada posição, *heads*, o valor *NULL*, garantindo que poderão ser preenchidos posteriormente. Então, a função retorna o objeto *hash_table*.

```
for(i = 0; i < hash_table->hash_table_size; i++)
{
    hash_table->heads[i] = NULL;
}

return hash_table;
```

Figura 4 - Atribuição do valor NULL para todos os campos da hash table

Hash_table_grow

A função *hash_table_grow*, que é responsável por fazer o incremento no tamanho da *hash table*. Esta é passada como parâmetro.

Inicialmente são instanciadas as variáveis *old_size*, *i* e *j* do tipo inteiro sem sinal e por declarar os objetos *old_heads* do tipo ponteiro para ponteiro de *hash_table_node_t*, além de *n* e *nn* do tipo ponteiro para *hash_table_node_t*.

Seguidamente, a variável *old_size* e *old_heads* recebem, respetivamente, o tamanho e os nós do objeto passado como parâmetro, para que, assim, fiquem salvos, e o tamanho da *hash table* é incrementado em 50% mais 1 do tamanho original.

```
unsigned int old_size, i, j;
hash_table_node_t **old_heads, *n, *nn;

old_size = hash_table->hash_table_size;
old_heads = hash_table->heads;
hash_table->hash_table_size *= 1.5 + 1;
```

Figura 5 - Instanciamento de variáveis

Na continuação, é realizada a alocação dinâmica da memória através da função *malloc*, para o novo tamanho da *hash table*. Também foi implementado um ciclo *for* para percorrer esta nova *hash table* com tamanho atualizado e passar todos os nós para o valor *NULL*, para que possam receber outro valor posteriormente.

```
hash_table->heads = (hash_table_node_t **)malloc(sizeof(hash_table_node_t) * hash_table->hash_table_size);  
for(i = 0; i < hash_table->hash_table_size; i++)  
    hash_table->heads[i] = NULL;
```

Figura 6 - Alocação de memória e atribuição do valor *NULL* aos nós

Finalmente, foram implementados dois ciclos *for* que serão responsáveis por percorrer o array *old_heads*, que armazena os nós da antiga hash table, e, através da função de dispersão, armazená-los na nova tabela, assim como a sua respectiva lista ligada, caso exista. Em último lugar, libertamos o espaço alocado pela tabela antiga através do comando *free(old_heads)*.

```
for(i = 0; i < old_size; i++)  
    for(n = old_heads[i]; n != NULL; n = nn)  
    {  
        nn = n->next;  
        j = crc32(n->word) % hash_table->hash_table_size;  
        n->next = hash_table->heads[j];  
        hash_table->heads[j] = n;  
        old_heads[i] = NULL;  
    }  
free(old_heads);
```

Figura 7 - Armazenamento na nova hash table e liberação de memória

Hash_table_free

Já a função responsável por libertar a memória alocada para a *hash table* inicia por declarar uma variável *i* do tipo inteiro sem sinal, instanciar um objeto *temp* do tipo *hash_table_node_t** e um outro objeto *adj_temp* do tipo *adjacency_node_t**.

```
unsigned int i;
hash_table_node_t* temp;
adjacency_node_t* adj_temp;
```

Figura 8 - Instanciamento de variáveis

Em seguida, foi desenvolvido um ciclo *for*, que será responsável por percorrer todos os nós da *hash table*, e, para cada nó, a função percorre e liberta a memória alocada para a sua lista ligada. Posteriormente, liberta a memória alocada para o nó em si. Garantindo, portanto, que toda a memória seja libertada, ao invés de apenas a memória alocada para um índice da tabela.

Por fim, a função liberta a memória alocada para os *heads* e para a *hash table* em si, através do comando *free*.

```
for(i = 0; i < hash_table->hash_table_size; i++)
{
    temp = hash_table->heads[i];
    while(temp != NULL)
    {
        adj_temp = temp->head;
        while(adj_temp != NULL)
        {
            temp->head = temp->head->next;
            free_adjacency_node(adj_temp);
            adj_temp = temp->head;
        }
        hash_table->heads[i] = hash_table->heads[i]->next;
        free_hash_table_node(temp);
        temp = hash_table->heads[i];
    }
}
free(hash_table->heads);
free(hash_table);
```

Figura 9 - Ciclo responsável por liberar a memória do nó e sua lista ligada

Find_word

A função *find_word*, é responsável por encontrar e/ou inserir uma palavra na tabela. Recebe como parâmetros **hash_table*, **word* e a variável inteira *insert_if_not_found*, que irá determinar se, caso a palavra não seja encontrada, ela deve ou não ser inserida na *hash table*.

Inicialmente, a variável **word* é passada como argumento da função de dispersão que devolve um índice e, em seguida, para os casos em que o nó correspondente ao índice seja nulo (i.e, a palavra não está na tabela), surgem duas opções. Caso o valor da variável *insert_if_not_found*, seja igual a zero (i.e, a palavra não deve ser inserida), é apenas realizado o return do objeto *node* ainda com valor nulo. Em contrapartida, caso o valor da variável responsável por decidir se a palavra deve ser inserida seja diferente de zero, são realizados os seguintes procedimentos para inseri-la:

- É feito, através da função *allocate_hash_table_node*, a alocação da memória para o nó;
- *node->word* guarda a palavra;
- *node->next* passa a apontar para o nó no índice da tabela;
- A posição da tabela, *hash_table->heads[i]* passa a ser o novo nó;
- Como ainda não foi criado o grafo o seu representativo é ele próprio;
- *node->visited* recebe o valor 0.
- O caminho *node->previous* recebe o valor *NULL*, uma vez que este ponteiro apenas será útil mais para a frente;
- O número de vértices inicial é 1.
- O número de arestas inicial é 0.
- O número de entradas da *hashtable* é incrementado em uma unidade.

E, por fim, é feito o *return* da função com o valor do nó que foi, de certa forma, criado

No entanto, para o caso de o nó correspondente ao índice da função de dispersão ser diferente de *NULL* (i.e, aquela posição está ocupada), foi implementado um ciclo *for* que será responsável por, através dos ponteiros *node->next* de cada nó, percorrer todas as palavras presentes naquele índice (lista ligada) e, com a função *strcmp(node->word, word)*, encontrar, ou não, a palavra desejada. Então, é realizado o *return* do nó.

Vale ressaltar que, antes do return da função e independente das possibilidades descritas acima, é verificado, através de um bloco condicional, se a tabela já está com 50% da sua ocupação, e, em caso positivo, é chamada a função *hash_table_grow* para realizar o aumentar o tamanho da nossa *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;
    for(node = hash_table->heads[i]; node != NULL && strcmp(node->word, word) !=
0; node = node->next)
        ;
    if(node == NULL && insert_if_not_found != 0)
    {
        node = allocate_hash_table_node();
        strcpy(node->word, word);
        node->next = hash_table->heads[i];
        hash_table->heads[i] = node;
        node->representative = node;
        node->visited = 0;
        node->previous = NULL;
        node->number_of_vertices = 1;
        node->number_of_edges = 0;
        hash_table->number_of_entries = hash_table->number_of_entries + 1;
    }

    if(hash_table->number_of_entries*2 > hash_table->hash_table_size){
        hash_table_grow(hash_table);
    }

    return node;
}

```

Figura 10 - Função find_word

Construção do grafo

Find_representative

Esta função tem como objetivo encontrar o nó representativo do nó introduzido como parâmetro da função, **node**, para poder auxiliar na construção do grafo. Cada um dos componentes conexos que constituem o grafo possui um nó representativo.

Inicialmente, são declarados três *hash_table_nodes*, **n1**, **n2**, e **n3**, em que o primeiro é usado para obter o nó representativo através de um *for*. Este *loop* começa por igualar **n1** ao nó passado como argumento. Em cada incremento vamos verificar se esse nó corresponde ao seu próprio representativo. Caso isto não se verifique, igualamos **n1** ao seu nó representativo até que estes sejam iguais, uma vez que, quando isto acontecer, **n1** corresponde ao nó representativo que procurávamos.

De seguida, o segundo *loop for* é bastante similar ao primeiro, no entanto, este tem como objetivo colocar em **node->representative** (sendo node, um nó pertencente ao grafo) o valor do nó representativo encontrado.

Por fim, tal como pretendido, a função retorna **n1**, que, como visto antes, representa o nó representativo.

```
static hash_table_node_t *find_representative(hash_table_node_t *node)
{
    hash_table_node_t *n1,*n2, *n3;

    //
    // complete this
    //
    for(n1 = node; n1->representative != n1; n1 = n1->representative)
        ;
    for(n2 = node; n2 != n1; n2 = n3)
    {
        n3 = n2->representative;
        n2->representative = n1;
    }
    return n1;
}
```

Figura 11 - Função *find_representative*

Add_edge

A função **add_edge** recebe como argumentos a *hash table*, o nó, **from**, e uma *char* constante, **word**, e tem o intuito de contruir as ligações entre nós de palavras semelhantes que constituem a *hash table*, de maneira a possibilitar a construção do grafo. A *char* constante, **word**, representa, portanto, uma palavra semelhante aquela que o nó, **from**, possui, pelo que a ligação a ser estabelecida será entre estes.

Primeiramente, é utilizada a função **find_word** para verificar a existência do nó que possui **word**. Caso exista, o número de *edges* da *hash table* é incrementado e a ligação entre o nó inicial, **from**, e o nó agora definido, **to**, é formada. Caso contrário, ou seja, caso a função **find_word** retorne *NULL*, esta função termina.

Por sua vez, para estabelecer a ligação, são usados os nós adjacentes, **link** e **link2**. O primeiro vai realizar a ligação de **from** para **to** e o segundo o contrário, através do **vertex**. Em último lugar é necessário associar este novo nó à *linked list* de adjacência.

Finalmente, uma vez que estamos a criar um componente conexo, é necessário atribuir a estas, até antes, componentes separadas, um mesmo representativo. Se já for o caso, significa que estes já pertencem ao mesmo componente conexo, pelo que, apenas era preciso estabelecer a ligação entre os nós, pelo que a função termina aqui. Em oposição, verifica-se qual dos componentes separados possui o maior tamanho. Este possui o nó representativo que será utilizado, pois, desta forma, serão necessárias menos alterações. São, portanto, atribuídos os **edges** e os **vertices** do menor ao maior, e por fim, o representativo do, então, menor componente passa a ser igual ao do maior.

```

static void add_edge(hash_table_t *hash_table, hash_table_node_t *from, const char *word)
{
    hash_table_node_t *to, *from_representative, *to_representative;
    adjacency_node_t *link, *link2;

    //
    // complete this
    //
    to = find_word(hash_table, word, 0);
    if(to == NULL) // word nao existe
        return;

    hash_table->number_of_edges++;

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

    // temos de criar as ligações entre o from e o to
    // from
    link = allocate_adjacency_node(); // reserva a memoria para poder ser usado
    link->vertex = to;
    link->next = from->head;
    from->head = link;

    // to
    link2 = allocate_adjacency_node(); // reserva a memoria para poder ser usado
    link2->vertex = from;
    link2->next = to->head;
    to->head = link2;

    if(from_representative == to_representative) // têm o mesmo representativo logo so precisava da ligação
        return;

    if(from_representative->number_of_vertices < to_representative->number_of_vertices)
    {
        to_representative->number_of_edges = to_representative->number_of_edges + 1 + from_representative->number_of_edges;
        to_representative->number_of_vertices += from_representative->number_of_vertices;
        from_representative->representative = to_representative;
    }
    else{
        from_representative->number_of_edges = from_representative->number_of_edges + 1 + to_representative->number_of_edges;
        from_representative->number_of_vertices += to_representative->number_of_vertices;
        to_representative->representative = from_representative;
    }
}

```

Figura 12 - Função add_edge

Componente conexo:

Breadth_first_search

Esta função tem como objetivo, a partir um nó inicial, **origin**, tentar encontrar um nó final, **goal**, retornando o número de vértices “visitados” entre estes nós. Recebe como parâmetros o número máximo de vértices que uma componente conexas pode ter, **maximum_number_of_vertices**, um *array* que guarda os vértices a serem utilizados, **list_of_vertices**, e os nós previamente referidos, **origin** e **goal**.

Para realizar esta função de procura, uma vez que em oposição a uma procura *depth_first*, a função começa por procurar o nó pretendido em termos de largura. Com isto em mente, o grupo decidiu utilizar dois números inteiros, em que um possui o índice do nó que é lido para escrever os seus nós adjacentes, **read** (com valor inicial 0), e o outro é um índice do nó a ser escrito, **write** (com valor inicial 1). O primeiro nó a ser lido será o **origin**, uma vez que é suposto encontrar um caminho deste até ao **goal**.

À medida que estes nós são introduzidos no *array list_of_vertices*, são colocados como “visitados”, para indicar que já foram escritos, e o **previous** de cada nó passa a apontar para o nó do qual está a ser lido.

Assim que o nó que está a ser lido já não possui mais vértices adjacentes para adicionar ao *array*, o índice de **read** incrementa, começando a ler o próximo e a escrever os adjacentes deste.

Caso a palavra que está a ser escrita no array seja o nosso nó **goal**, é acionada a *flag found_flag* (colocada a 1), para que seja possível obter o caminho mais curto a partir do nó **origin**. Para tal, basta seguir os **previous** que foram colocados ao longo de cada iteração, obtendo assim também o número de vértices “visitados”, que era o nosso objetivo.

Em último lugar, é necessário esvaziar o *array* que contém os vértices para poder ser novamente usado, bem como colocar **visited** de cada nó a 0 para o mesmo efeito. A função retorna o número inteiro pretendido caso o nó seja encontrado, caso contrário retorna -1.

```

static int breath_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
    //

    // supondo que os vertices ja estao no list_of_vertices:
    int write, read, found_flag, count, clear;
    adjacency_node_t *node;
    hash_table_node_t *go_back_node;

    write = 1; read = 0, found_flag = 0;
    origin->visited = 1;
    origin->previous = NULL; // garantir que esta NULL para fazer o search mais tarde
    list_of_vertices[0] = origin; // o primeiro é o from e vamos procurar o to
    node = list_of_vertices[0]->head; // a primeira palavra adjacente á origin

    while(read < maximum_number_of_vertices) // para ler todas as palavras na lista se necessário
    {
        while(node != NULL)
        {
            if(node->vertex->visited == 1) // ja esteve neste node, nao vai ca estar again
            {
                node = node->next;
                continue;
            }

            node->vertex->visited = 1; // foi visitado;
            node->vertex->previous = list_of_vertices[read]; // definir o node anterior para mais tarde conseguir descobrir o
caminho
            list_of_vertices[write] = node->vertex;
            if(node->vertex == goal) // encontramos a solução
            {
                found_flag = 1;
                break;
            }
            write++;
            node = node->next; // avança para o proximo adjacnete
        }
        // vai sempre encontrar algures uma vez que a palavra esta no mesmo componente conexo
        if(found_flag == 1) // encontrou o goal --> é o node->vertex
        {
            // limpar os visited
            for(clear = 0; clear <= write; clear++)
                list_of_vertices[clear]->visited = 0;

            // determinar o pretendido
            count = 1; // começa com um que é no que está
            go_back_node = node->vertex; // node que vai sendo atualizado com o previous para encontrar o caminho de volta
            while(go_back_node->previous != NULL){
                count++;
                go_back_node = go_back_node->previous;
            }
            return count;
        }
        read++; // avança para o proximo elemento a ler --> seguinfo as regras deste
        if(list_of_vertices[read] == NULL)
            break;

        node = list_of_vertices[read]->head;
    }
    for(clear = 0; clear < write; clear++)
        list_of_vertices[clear]->visited = 0;
    return -1;
}

```

Figura 13 - Função Breadh_first_search

List_connected_component

Esta função tem o intuito de listar todos os vértices que pertencem a um determinado componente conexo. Os parâmetros da função são a *hash table*, e uma *char** constante, **word** que será a palavra inicial para esta listagem. Esta função é utilizada na opção 1 quando executamos o programa.

A ideia base por de trás desta função era utilizar a função já apresentada anteriormente, **breath_frist_search**, em que o **goal** seria um nó com valor *NULL*. Desta forma, o *array* de vértices, **list_of_vertices**, será totalmente preenchido, possuindo assim todos os vértices daquela componente conexa, que é o que pretendemos listar.

Inicialmente, é então feita uma alocação de memória usando o comando **malloc** para o *array*, **list_of_vertices**, que juntamente com a variável, **maximum_number_of_vertices**, serão utilizados como parâmetros da função **breath_frist_search**. Esta variável recebe o valor do tamanho da *hash table*, **hash_table_size**.

De seguida, é utilizada a função **find_word**, para verificar se a palavra introduzida corresponde a um nó e, portanto, existe. O **goal** é, por sua vez, colocado a *NULL*, para que possa preencher o *array* até o final.

É então realizado um **breath_first_search** com os parâmetros já referidos, de forma a obter a lista com todos os vértices do componente conexo.

Finalmente, apenas é necessário imprimir os vértices do *array*. É feito um *free* do **list_of_vertices**.

```
static void list_connected_component(hash_table_t *hash_table, const char *word) //nao sei se ja esta --> nem esta nem a
de cima
{
    //
    // complete this
    //
    // usar o array dado pelo bfs inteiro
    hash_table_node_t **list_of_vertices;
    int maximum_number_of_vertices;
    unsigned int i;
    hash_table_node_t *origin, *goal, *node;

    list_of_vertices = (hash_table_node_t **)malloc(sizeof(hash_table_node_t) * hash_table->hash_table_size);
    maximum_number_of_vertices = hash_table->hash_table_size;
    origin = find_word(hash_table, word, 0);
    if(origin == NULL)
        return;
    goal = NULL;

    breath_first_search(maximum_number_of_vertices, list_of_vertices, origin, goal);

    i = 0;
    node = list_of_vertices[i];
    while(node != NULL)
    {
        printf("%s\n", node->word);
        list_of_vertices[i] = NULL;
        i++;
        node = list_of_vertices[i];
    }
    free(list_of_vertices);
}
```

Figura 14 - Função *list_connected_component*

Connected_component_diameter

A função *connected_component_diameter* é responsável por calcular o diâmetro de um componente conexo, ou seja, a maior distância entre dois vértices, utilizando o caminho mais curto, entre duas palavras de um mesmo componente conexo.

A implementação inicia por realizar a alocação de memória para o objeto *list_of_vertices* e por atribuir à variável *maximum_number_of_vertices* o tamanho da *hashtable*.

Em seguida, é invocada a função *breath_first_search*, que recebe como argumento as variáveis *maximum_number_of_vertices*, *list_of_vertices*, *node* e *NULL*. Então, uma vez que o valor referente ao *goal* da função é *NULL* o array *list_of_vertices* vai armazenar todos os vértices do componente conexo em questão, podendo, portanto, ser usado a seguir.

Posteriormente, foi implementado um ciclo *while* que será responsável por percorrer todos os vértices do componente conexo e armazenar na variável *node* o valor do seu último vértice. Na continuação, foi desenvolvido um outro ciclo *while* para percorrer, a partir do caminho *node->previous*, todo o componente e incrementar em uma unidade o valor da variável inteira *diameter*, que vai guardar o valor final do diâmetro do componente conexo.

Por fim, após libertar a memória usada pelo array *list_of_vertices*, é realizado o *return* da variável *diameter*.

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

    hash_table_node_t **list_of_vertices;
    int maximum_number_of_vertices;

    list_of_vertices = (hash_table_node_t **)malloc(sizeof(hash_table_node_t) * size);
    maximum_number_of_vertices = size;

    breath_first_search(maximum_number_of_vertices, list_of_vertices, node, NULL);

    int i = 0;
    while(list_of_vertices[i] != NULL){
        node = list_of_vertices[i];
        list_of_vertices[i] = NULL;
        i++;
    }

    diameter = 0
    while(node->previous != NULL)
    {
        diameter++;
        node = node->previous;
    }

    free(list_of_vertices);
    return diameter;
}
```

Figura 15 - Função *connected_component_diameter*

Caminho mais curto entre duas palavras:

Path_finder

A função **path_finder** é utilizada quando a opção 2 que aparece quando o programa é executado, e tem como objetivo obter a *word ladder* de uma certa palavra a outra, ambas introduzidas pelo utilizador, caso seja possível.

Em primeiro lugar, uma vez que esta função utiliza a **breath_first_search**, é necessário alocar memória para um *array* de vértices, **list_of_vertices**, e atribuir o máximo de vértices que um componente conexo pode ter, **maximum_number_of_vertices**. Os outros dois restantes parâmetros necessários para a realização desta função de procura será os nós, inicial e final. Estes são dados pelas duas palavras introduzidas como parâmetro da função.

É necessário realizar o **find_word** para ambas as palavras, com o intuito de obter os dois nós. Isto só é possível caso as palavras já existam na *hash table*.

Seguidamente, verifica-se que as duas palavras fazem parte do mesmo componente conexo, uma vez que, caso isto não aconteça, não existe *word ladder* de uma palavra à outra.

Caso tudo se verifique, é então realizado o **breath_first_search**, com todos os parâmetros previamente determinados. O *array* de vértices é, desta forma, preenchido.

Posteriormente, uma vez que é possível traçar o caminho utilizando os **previous** desde o nó final até ao inicial, é necessário inverter esta ordem. A forma que o grupo decidiu tomar para esta implementação foi um *array* **previous_words** que á medida que vamos procurando as palavras com o **previous**, este guarda os nós.

No final basta imprimir este *array* pela ordem inversa e obtemos, desta maneira, a *word ladder*. É feito um *free* do **list_of_vertices** e do **previous_words**.

```

static void path_finder(hash_table_t *hash_table, const char *from_word, const char *to_word)
{
    //
    // complete this
    //
    hash_table_node_t **list_of_vertices, *from, *to, *from_representative, *to_representative;
    int maximum_number_of_vertices, count, word_number;

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

    from = find_word(hash_table, from_word, 0);
    to = find_word(hash_table, to_word, 0);
    if(from == NULL || to == NULL)
    {
        printf("Palavra inexistente\n");
        return;
    }
    from_representative = find_representative(from);
    to_representative = find_representative(to);
    // printf("%s == %s\n", from_representative->word, to_representative->word);
    if(from_representative != to_representative)
    {
        printf("Palavras introduzidas não apresentam word ladder\n");
        return;
    }
    breadth_first_search(maximum_number_of_vertices, list_of_vertices, from, to);
    hash_table_node_t **previous_words = (hash_table_node_t **)malloc(sizeof(hash_table_node_t) * hash_table->hash_table_size);
    count = 0;

    while(to != NULL)
    {
        previous_words[count] = to;
        to = to->previous;
        count++;
    }

    word_number = 0;
    count--;
    while(count >= 0)
    {
        printf("[%d] %s\n", word_number, previous_words[count]->word);
        count--;
        word_number++;
    }
    free(list_of_vertices); free(previous_words);
}

```

Figura 16 - Função path_finder

Representação de informação sobre o grafo

Graph_info

Esta função recebe como parâmetro a própria *hash table* e tem como objetivo obter e apresentar algumas informações sobre o grafo. Alguns dados que o grupo achou interessantes foram: o número de componentes conexos, o número de vértices, o número de arestas, o maior diâmetro que este componente conexo possui, apresentando a sua *word ladder* e, por fim, o número de componentes conexos com o mesmo diâmetro.

Com o intuito de obter estas informações é necessário percorrer todas as palavras que constituem a *hashtable*.

Sempre que aparece um nó que possui um diferente representativo, o número de componentes conexos é incrementado. Para verificar que aquele componente já foi adicionado, a variável *visited*, utilizada anteriormente em outras funções, é colocada a 1. Através de cada representativo também é possível obter o número de vértices de cada componente conexo, pelo que, para obter o número de vértices total do grafo basta somar os vértices de todos os componentes conexos a uma variável previamente definida **vertices**. Os representativos são guardados num *array* **representatives**.

De forma a obter o diâmetro de cada componente conexo o grupo decidiu adicionar uma variável inteira à *struct* dos nós da *hashtable*, **biggest_diameter_of_cc**. Sempre que um diâmetro de um nó é maior do que o guardado nesta variável, este passa a ser o novo **biggest_diameter_of_cc**. No final, utilizando o *array* que possui todos os representativos, é possível calcular quantos componentes conexos têm o mesmo diâmetro apenas incrementando o índice correspondente ao diâmetro do *array* **all_diameters**. Finalmente, é só necessário percorrer o *array* e ignorar as posições preenchidas com 0.

O maior diâmetro é obtido calculando o diâmetro, através da função **connected_component_diameter**, de todos os nós do grafo, sendo, por isso, um processo demorado. Todos os nós são armazenados com o seu respetivo índice no *array* **largest_diameter_example**, de forma que possam ser de fácil acesso. Quando um novo maior diâmetro é encontrado, a sua posição neste último *array* é guardada.

Esta posição é útil para obter, utilizando a função **breadth_first_search**, o *array* total de vértices começando naquela palavra. Deste modo é obtida a última palavra da maior *word ladder* daquele componente conexo. Por fim basta apresentá-la utilizando a função **path_finder**.

Em último lugar, para obter as arestas totais do grafo, basta utilizar a variável, previamente preenchida na função **add_edge**, *hash_table->number_of_edges*. Finalmente são libertadas todas as alocações de memória utilizadas.

```

static void graph_info(hash_table_t *hash_table)
{
    //
    // complete this
    //
    hash_table_node_t **representatives = (hash_table_node_t **)malloc(sizeof(hash_table_node_t) * hash_table->hash_table_size);
    largest_diameter_example = (hash_table_node_t **)malloc(sizeof(hash_table_node_t) * hash_table->hash_table_size);
    hash_table_node_t *node, *node_representative, *goal;
    unsigned int i, h;
    int cc, vertices, j, diameter, store_ld_index, ind, all_diameter_counter;
    cc = 0, vertices = 0, largest_diameter = 0, store_ld_index = -1, ind = 0;
    goal = NULL;

    for(i = 0; i < hash_table->hash_table_size; i++){
        for(node = hash_table->heads[i]; node != NULL; node = node->next){
            node_representative = find_representative(node);

            for(h = 0; h < hash_table->hash_table_size; h++){ // para guardar o numero de cc e tudo mais
                if(node_representative == representatives[h])
                    break;

                if(h == hash_table->hash_table_size - 1){// caso ainda nao tenha visto o representante
                    representatives[cc] = node_representative;
                    cc++;
                    node_representative->visited = 1;
                    vertices += node_representative->number_of_vertices;
                }
            }

            diameter = connected_component_diameter(node); // encontra o diametro daquele cc
            largest_diameter_example[ind] = node;

            if(diameter > largest_diameter){
                largest_diameter = diameter;
                store_ld_index = ind;
            }

            if(diameter > node_representative->biggest_diameter_of_cc)
                node_representative->biggest_diameter_of_cc = diameter;

            ind++;
        }
    }

    int all_diameters[largest_diameter+1];
    for(int a = 0; a < largest_diameter+1; a++)
        all_diameters[a] = 0;

    // vertices, edges, cc's uma cena assim // maximo vertices e edces num determinado cc
    printf("Number of connected components: %d\n", cc);
    printf("Number of vertices: %d\n", vertices);
    printf("Number of edges: %d\n", hash_table->number_of_edges);
    printf("Largest Diameter: %d\n", largest_diameter);
    hash_table_node_t **list_of_vertices = list_of_vertices = (hash_table_node_t **)malloc(sizeof(hash_table_node_t) *
hash_table->hash_table_size);
    int maximum_number_of_vertices = hash_table->hash_table_size;
    breadth_first_search(maximum_number_of_vertices, list_of_vertices, largest_diameter_example[store_ld_index], NULL);
    int count = 0;
    while (list_of_vertices[count] != NULL)
    {
        goal = list_of_vertices[count]; // tem o intuito de guardar o ultimo
        list_of_vertices[count] = NULL; // vai limpando
        count++;
    }
    printf("Longest word ladder:\n");
    path_finder(hash_table, largest_diameter_example[store_ld_index]->word, goal->word);

    j = 0;
    while(representatives[j] != NULL){
        all_diameters[representatives[j]->biggest_diameter_of_cc] += 1;
        representatives[j]->visited = 0;
        largest_diameter_example[j] = NULL; // tem ambos o mesmo comprimento
        j++;
    }

    printf("Number of connected components with a diameter of:\n");
    for(all_diameter_counter = 0; all_diameter_counter < largest_diameter+1; all_diameter_counter++){
        if(all_diameters[all_diameter_counter] != 0)
            printf("%d: %d\n", all_diameter_counter, all_diameters[all_diameter_counter]);
    }

    free(representatives); free(largest_diameter_example); free_hash_table_node(goal); free(list_of_vertices);
}

```

Figura 17 - Função graph_info

Resultados obtidos:

Na obtenção de resultados, o grupo concluiu que seria mais prático utilizar como argumento na execução do programa o ficheiro previamente fornecido, “wordlist-four-letters.txt”, uma vez que dos três é aquele que possui menor número de palavras. Isto significa que a execução do programa seria mais rápida, uma vez que utilizando o ficheiro base este demora a terminar cerca de 20 minutos, o que tornaria impossível testar repetidamente os métodos implementados.

Wordlist-four-letters

```
mendes@mendesze:~/Desktop/AED/Projeto2/A02$ ./word_ladder wordlist-four-letters.txt
Number of connected components: 187
Number of vertices: 2149
Number of edges: 9267
Largest Diameter: 15
Longest word ladder:
[0] Hong
[1] Kong
[2] King
[3] Ping
[4] Pina
[5] tina
[6] tino
[7] tipo
[8] aipo
[9] arpo
[10] arpa
[11] aspa
[12] assa
[13] essa
[14] esta
[15] está
Number of connected components with a diameter of:
0: 164
1: 19
2: 2
3: 1
15: 1
Your wish is my command:
 1 WORD      (list the connected component WORD belongs to)
 2 FROM TO   (list the shortest path from FROM to TO)
 3           (terminate)
> 
```

Figura 18 - Saída inicial para wordlist-four-letters.txt

```
Your wish is my command:
  1 WORD      (list the connected component WORD belongs to)
  2 FROM TO   (list the shortest path from FROM to TO)
  3           (terminate)
> 1 pena
pena
pela
peca
peja
pega
peno
pene
peça
peva
peta
pesa
pera
puna
sena
rena
lena
cena
Sena
pana
pelo
pele
pula
pila
zela
vela
tela
sela
rela
nela
mela
gela
dela
cela
bela
pala
peco
pica
teca
seca
Zeca
Meca
pejo
peje
puja
poja
veja
seja
```

Figura 19 - Componente conexo da palavra "pena" em wordlist-four-letters.txt

```

Your wish is my command:
  1 WORD      (list the connected component WORD belongs to)
  2 FROM TO   (list the shortest path from FROM to TO)
  3           (terminate)
> 2 pena lixo
[0] pena
[1] pega
[2] pego
[3] lego
[4] ligo
[5] lixo
Your wish is my command:
  1 WORD      (list the connected component WORD belongs to)
  2 FROM TO   (list the shortest path from FROM to TO)
  3           (terminate)
> 

```

Figura 20 - Menor caminho entre "pena" e "lixo" em wordlist-four-letters.txt

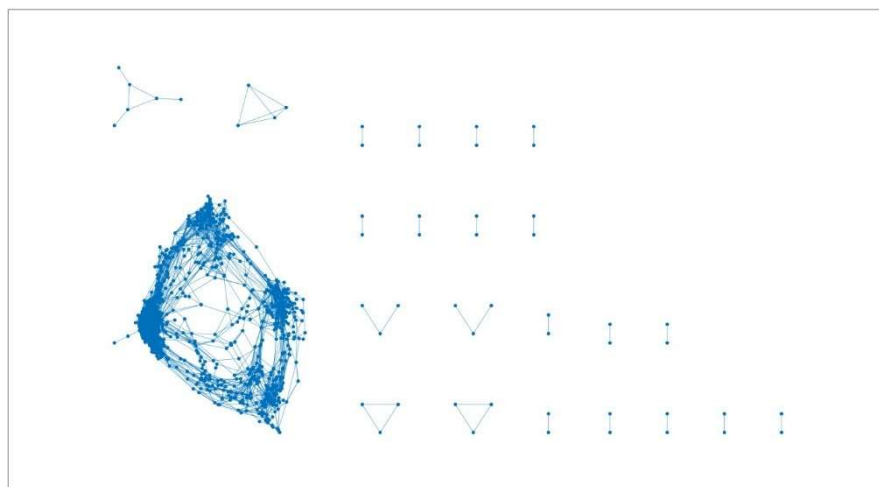


Figura 21 - Componentes conexos de wordlist-four-letters.txt

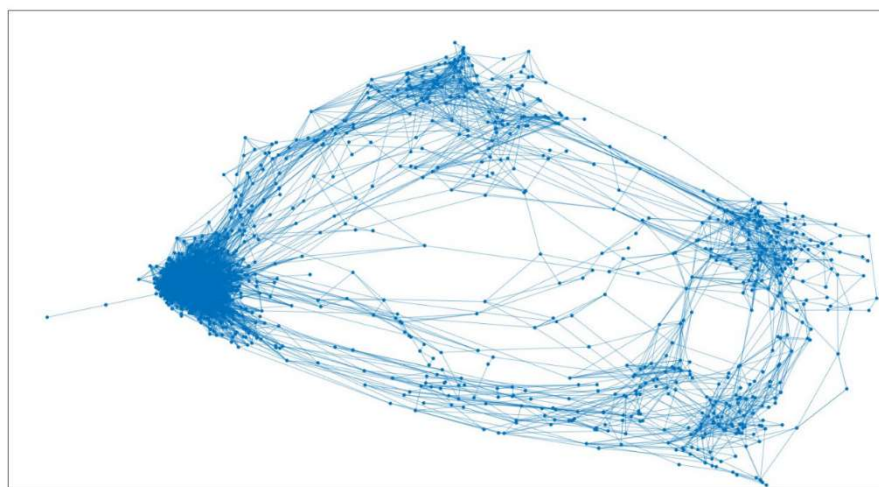


Figura 22 - Maior componente conexo de wordlist-four-letters.txt

Wordlist-five-letters

```
mendes@mendesze:~/Desktop/AED/Projeto2/A02$ ./word_ladder wordlist-five-letters.txt
Number of connected components: 575
Number of vertices: 7166
Number of edges: 23446
Largest Diameter: 33
Longest word ladder:
[0] expõe
[1] expie
[2] espie
[3] espio
[4] estio
[5] estão
[6] então
[7] entro
[8] entra
[9] extra
[10] exara
[11] exala
[12] exila
[13] axila
[14] afile
[15] afira
[16] atira
[17] ativa
[18] atava
[19] alava
[20] flava
[21] fiava
[22] fiara
[23] fibra
[24] vibra
[25] viera
[26] viela
[27] vi-la
[28] vi-lo
[29] vê-lo
[30] dê-lo
[31] dá-lo
[32] fá-lo
[33] fo-lo
Number of connected components with a diameter of:
0: 454
1: 84
2: 23
3: 9
4: 1
5: 2
6: 1
33: 1
Your wish is my command:
 1 WORD      (list the connected component WORD belongs to)
 2 FROM TO   (list the shortest path from FROM to TO)
 3           (terminate)
 4
```

Figura 23 - Saída inicial para Wordlist-five-letters

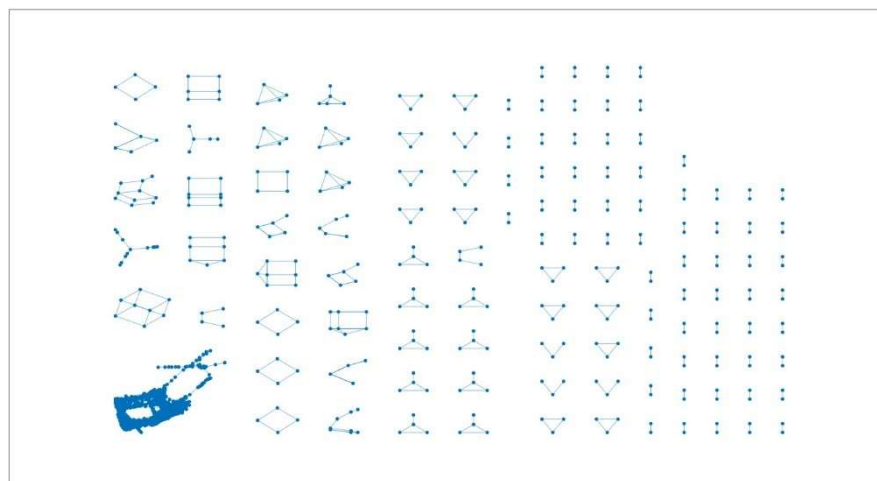


Figura 24 - Componentes conexos de Wordlist-five-letters

Wordlist-big-latest/ficheiro base

A obtenção da representação dos componentes conexos deste ficheiro não foi possível devido ao elevado número de palavras.

```
mendes@mendesze:~/Desktop/AED/Projeto2/A02$ ./word_ladder
Number of connected components: 377234
Number of vertices: 999282
Number of edges: 1060534
Largest Diameter: 92
Longest word ladder:
[0] visse-se
[1] visse-te
[2] viste-te
[3] veste-te
[4] deste-te
[5] despe-te
[6] desperte
[7] desperto
[8] desporto
[9] desponto
[10] desconto
[11] desconte
[12] descente
[13] descende
[14] descenda
[15] despenda
[16] despena
[17] dispensa
[18] dispenso
[19] distenso
[20] distendo
[21] distando
[22] discando
[23] riscando
[24] rascando
[25] lascando
[26] lassando
[27] passando
[28] pastando
[29] bastando
[30] bastardo
[31] bastarda
[32] bastaria
[33] bostaria
[34] tostaria
[35] tontaria
[36] contaria
[37] coutaria
[38] chutaria
[39] chuparia
[40] chaparia
[41] chavaria
[42] cravaria
[43] crivaria
[44] privaria
[45] provaria
[46] proveria
[47] proferia
[48] preferia
```

Figura 25 - Saída inicial para ficheiro base

```
Number of connected components with a diameter of:
0: 184869
1: 135815
2: 34797
3: 11578
4: 3026
5: 1445
6: 3574
7: 661
8: 344
9: 456
10: 122
11: 71
12: 67
13: 33
14: 15
15: 66
16: 43
17: 18
18: 39
19: 56
20: 25
21: 55
22: 20
23: 12
24: 5
25: 4
26: 2
27: 3
29: 1
33: 1
34: 2
35: 1
36: 3
46: 1
49: 1
57: 1
83: 1
92: 1
Your wish is my command:
 1 WORD      (list the connected component WORD belongs to)
 2 FROM TO   (list the shortest path from FROM to TO)
 3           (terminate)
> 
```

Figura 26 - Saída inicial para ficheiro base

Memory leaks

Por fim, foi realizado o teste para verificar a existência de *memory leaks* neste programa utilizando o programa *valgrind* com a opção *leak-check=full*:

```
=4544==
=4544==
=4544== HEAP SUMMARY:
=4544==   in use at exit: 0 bytes in 0 blocks
=4544== total heap usage: 22,848 allocs, 22,848 frees, 1,852,324,872 bytes allocated
=4544==
=4544== All heap blocks were freed -- no leaks are possible
=4544==
```

Figura 27 - Teste de memory leaks

Como é possível verificar, não são encontrados memory leaks, indicando então que todas as alocações de memória foram libertadas com sucesso.

Algumas Word ladders:

A opção 2 do menu do programa permite que seja encontrada, se possível, a menor distância entre duas palavras, por exemplo:

```

Your wish is my command:
  1 WORD      (list the connected component WORD belongs to)
  2 FROM TO   (list the shortest path from FROM to TO)
  3           (terminate)
> 2 bem mal
[0] bem
[1] bom
[2] som
[3] sol
[4] sal
[5] mal
```

Figura 28 - Menor distância entre "bem" e "mal"

```

Your wish is my command:
  1 WORD      (list the connected component WORD belongs to)
  2 FROM TO   (list the shortest path from FROM to TO)
  3           (terminate)
> 2 frio neve
[0] frio
[1] feio
[2] leio
[3] levo
[4] leve
[5] neve
```

Figura 29 - Menor distância entre "frio" e "neve"

```

Your wish is my command:
  1 WORD      (list the connected component WORD belongs to)
  2 FROM TO   (list the shortest path from FROM to TO)
  3           (terminate)
> 2 bebé xixi
[0] bebé
[1] bebe
[2] bibe
[3] bise
[4] vise
[5] vive
[6] vivi
[7] viii
[8] xiii
[9] xixi

```

Figura 30 - Menor distância entre "bebé" e "xixi"

```

Your wish is my command:
  1 WORD      (list the connected component WORD belongs to)
  2 FROM TO   (list the shortest path from FROM to TO)
  3           (terminate)
> 2 mão pés
[0] mão
[1] são
[2] sãs
[3] sés
[4] pés

```

Figura 31 - Menor distância entre "mão" e "pés"

Conclusão

O segundo trabalho prático garantiu que o grupo consolidasse conhecimentos em vários assuntos abordados ao longo da disciplina de Algoritmos e Estruturas de Dados, entre eles Hash Tables, Linked Lists, Ponteiros, Grafos e Search. Além disso, este trabalho fez com que o grupo descobrisse e utilizasse diversos conceitos e particularidades da Linguagem de Programação “C”, como, por exemplo, a alocação de memória. Contudo, concluiu-se que a implementação e alterações de Hash tables necessitam de extremo cuidado, uma vez que devem ser consideradas, entre outros aspectos, as colisões, a gestão de memória e o tratamento correto de endereços e ponteiros.

Portanto, com base nesse trabalho, é importante frisar que, de uma maneira eficiente, o grupo aprendeu e adquiriu experiência ao expandir, para além do campo da matemática, a utilização de grafos e em lidar com uma grande quantidade de dados.

Apêndice do código

Código C:

```
//  
  
// AED, November 2022 (Tomás Oliveira e Silva)  
  
//  
// Second practical assignement (speed run)  
  
//  
// Place your student numbers and names here  
//   N.Mec. 107188   Name: 105926  
  
//  
// Do as much as you can  
  
//   1) MANDATORY: complete the hash table code  
//       *) hash_table_create --> done  
//       *) hash_table_grow --> done  
//       *) hash_table_free --> done  
//       *) find_word --> done  
//       +) add code to get some statistical data about the hash table -->  
>TODO  
  
//   2) HIGHLY RECOMMENDED: build the graph (including union-find data) -- use  
//       the similar_words function...  
//       *) find_representative --> done  
//       *) add_edge --> done  
  
//   3) RECOMMENDED: implement breadth-first search in the graph  
//       *) breadth_first_search --> done  
  
//   4) RECOMMENDED: list all words belonginh to a connected component  
//       *) breadth_first_search --> done  
//       *) list_connected_component --> done  
  
//   5) RECOMMENDED: find the shortest path between to words  
//       *) breadth_first_search --> done  
//       *) path_finder --> done  
//       *) test the smallest path from bem to mal  
  
//           [ 0] bem  
//           [ 1] tem  
//           [ 2] teu  
//           [ 3] meu  
//           [ 4] mau
```

```

//      [ 5] mal
//      *) find other interesting word ladders
//      6) OPTIONAL: compute the diameter of a connected component and list the
longest word chain
//      *) breadth_first_search --> done
//      *) connected_component_diameter --> done
//      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>

```

```

// #define malloc(s)  ({ void *m = malloc(s); printf("M %3d %016lX
%d\n", __LINE__, (long)m, (long)(s)); m; })

// #define free(m)    ({ printf("F %3d %016lX\n", __LINE__, (long)m); free(m);
})

```

```

//
// 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 the next adjacency list node
    hash_table_node_t *vertex;       // the other vertex
};

struct hash_table_node_s
{
    // the hash table data
    char word[_max_word_size_];      // the word
    hash_table_node_t *next;         // next hash table linked list node
    // the vertex data
    adjacency_node_t *head;          // head of the linked list of adjacency
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 connected
component (only correct for the representative of each connected component)
    int number_of_edges;             // number of edges of the connected
component (only correct for the representative of each connected component)

    // added for graph information
    int biggest_diameter_of_cc;
};

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
};

static int size;

```

```

//
// allocation and deallocation of linked list nodes (done)
//

static adjacency_node_t *allocate_adjacency_node(void)
{
    adjacency_node_t *node;

    node = (adjacency_node_t *)malloc(sizeof(adjacency_node_t));
    if(node == NULL)
    {
        fprintf(stderr, "allocate_adjacency_node: out of memory\n");
        exit(1);
    }
    return node;
}

static void free_adjacency_node(adjacency_node_t *node)
{
    free(node);
}

static hash_table_node_t *allocate_hash_table_node(void)
{
    hash_table_node_t *node;

    node = (hash_table_node_t *)malloc(sizeof(hash_table_node_t));
    if(node == NULL)
    {
        fprintf(stderr, "allocate_hash_table_node: out of memory\n");
        exit(1);
    }
    return node;
}

static void free_hash_table_node(hash_table_node_t *node)
{

```

```

    free(node);
}

//
// hash table stuff (mostly to be done)
//

unsigned int crc32(const char *str)
{
    static unsigned int table[256];
    unsigned int crc;

    if(table[1] == 0u) // do we need to initialize the table[] array?
    {
        unsigned int i,j;

        for(i = 0u; i < 256u; i++)
            for(table[i] = i, j = 0u; j < 8u; j++)
                if(table[i] & 1u)
                    table[i] = (table[i] >> 1) ^ 0xAED00022u; // "magic" constant
                else
                    table[i] >>= 1;
    }
    crc = 0xAED02022u; // initial value (chosen arbitrarily)
    while(*str != '\0')
        crc = (crc >> 8) ^ table[crc & 0xFFu] ^ ((unsigned int)*str++ << 24);
    return crc;
}

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

    hash_table = (hash_table_t *)malloc(sizeof(hash_table_t));
    if(hash_table == NULL)

```

```

{
    fprintf(stderr, "create_hash_table: out of memory\n");
    exit(1);
}

//
// complete this
//

hash_table->number_of_entries = 0;

hash_table->hash_table_size = 100; // tamanho escolhido pois é o tamanho do
file mais pequeno

hash_table->number_of_edges = 0;

hash_table->heads = (hash_table_node_t **)malloc(sizeof(hash_table_node_t) *
hash_table->hash_table_size); // como é um array de ponteiros é ponteiro para
ponteiro

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

for(i = 0; i < hash_table->hash_table_size; i++)
{
    hash_table->heads[i] = NULL; // deixar todos os heads a NULL para poder
serem preenchidos

    // printf("%s", hash_table->heads[i]->word); // ok faz sentido ne, agora,
usar isto na main perceber o que se esta a passar e tentar por o grow
}

return hash_table;
}

static void hash_table_grow(hash_table_t *hash_table) // quando a tabela
cresce, os itens ate entao colocados precisam de ser reentrados

{
    //
    // complete this
    //

    unsigned int old_size, i, j;

    static hash_table_node_t **old_heads;

```

```

hash_table_node_t *n, *nn;

old_size = hash_table->hash_table_size;
old_heads = hash_table->heads;

hash_table->hash_table_size *= 1.5 + 1; // incrementa mais metade do tamanho

hash_table->heads = (hash_table_node_t **)malloc(sizeof(hash_table_node_t) *
hash_table->hash_table_size); // como é um array de ponteiros é ponteiro para
ponteiro

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

    hash_table->heads[i] = NULL;

for(i = 0; i < old_size; i++)

    for(n = old_heads[i]; n != NULL; n = nn)

        {

            nn = n->next;

            j = crc32(n->word) % hash_table->hash_table_size;

            n->next = hash_table->heads[j];

            hash_table->heads[j] = n;

            old_heads[i] = NULL; // para conseguir dar free

        }

    free(old_heads);
}

static void hash_table_free(hash_table_t *hash_table)

{

    unsigned int i;

    hash_table_node_t* temp;

    adjacency_node_t* adj_temp;

    //

    // complete this

    //

    for(i = 0; i < hash_table->hash_table_size; i++) // para limpar a hashtable

    {

        temp = hash_table->heads[i]; // para o inicio daquele indice

        while(temp != NULL)

        {

```

```

    adj_temp = temp->head;
    while(adj_temp != NULL)
    {
        temp->head = temp->head->next;
        free_adjacency_node(adj_temp);
        adj_temp = temp->head;
    }

    hash_table->heads[i] = hash_table->heads[i]->next; // avança para o
    proximo para tambem o libertar os nodes

    free_hash_table_node(temp);
    temp = hash_table->heads[i];
}
}

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 *head_cpy, *prev;
    hash_table_node_t *node; // é o principal que possui tudo

    // we need to allocate memory
    unsigned int i;

    i = crc32(word) % hash_table->hash_table_size;
    for(node = hash_table->heads[i]; node != NULL && strcmp(node->word, word) !=
0; node = node->next)
    ;

    if(node == NULL && insert_if_not_found != 0)
    {
        node = allocate_hash_table_node();
        strcpy(node->word, word);
        node->next = hash_table->heads[i];
        hash_table->heads[i] = node;
        node->representative = node;
        node->visited = 0;
        node->previous = NULL; // nao possui anteriores pois é o primeiro

```

```

        node->number_of_vertices = 1;
        node->number_of_edges = 0;
        node->biggest_diameter_of_cc = 0;

        hash_table->number_of_entries = hash_table->number_of_entries + 1;
    }

    if(hash_table->number_of_entries*2 > hash_table->hash_table_size){ // se a
tabela ja estiver meio cheia

        hash_table_grow(hash_table);
    }

    return node;
}

```

```

//
// add edges to the word ladder graph (mostly do be done)
//

```

```

static hash_table_node_t *find_representative(hash_table_node_t *node)
{
    hash_table_node_t *n1,*n2, *n3;

    //
    // complete this
    //

    for(n1 = node; n1->representative != n1; n1 = n1->representative)
        ;

    for(n2 = node; n2 != n1; n2 = n3)
    {
        n3 = n2->representative;
        n2->representative = n1;
    }

    return n1;
}

```

```

static void add_edge(hash_table_t *hash_table,hash_table_node_t *from,const
char *word)
{

```

```

hash_table_node_t *to,*from_representative,*to_representative;
adjacency_node_t *link;

//
// complete this
//

to = find_word(hash_table,word,0);
if(to == NULL) // word nao existe
    return;

hash_table->number_of_edges++;

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

// printf("%s %s\n", from->word, to->word); // to get information for
matlab graph

// temos de criar as ligações entre o from e o to
// from
link = allocate_adjacency_node(); // reserva a memoria para poder ser usado
link->vertex = to;
link->next = from->head;
from->head = link;

// to
link = allocate_adjacency_node(); // reserva a memoria para poder ser usado
link->vertex = from;
link->next = to->head;
to->head = link;

if(from_representative == to_representative) // têm o mesmo representativo
logo so precisava da ligação
    return;

if(from_representative->number_of_vertices < to_representative-
>number_of_vertices)

```



```

    {
        to_representative->number_of_edges = to_representative->number_of_edges +
1 + from_representative->number_of_edges;

        to_representative->number_of_vertices += from_representative-
>number_of_vertices;

        from_representative->representative = to_representative;
    }

    else{
        from_representative->number_of_edges = from_representative-
>number_of_edges + 1 + to_representative->number_of_edges;

        from_representative->number_of_vertices += to_representative-
>number_of_vertices;

        to_representative->representative = from_representative;
    }
}

```

```

//

// generates a list of similar words and calls the function add_edge for each
one (done)

//

// man utf8 for details on the uft8 encoding

//

```

```

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

    while(*word != '\0')
    {
        byte0 = (int)(*(word++)) & 0xFF;
        if(byte0 < 0x80)
            *(individual_characters++) = byte0; // plain ASCII character
        else
        {
            byte1 = (int)(*(word++)) & 0xFF;

            if((byte0 & 0b11100000) != 0b11000000 || (byte1 & 0b11000000) !=
0b10000000)
            {

```

```

        fprintf(stderr, "break_utf8_string: unexpected UTF-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 UTF-8 character\n");
            exit(1);
        }
    }

    *word = '\0'; // mark the end
}

```

```

static void similar_words(hash_table_t *hash_table, hash_table_node_t *from)
{
    static const int valid_characters[] =
    { // unicode!

```

```

        0x2D,
// -

        0x41,0x42,0x43,0x44,0x45,0x46,0x47,0x48,0x49,0x4A,0x4B,0x4C,0x4D,
// A B C D E F G H I J K L M

        0x4E,0x4F,0x50,0x51,0x52,0x53,0x54,0x55,0x56,0x57,0x58,0x59,0x5A,
// N O P Q R S T U V W X Y Z

        0x61,0x62,0x63,0x64,0x65,0x66,0x67,0x68,0x69,0x6A,0x6B,0x6C,0x6D,
// a b c d e f g h i j k l m

        0x6E,0x6F,0x70,0x71,0x72,0x73,0x74,0x75,0x76,0x77,0x78,0x79,0x7A,
// n o p q r s t u v w x y z

        0xC1,0xC2,0xC9,0xCD,0xD3,0xDA,
// Á Â É Í Ó Ú

0xE0,0xE1,0xE2,0xE3,0xE7,0xE8,0xE9,0xEA,0xED,0xEE,0xF3,0xF4,0xF5,0xFA,0xFC, //
à á â ã ç è é ê í î ó ô õ ú ü

    0

};

int i,j,k,individual_characters[_max_word_size_];
char new_word[2 * _max_word_size_];

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

//

// breadth-first search (to be done)

//

```

```

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

//

static 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
    //

    // supondo que os vertices ja estao no list_of_vertices:
    int write, read, found_flag, count, clear;
    adjacency_node_t *node;
    hash_table_node_t *go_back_node;

    write = 1; read = 0, found_flag = 0;
    origin->visited = 1;
    origin->previous = NULL; // garantir que esta NULL para fazer o search mais
tarde
    list_of_vertices[0] = origin; // o primeiro é o from e vamos procurar o to
    node = list_of_vertices[0]->head; // a primeira palavra adjacente á origin

    while(read < maximum_number_of_vertices) // para ler todas as palavras na
lista se necessário
    {
        while(node != NULL)
        {
            if(node->vertex->visited == 1) // ja esteve neste node, nao vai ca estar
again
            {
                node = node->next;
                continue;
            }

            node->vertex->visited = 1; // foi visitado;

            node->vertex->previous = list_of_vertices[read]; // definir o node
anterior para mais tarde conseguir descobrir o caminnho

            list_of_vertices[write] = node->vertex;

```

```

    if(node->vertex == goal) // encontramos a solução
    {
        found_flag = 1;
        break;
    }

    write++;

    node = node->next; // avança para o proximo adjacnete
}

// vai sempre encontrar algures uma vez que a palavra esta no mesmo
componente conexo

if(found_flag == 1) // encontrou o goal --> é o node->vertex
{
    // limpar os visited

    for(clear = 0; clear <= write; clear++)

        list_of_vertices[clear]->visited = 0;

    // determinar o pretendido

    count = 1; // começa com um que é no que está

    go_back_node = node->vertex; // node que vai sendo atualizado com o
previous para encontrar o caminho de volta

    while(go_back_node->previous != NULL){

        count++;

        go_back_node = go_back_node->previous;

    }

    return count;

}

read++; // avança para o proximo elemento a ler --> seguinfo as regras
deste

if(list_of_vertices[read] == NULL)

    break;

    node = list_of_vertices[read]->head;

}

for(clear = 0; clear < write; clear++)

    list_of_vertices[clear]->visited = 0;

return -1;

}

```

```

//
// list all vertices belonging to a connected component (complete this)
//

static void list_connected_component(hash_table_t *hash_table, const char
*word) //nao sei se ja esta --> nem esta nem a de cima
{
    //
    // complete this
    //
    // usar o array dado pelo bfs inteiro
    hash_table_node_t **list_of_vertices;
    int maximum_number_of_vertices;
    unsigned int i;
    hash_table_node_t *origin, *goal, *node;

    list_of_vertices = (hash_table_node_t **)malloc(sizeof(hash_table_node_t) *
hash_table->hash_table_size);
    maximum_number_of_vertices = hash_table->hash_table_size;
    origin = find_word(hash_table, word, 0);
    if(origin == NULL)
        return;
    goal = NULL;

    breath_first_search(maximum_number_of_vertices, list_of_vertices, origin,
goal);

    i = 0;
    node = list_of_vertices[i];
    while(node != NULL)
    {
        printf("%s\n", node->word);
        list_of_vertices[i] = NULL;
        i++;
        node = list_of_vertices[i];
    }
    free(list_of_vertices);
}

```

```

}

//

// compute the diameter of a connected component (optional)
//

static int largest_diameter;
static hash_table_node_t **largest_diameter_example;

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

    // array completo como o de cima e vai ver quantos previous tem do ultimo
    até ao primeiro --> assim da o maior

    // com o representante talvez
    //
    // complete this
    //

    hash_table_node_t **list_of_vertices;
    int maximum_number_of_vertices;

    list_of_vertices = (hash_table_node_t **)malloc(sizeof(hash_table_node_t) *
size); // size é um static int definido em cima e que obtem o valor de
hash_table_size na main

    maximum_number_of_vertices = size;

    breadth_first_search(maximum_number_of_vertices, list_of_vertices, node,
NULL); // vai ter o maior caminho em nós logo é so ver ate ao final

    int i = 0;
    while(list_of_vertices[i] != NULL){
        node = list_of_vertices[i]; // usar o node agora para guardar o valor do
último nó do array

        list_of_vertices[i] = NULL; // limpar o array

        i++;
    }

    diameter = 0;

```

```

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

free(list_of_vertices); // libertar a memoria usada pelo array
return diameter;
}

//
// find the shortest path from a given word to another given word (to be done)
//

static void path_finder(hash_table_t *hash_table, const char *from_word, const
char *to_word)
{
    //
    // complete this
    //
    // acho que basta usar os previous do to ate ao from porque o bfs ja o faz
    hash_table_node_t **list_of_vertices, *from, *to, *from_representative,
    *to_representative;

    int maximum_number_of_vertices, count, word_number;

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

    maximum_number_of_vertices = hash_table->hash_table_size;

    from = find_word(hash_table, from_word, 0);
    to = find_word(hash_table, to_word, 0);
    if(from == NULL || to == NULL)
    {
        printf("Palavra inexistente\n");
        return;
    }

    from_representative = find_representative(from);

```



```

to_representative = find_representative(to);
// printf("%s == %s\n", from_representative->word, to_representative->word);
if(from_representative != to_representative)
{
    printf("Palavras introduzidas não apresentam word ladder\n");
    return;
}

breadh_first_search(maximum_number_of_vertices, list_of_vertices, from, to);

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

count = 0;

while(to != NULL)
{
    previous_words[count] = to;
    to = to->previous;
    count++;
}

word_number = 0;
count--;
while(count >= 0)
{
    printf("[%d] %s\n", word_number, previous_words[count]->word);
    count--;
    word_number++;
}

free(list_of_vertices); free(previous_words);
}

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

static void graph_info(hash_table_t *hash_table)
{

```

```

//
// complete this
//

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

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

hash_table_node_t *node, *node_representative, *goal;

unsigned int i, h;

int cc, vertices, j, diameter, store_ld_index, ind, all_diameter_counter;

cc = 0, vertices = 0, largest_diameter = 0, store_ld_index = -1, ind = 0;

goal = NULL;

for(i = 0u; i < hash_table->hash_table_size; i++){
    for(node = hash_table->heads[i]; node != NULL; node = node->next){
        node_representative = find_representative(node);

        for(h = 0u; h < hash_table->hash_table_size; h++){ // para guardar o
numero de cc e tudo mais

            if(node_representative == representatives[h])
                break;

            if(h == hash_table->hash_table_size - 1){ // caso ainda nao tenha visto
o representante

                representatives[cc] = node_representative;

                cc++;

                node_representative->visited = 1;

                vertices += node_representative->number_of_vertices;

            }
        }

        diameter = connected_component_diameter(node); // encontra o diametro
daquele cc

        largest_diameter_example[ind] = node;

        if(diameter > largest_diameter){
            largest_diameter = diameter;

            store_ld_index = ind;

        }
    }
}

```

```

        if(diameter > node_representative->biggest_diameter_of_cc)
            node_representative->biggest_diameter_of_cc = diameter;

        ind++;
    }
}

int all_diameters[largest_diameter+1];
for(int a = 0; a < largest_diameter+1; a++)
    all_diameters[a] = 0;

// vertices, edges, cc's uma cena assim // maximo vertices e edces num
determinado cc

printf("Number of connected components: %d\n", cc);
printf("Number of vertices: %d\n", vertices);
printf("Number of edges: %d\n", hash_table->number_of_edges);
printf("Largest Diameter: %d\n", largest_diameter);

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

int maximum_number_of_vertices = hash_table->hash_table_size;

breadh_first_search(maximum_number_of_vertices, list_of_vertices,
largest_diameter_example[store_ld_index], NULL);

int count = 0;

while (list_of_vertices[count] != NULL)
{
    goal = list_of_vertices[count]; // tem o intuito de guardar o ultimo
    list_of_vertices[count] = NULL; // vai limpando
    count++;
}

printf("Longest word ladder:\n");

path_finder(hash_table, largest_diameter_example[store_ld_index]->word,
goal->word);

j = 0;
while(representatives[j] != NULL){
    all_diameters[representatives[j]->biggest_diameter_of_cc] += 1;
    representatives[j]->visited = 0;
}

```

```

        largest_diameter_example[j] = NULL; // têm ambos o mesmo comprimento
        j++;
    }

    printf("Number of connected components with a diameter of:\n");

    for(all_diameter_counter = 0; all_diameter_counter < largest_diameter+1;
        all_diameter_counter++){

        if(all_diameters[all_diameter_counter] != 0)

            printf("%d: %d\n", all_diameter_counter,
                all_diameters[all_diameter_counter]);

    }

    free(representatives); free(largest_diameter_example);
    free(list_of_vertices);

}

//
// main program
//

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

    // initialize hash table
    hash_table = hash_table_create();

    // read words
    fp = fopen((argc < 2) ? "wordlist-big-latest.txt" : argv[1], "rb");
    if(fp == NULL)
    {
        fprintf(stderr, "main: unable to open the words file\n");
        exit(1);
    }

```

```

}

while(fscanf(fp,"%99s",word) == 1){
    (void)find_word(hash_table,word,1);
}

// int num_words = 0;
// for(i = 0u;i < hash_table->hash_table_size;i++){
//     for(node = hash_table->heads[i];node != NULL;node = node->next){
//         num_words++;
//         printf("%d-> %s;%d --- ", i, node->word, num_words);
//     }
//     printf("\n");
// }

// printf("%d", hash_table->hash_table_size);
fclose(fp);

size = hash_table->hash_table_size;
// 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(i = 0u;i < hash_table->hash_table_size;i++){
//     for(node = hash_table->heads[i];node != NULL;node = node->next){
//         printf("%s --> ", node->word);
//         while(node->head != NULL){
//             printf("%s -- ", node->head->vertex->word);
//             node->head = node->head->next;
//         }
//         printf("\n");
//     }
// }

graph_info(hash_table);
// ask what to do
for(;;)

```

```

{
    fprintf(stderr, "Your wish is my command:\n");
    fprintf(stderr, " 1 WORD          (list the connected component WORD belongs
to)\n");
    fprintf(stderr, " 2 FROM TO      (list the shortest path from FROM to
TO)\n");
    fprintf(stderr, " 3              (terminate)\n");
    fprintf(stderr, "> ");
    if (scanf("%99s", word) != 1)
        break;
    command = atoi(word);
    if (command == 1)
    {
        if (scanf("%99s", word) != 1)
            break;
        list_connected_component(hash_table, word);
    }
    else if (command == 2)
    {
        if (scanf("%99s", from) != 1)
            break;
        if (scanf("%99s", to) != 1)
            break;
        path_finder(hash_table, from, to);
    }
    else if (command == 3)
        break;
    }
    // clean up
    hash_table_free(hash_table);
    return 0;
}

```

Código Matlab:

A obtenção das restantes representações do grafos foi obtida utilizando o mesmo código, apenas alternado o ficheiro utilizado.

```
file_id = fopen("graph_fourword_data.txt");  
all_nodes = textscan(file_id, "%s %s");  
node_A = all_nodes{:,1}';  
node_B = all_nodes{:,2}';  
G = graph(node_A,node_B);  
plot(G)
```

Figura 32 - Código Matlab