



# WORD LADDER

## ALGORITMOS E ESTRUTURAS DE DADOS

### Trabalho realizado por:

- Pedro Ramos > 34%  
> N.º Mec: 107348
- Daniel Madureira > 33%  
> N.º Mec: 107603
- Rafael Kauati > 33%  
> N.º Mec: 105925



universidade  
de aveiro



**deti**  
universidade de aveiro



# ÍNDICE

INTRODUÇÃO.....	3
OBJETIVO DO TRABALHO .....	3
PROPOSIÇÃO DO PROBLEMA.....	4
FUNÇÕES MODIFICADAS .....	5
ANÁLISE DAS PRINCIPAIS ESTRUTURAS DE DADOS .....	6
PONTEIROS .....	6
LINKED LISTS.....	6
HASH TABLES .....	7
CONNECTED COMPONENTS .....	8
ALGORITMOS DE GERAÇÃO DAS ESTRUTURAS PRINCIPAIS .....	9
HASH TABLE .....	9
COMPONENTES CONEXOS.....	12
IMPLEMENTAÇÃO DAS OPÇÕES.....	16
WORD.....	17
FROM TO.....	18
DISPLAY HASH TABLE .....	20
DISPLAY HASH TABLE INFO .....	21
DISPLAY GRAPH .....	22
DISPLAY GRAPH INFO.....	24
TERMINATE.....	25
FUNÇÕES SECUNDÁRIAS .....	27
setnNodesVisitedTo0.....	27
progressBar.....	28
RESULTADOS.....	29
ALOCAÇÕES E MEMORY LEAKS .....	29
CONCLUSÃO .....	30
WEBGRAFIA.....	31
APÊNDICE .....	32
CÓDIGO EM C .....	32



# INTRODUÇÃO

## OBJETIVO DO TRABALHO

O objetivo deste trabalho é desenvolver uma série de algoritmos na linguagem C com o objetivo de agrupar elementos e interconectá-los de forma eficiente nas estruturas de dados que nos foram propostas.

Especificamente, o conjunto de elementos a tratar é formado por um grupo de palavras da língua portuguesa que serão conectadas a partir de um algoritmo que identifica a semelhança de caracteres entre duas palavras.

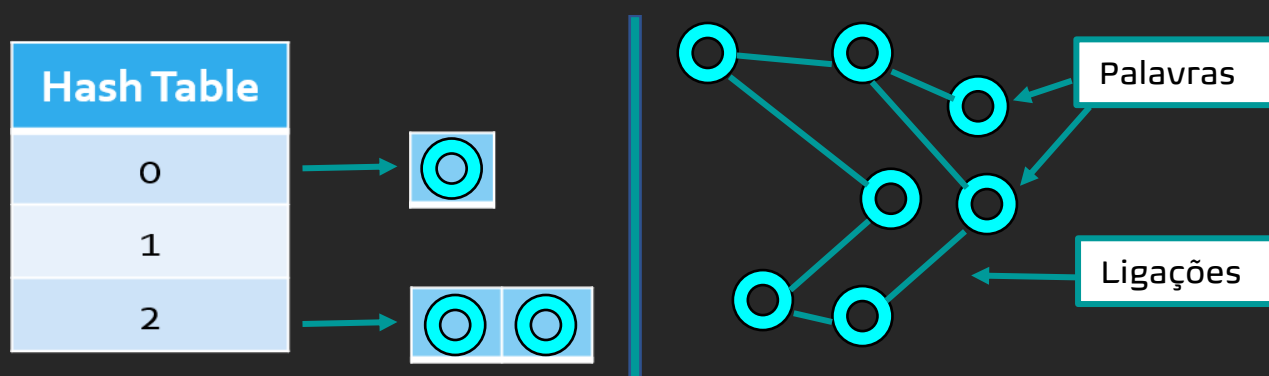
Este projeto procura desenvolver as nossas competências de criação de pseudo-estruturas de dados, assim como a sua manipulação eficiente e sem repetição, procurando aplicar sempre boas práticas de programação, como por exemplo, libertar toda a memória alocada ao longo do projeto.

# PROPOSIÇÃO DO PROBLEMA

Antes de começar com qualquer tipo de processamento de palavras, temos de as colocar numa estrutura à qual possamos aceder rapidamente.

Normalmente, seriam utilizados vetores ou *arrays*, mas isto levaria a que tivéssemos de iterar sobre todo o *array* à procura da palavra pretendida, o que seria extremamente lento para os ficheiros de grandes dimensões (mais de 2 milhões de palavras) que pretendemos analisar.

Logo, a melhor solução é inserir as palavras numa *hash table* para termos um acesso mais rápido e estabelecer as relações entre palavras numa pseudo-estrutura denominada *componente conexo* (um grafo). Posteriormente, colocamos todos os componentes conexos num conjunto de forma a que cada palavra esteja presente num único componente conexo.



Assim, cada palavra é representada por um *node* que é acedido através de uma *hash table*, ou seja, cada palavra terá uma *key* associada que representa o seu índice na *hash table*. A *key* é um valor numérico criptografado através de uma função de *hash*.

A *hash table* será do tipo *separate chaining*. Este tipo de *hash table* é usado para resolver problemas de colisões entre palavras inertes a este método de *hash*, uma vez que elementos com a mesma *key* são inseridos numa *linked list*, podendo aceder a estes elementos através dessa *key*.

# FUNÇÕES MODIFICADAS

Para que o código do projeto funcione da forma mais completa possível foram implementados/ completados algoritmos nas seguintes funções dadas:

- hash\_table\_create
- hash\_table\_grow
- hash\_table\_free
- find\_word
- find\_representative
- add\_edge
- path\_finder
- breadth\_first\_search
- free\_adjacency\_node
- allocate\_hash\_table\_node
- free\_hash\_table\_node

Foram também criadas algumas funções de apoio. Estas funções têm como finalidade melhorar a interação entre o programa e o utilizador:

- hash\_table\_info
- graph\_info
- calculate\_info
- progresso\_bar
- setNodesVisitedTo0

# ANÁLISE DAS PRINCIPAIS ESTRUTURAS DE DADOS

## PONTEIROS

Ponteiro nada mais são do que um índice de memória que aponta para uma variável guardada nesse endereço.



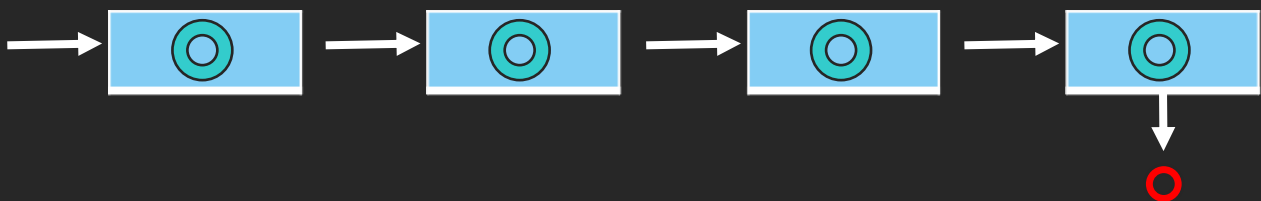
Iremos usá-los extensivamente ao longo deste projeto.

## LINKED LISTS

Apesar da sua simplicidade, as *linked lists* são a estrutura de dados fundamental deste, e de outros tantos projetos.

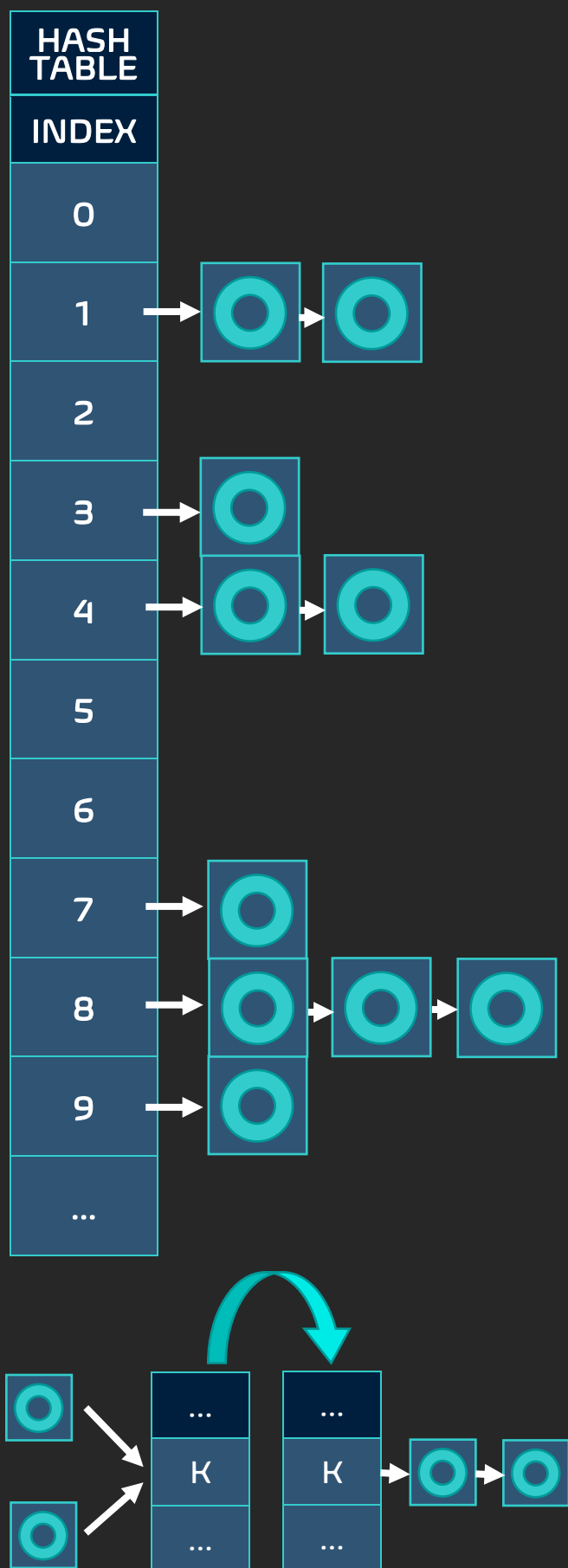
Esta estrutura permite inserir, alterar e remover dados de forma simples e muito eficiente.

Na sua essência, uma *linked list* não passa de uma lista de *nodes* que tem informação guardada assim como um ponteiro para o próximo *node* da lista. Existem vários tipos de listas ligadas, como as *double linked list*, que para além de terem um ponteiro para o próximo *node* também têm um ponteiro para o *node* anterior, sendo possível recuar na lista sem ter de a percorrer novamente.



Sabemos que estamos no fim da lista quando o ponteiro para o próximo *node* tem o valor NULL (por convenção).

# HASH TABLES



Antes de começar a construir os componentes conexos e analisar as conexões entre as palavras, será preciso colocar cada palavra numa *hash table*.

Esta estrutura será a nossa principal forma de guardar, aceder e comparar as palavras do ficheiro, fazendo com que o acesso a cada palavra seja extremamente eficiente (complexidade computacional de ordem  $O(1)$ ).

Sem esta estrutura seria preciso iterar sobre todos os elementos do ficheiro até se encontrar o que se procura (complexidade computacional de ordem  $O(n)$ ).

Esta *hash table* é do tipo *Separated Chaining hash table*.

Como os índices de cada palavra são gerados através de uma *hash function*, que não é perfeita, existe uma possibilidade (bastante pequena) de que várias palavras gerem o mesmo *hash code*, ou seja, de que se sobreponham na tabela.

Isto leva a colisões, ou seja, os dados podem ser sobrescritos por outros mais recentes.

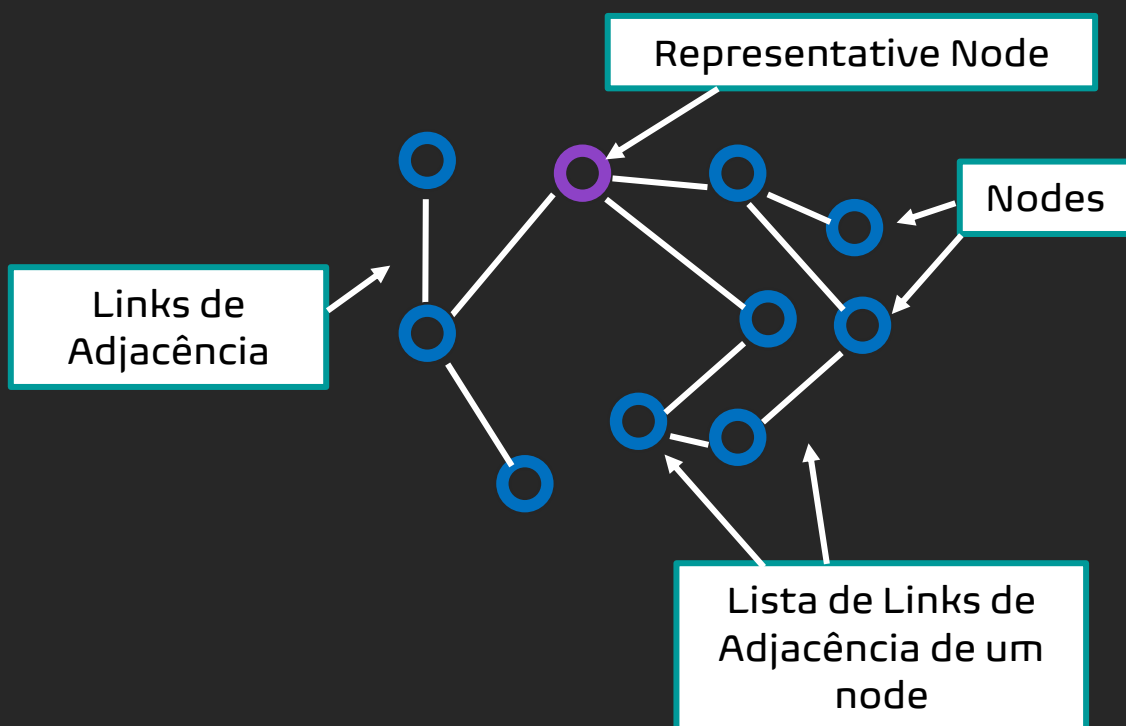
*Separate chaining* significa que cada entrada da *hash table* é uma *linked list*, onde todas as palavras com o mesmo *hash code* são colocadas na lista ligada correspondente ao *hash code*, evitando sobrescrita de dados.

Este método de evitar sobrescrita de dados reduz a eficácia da *hash table*, uma vez que temos de percorrer uma lista ligada após calcular o *hashcode* da palavra pretendida. No entanto, a lista gerada com as colisões é bastante pequena (geralmente na ordem de 2-20 palavras), não se comparando com o tempo que seria gasto a percorrer uma lista de, por exemplo, 2 milhões de *nodes*, o que seria o caso se não usássemos uma *hash table*.

## CONNECTED COMPONENTS

Após criar os *nodes* e de os colocar na *hash table* para um acesso mais rápido, podemos prosseguir para a próxima estrutura de dados, sendo esta, a que vai solucionar o problema final.

A estrutura que nos foi sugerida, e que vamos utilizar, é um componente conexo, ou seja, um grafo. Esta estrutura é bastante simples, sendo composta por *nodes*, gramaticalmente semelhantes, por *links*, que mostram o relacionamento entre estes, e pelos *links* de adjacência, que realizam a ligação entre os *nodes* já gravados na *hash table*. Cada *node* tem a sua lista de *adjacency nodes*.



O node representativo é o *node* "responsável" pelo seu componente conexo. Serve apenas para cálculos internos, mas é de notar que apenas este *node* contém a informação correta do seu componente conexo.



# ALGORITMOS DE GERAÇÃO DAS ESTRUTURAS PRINCIPAIS

## HASH TABLE

Inicialmente, o “esqueleto” da *hash table* é criado com a função `hash_table_create()`. Esta função cria uma variável do tipo `hash_table_t` que contém o tamanho da tabela, informação sobre a mesma, e o mais importante, um vetor cujos índices correspondem aos *hash codes* calculados pela função de *hash* e os valores correspondem ao ponteiro para o primeiro *node* de cada *linked lists*. Este array é alocado com o tamanho do ponteiro vezes o tamanho da *hash table*, e posteriormente preenchido com `NULL`.

```
static hash_table_t *hash_table_create(void)
{
    hash_table_t *hash_table;

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

    hash_table->hash_table_size = 200;
    hash_table->number_of_entries = 0;
    hash_table->number_of_edges = 0;

    int sizeofHashTableNode = sizeof(hash_table->heads);

    hash_table->heads = (unsigned int*) malloc(hash_table->hash_table_size * sizeofHashTableNode);
    memset(hash_table->heads, NULL, hash_table->hash_table_size * sizeofHashTableNode);

    return hash_table;
}
```

Após realizarmos alguns testes, concluímos que o tamanho inicial ideal para a nossa *hash table* era de 200, visto que este tamanho causava menos colisões sem alocar demasiado espaço “inútil” no *array*.

Após a criação da *hash table*, o ficheiro de texto inserido é lido de forma a conseguirmos retirar palavra a palavra.

```
// 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);
}

printf("\n Filling up the hash table...\n");
while(fscanf(fp, "%99s", word) == 1) {
    (void)find_word(&hash_table, word, 1);
}
```

De seguida, a função `find_word()` é aplicada a cada palavra. Esta função começa por verificar a percentagem de *hash table* que já foi preenchida. Se este valor for alto (por exemplo, maior que 50%), a quantidade de colisões irá aumentar exponencialmente. Por exemplo, ao tentar colocar um conjunto de 100 palavras numa *hash table* com tamanho 25, irão ocorrer, em média, 4 colisões por *hash code*.

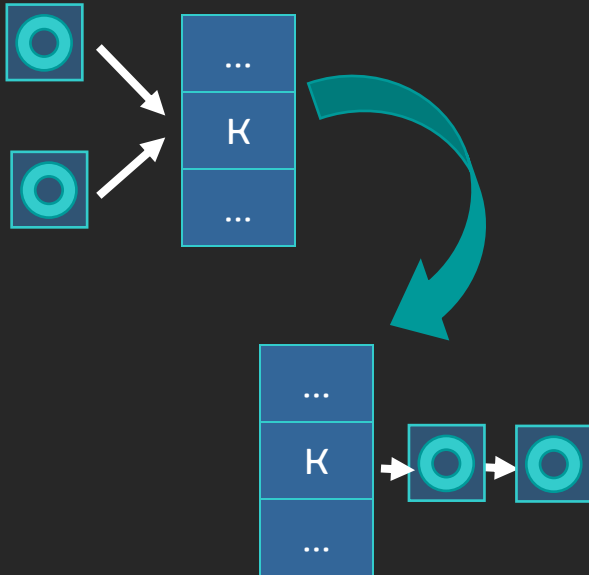
Para prevenir isto, temos de aumentar o tamanho da *hash table* dinamicamente em relação ao número de palavras inseridas.

Nesta função, é gerado o *hash code* de cada palavra, que será depois inserida na entrada da *hash table* correspondente. A palavra é então colocada num *node*, que serve como uma espécie de "contentor" que retém a palavra e várias informações sobre esta.

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

    // Verificar o preenchimento da hash table
    if ((*hash_table)->number_of_entries >= (*hash_table)->hash_table_size / 2) {
        hash_table_grow(*hash_table);
    }
    // Obter o hash code da palavra
    hashVal = crc32(word) % (*hash_table)->hash_table_size;
```

Se a entrada selecionada já estiver preenchida, a função percorre a lista até ao fim e coloca a palavra após a última posição, ligando-a à palavra anterior.



```
// Se a operação for de insert
if (insert_if_not_found == 1) {
    // Criar o Node
    hash_table_node_t *node = allocate_hash_table_node();
    node->next = NULL;
    node->head = NULL;
    node->visited = 0;
    node->representative = node;
    node->number_of_edges = 0;
    node->number_of_vertices = 1;

    // Se não existir um Node nesse hash value
    if ((*hash_table)->heads[hashVal] == NULL) {
        strcpy(node->word, word);

        (*hash_table)->heads[hashVal] = node;
        (*hash_table)->number_of_entries++;
    }
    // Se já existir um Node nesse hash value
    else {
        totalColisions++;
        hash_table_node_t *last_node;
        // Percorrer a lista até ao fim
        last_node = (*hash_table)->heads[hashVal];
        while (last_node->next != NULL)
        {
            last_node = last_node->next;
        }
        last_node->next = node;
        strcpy(node->word, word);
    }
    return NULL;
}
```

Esta função também pode ser chamada de forma a que não insira a palavra caso esta não exista, o que será útil para aceder aos *nodes* sem alterar a *hash table*.

Para isso, basta modificar o valor passado à função no argumento `insert_if_not_found`.

```
// Caso não seja para inserir nodes novos
else {
    hash_table_node_t *node;
    // Caso o hash value corresponda a algum node
    if ((*hash_table)->heads[hashVal] != NULL) {
        node = (*hash_table)->heads[hashVal];
        // Percorrer a lista de nodes
        // até encontrar o pretendido
        while (node != NULL) {
            if (strcmp(node->word, word) == 0) {
                return node;
            }
            else {
                node = node->next;
            }
        }
    }
    return NULL;
}
```

Para incrementar o tamanho de uma *hash table*, é necessário realizar o "*rehashing*" de todos os *nodes* da *hash table* original. Ou seja, é preciso recalculer o *hash code* de cada *node*, pois este depende do tamanho da *hash table* em que foi gerado.

Logo, a função `hash_table_grow()` começa por criar e alocar espaço para um novo *array* da *hash table*, que depois preenche ao iterar sobre os elementos da *hash table* original e recalculer o seu *hash code*.

Isto podia ser feito ao criar uma nova variável do tipo *hash table* e utilizar a função já existente `find_word()`, mas preferimos utilizar uma versão modificada desta função em nome da eficiência do código.

```
static void hash_table_grow(hash_table_t *hash_table)
{
    unsigned int hashVal;
    hash_table_node_t *node;
    hash_table_node_t *next_node;

    hash_table->hash_table_size = hash_table->hash_table_size * 2;
    totalGrows++;
    // Alocar o novo array de hash table
    hash_table_node_t **new_heads = (unsigned int*) malloc(hash_table->hash_table_size * sizeof(unsigned int));
    memset(new_heads, NULL, hash_table->hash_table_size * sizeof(unsigned int));

    // Iterar sobre a hash table antiga
    for (unsigned int i = 0; i < hash_table->hash_table_size / 2; i++)
    {
        if (hash_table->heads[i] != NULL)
        {
            node = hash_table->heads[i];
            // Iterar sobre as linked lists
            while (node != NULL)
            {
                next_node = node->next;
                hashVal = crc32(node->word, hash_table->hash_table_size);

                // Colocar na nova hash table
                if (new_heads[hashVal] == NULL)
                {
                    node->next = NULL;
                    new_heads[hashVal] = node;
                }
                else
                {
                    hash_table_node_t *last_node;

                    last_node = new_heads[hashVal];
                    while (last_node->next != NULL)
                    {
                        last_node = last_node->next;
                    }
                    last_node->next = node;
                    node->next = NULL;
                }
            }
            node = next_node;
        }
    }

    // Trocar o array na variável hash_table_t pelo novo array
    hash_table_node_t **old_heads = hash_table->heads;
    hash_table->heads = new_heads;
    free(old_heads);
}
```

## COMPONENTES CONEXOS

Após a inserção de todas as palavras e correspondentes *nodes* na *hash table*, podemos começar a criar os grafos que, como mencionados previamente, são o conjunto de ligações que conectam os *nodes* entre si.

Estes nós são os mesmos *nodes* da *hash table*, mas conectados de forma diferente, sempre sem afetar a *hash table*.

Para isto é preciso iterar sobre os *nodes* da própria *hash table* e aplicar-lhes a função `similar_words()`.

```
printf("\n Connecting all the nodes...\n");
// Iterar sobre todos os nodes
for(i = 0; i < hash_table->hash_table_size; i++)
{
    for(node = hash_table->heads[i]; node != NULL; node = node->next)
    {
        similar_words(hash_table, node);
    }
}
```

A função `similar_words()` simplesmente cria todas as variantes possíveis de cada palavra inserida ao modificar apenas uma letra. Por exemplo, a palavra "AAA" seria modificada para:

"BAA", "CAA", ...,  
"ABA", "ACA", ...,  
"AAB", "AAC", ...,  
"ZZZ".

```
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,
        0x4E, 0x4F, 0x50, 0x51, 0x52, 0x53, 0x54, 0x55, 0x56, 0x57, 0x58, 0x59, 0x5A,
        0x61, 0x62, 0x63, 0x64, 0x65, 0x66, 0x67, 0x68, 0x69, 0x6A, 0x6B, 0x6C, 0x6D,
        0x6E, 0x6F, 0x70, 0x71, 0x72, 0x73, 0x74, 0x75, 0x76, 0x77, 0x78, 0x79, 0x7A,
        0xC1, 0xC2, 0xC9, 0xCD, 0xDA,
        0xE0, 0xE1, 0xE2, 0xE3, 0xE7, 0xE8, 0xE9, 0xEA, 0xED, 0xEE, 0xEF, 0xF4, 0xF5, 0xFA, 0xFC,
        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;
    }
    if (from->head == NULL) {
        return;
    }
    for(adjacency_node_t *link = from->head; link->next != NULL; link = link->next)
        link->vertex->visited = 0;
}
```

O algoritmo passa ambas as palavras (gerada e original) pela função `add_edge()`, que verifica se a palavra gerada existe na *hash table*. Caso exista, adiciona uma ligação (*adjacency node*) entre as duas palavras.

A função `add_edge()` é mais complicada.

Inicialmente apenas procura, com a função `find_word()` (sem inserção), se a palavra gerada existe, o que na maior parte dos casos não acontece.

```
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;
    to = find_word(&hash_table, word, 0);

    if (to == NULL) {
        return;
    }
}
```

Se a palavra gerada existir, a função `find_word()` retorna o ponteiro para o *node* dessa palavra, que será o que vamos usar para as próximas operações.

É preciso notar que cada *node*, quando criado, é, por *default*, um componente conexo com apenas ele próprio, do qual ele é o representante e não existe nenhuma ligação para nenhum outro *node* em termos de componentes conexos (mas este ainda está paralelamente na *hash table*).

De seguida, são comparados os tamanhos dos componentes conexos em que as duas palavras estão inseridas, afim de decidir qual o componente conexo que prevalece na junção dos dois.



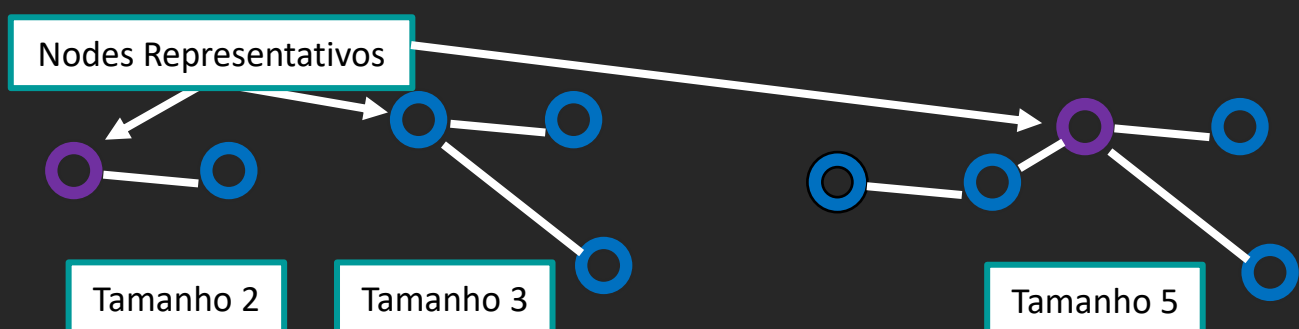
Para melhor eficiência, o que prevalece é o maior, sendo que o mais pequeno é "absorvido" por este (mudando o seu representante para o representante do componente maior). Se os componentes tiverem o mesmo número de *nodes*, o que prevalece é o da palavra original (por convenção).

```
from->number_of_edges++;
to->number_of_edges++;

// Encontrar o representante de cada node
from_representative = find_representative(from);
to_representative = find_representative(to);
totalEdges++;

if (from_representative != to_representative) {
    // Comparar os componentes conexos de cada node para decidir qual
    // componente conexo prevalece na sua junção
    if (from_representative->number_of_vertices > to_representative->number_of_vertices) {
        from_representative->number_of_vertices += to_representative->number_of_vertices;
        to_representative->representative = from_representative;
    }
    // Se forem menores ou iguais, vai para o node menor ou o que tiver mais valor
    // em strcmp(sempre o to)
    else if (from_representative->number_of_vertices < to_representative->number_of_vertices) {
        to_representative->number_of_vertices += from_representative->number_of_vertices;
        from_representative->representative = to_representative;
    }
    // Automaticamente atribui a prioridade à palavra com menor valor no strcmp,
    // visto que só essas são usadas nesta função
    else {
        from_representative->number_of_vertices += to_representative->number_of_vertices;
        to_representative->representative = from_representative;
    }
}
```

O representativo do componente conexo começa a apontar para o novo representativo e o novo representativo atualiza o número de Nodes no novo componente conexo. É de notar que apenas o Node representativo tem a informação correta sobre o componente conexo que representa.



```

// Adicionar link ao node from
adjacency_node_t *new_link0 = allocate_adjacency_node();
new_link0->vertex = to;
new_link0->next = NULL;
link = from->head;
to->visited = 1;
// Colocar na lista de links do node
if(link == NULL) {
    from->head = new_link0;
}
else {
    while(link->next != NULL) {
        link = link->next;
    }
    link->next = new_link0;
}

// Adicionar link ao node to
adjacency_node_t *new_link1 = allocate_adjacency_node();
new_link1->vertex = from;
new_link1->next = NULL;
link = to->head;
// Colocar na lista de links do node
if(link == NULL) {
    to->head = new_link1;
}
else {
    while(link->next != NULL) {
        link = link->next;
    }
    link->next = new_link1;
}

return;
}

```

Após juntar os componentes conexos é preciso criar dois *links* (*adjacency nodes*) para cada um de forma a juntar o *node* original (*from*) ao da palavra gerada (*to*).

Algo que ainda não foi mencionado é como procurar o *representative node* de cada componente conexo.

Todos os *nodes* têm um ponteiro para o que acham ser o seu representativo.

Este caminho vai sendo seguido até ser encontrado um *node* que aponte para si mesmo.

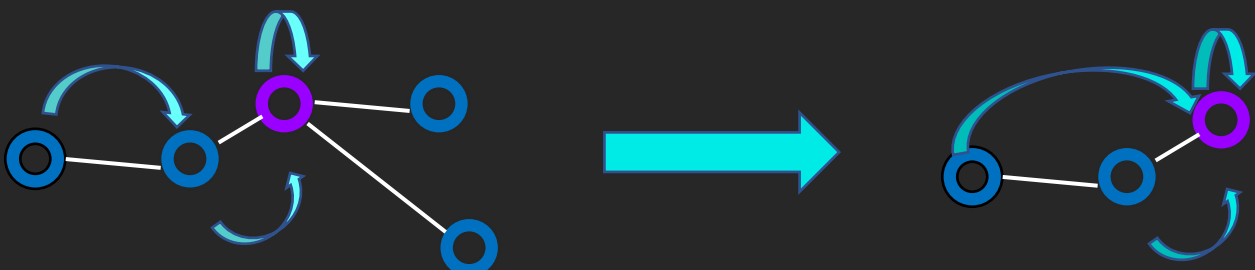
Esse sim, é o real *node* representativo do componente conexo. A função `find_representative()` é a responsável por encontrar o *representative node* de cada componente conexo.

```

static hash_table_node_t *find_representative(hash_table_node_t *node)
{
    hash_table_node_t *representative, *next_node;
    representative = node;
    // Mover para o node representativo até um deles apontar para si próprio
    while (representative->representative != representative) {
        representative = representative->representative;
    }
    // Optimizar o node, apontando diretamente para o node representativo real
    node->representative = representative;
    return representative;
}

```

Após encontrarmos o nó representativo de cada componente conexo podemos otimizar o *node* de forma a que este aponte diretamente para o nó representativo do componente.



# IMPLEMENTAÇÃO DAS OPÇÕES

Ao executar o programa principal, é impresso um menu no terminal com um total de 7 opções para o utilizador escolher (foram acrescentadas mais 4 opções em relação ao código original), sendo estas:

- 1 → **WORD**
- 2 → **FROM TO**
- 3 → **DISPLAY HASH TABLE**
- 4 → **DISPLAY HASH TABLE INFO**
- 5 → **DISPLAY GRAPH**
- 6 → **DISPLAY GRAPH INFO**
- 7 → **TERMINATE**



# WORD

Esta opção lê a palavra inserida pelo utilizador e lista o componente conexo a que pertence. Invoca a função `list_connected_component()`, que recursivamente percorre todos os caminhos de *links* de cada *node* (sem repetição).

```
//static void list_connected_component(hash_table_t *hash_table, const char *word, int numSpaces)
{
    hash_table_node_t *node = find_word(&hash_table, word, 0);
    // Caso a palavra não exista
    if (node == NULL) {
        printf("                ERRO!!!\n", word);
        printf("                Essa palavra não existe\n", word);
        printf("                no ficheiro seleccionado!\n", word);
        return;
    }
    // Caso tenha-mos chegado ao fim da lista de links da palavra
    if (node->head == NULL) {
        return;
    }

    // Marcar o node atual como visitado
    node->visited = 1;
    // Iterar sobre todas os nodes ligados ao original
    for(adjacency_node_t *link = node->head; link != NULL; link = link->next) {
        // Não listar nodes visitados
        if (link->vertex->visited == 1) {
            continue;
        }
        // Imprimir o node
        printf("                Nível %4i | %14s\n", numSpaces, link->vertex->word);
        // Recursivamente ler o próximo node e os seus links
        list_connected_component(hash_table, link->vertex->word, numSpaces+1);
    }
}
```

Teste com a palavra "Juan":

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	DISPLAY HASH TABLE	display the hash table and appropriate info
4	DISPLAY HASH TABLE INFO	display appropriate info about the hash table
5	DISPLAY GRAPH	display the graph
6	DISPLAY GRAPH INFO	display appropriate info about the graph
7		terminate

→ 1 Juan

→ Juan

Nível	0	yuan
Nível	0	Joan
Nível	1	John
Nível	1	Jean
Nível	2	jean

# FROM TO

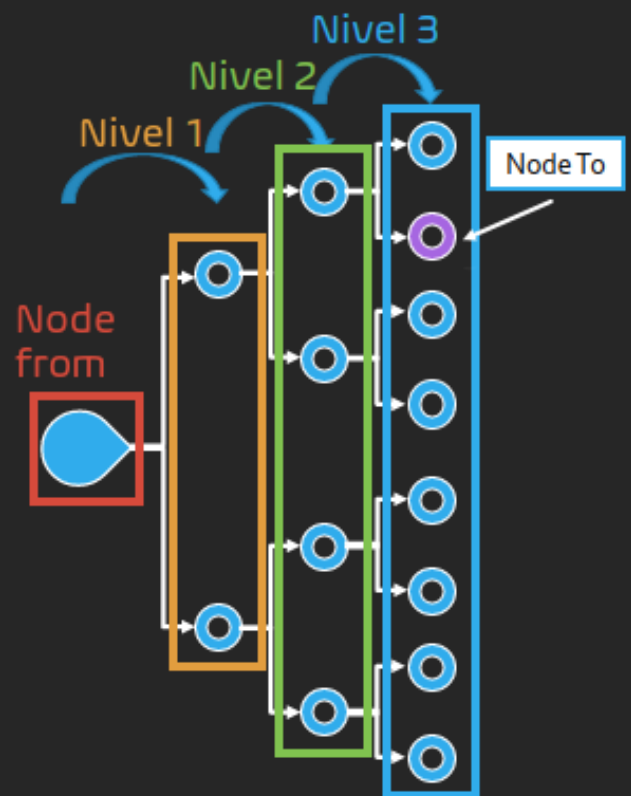
Imprime o (menor) caminho entre duas palavras dadas - a origem (*from*) e destino (*to*) - verificando previamente que ambas AS PALAVRAS existem no ficheiro a ser lido e que ambas pertencem ao mesmo componente conexo (caso eles existam no mesmo componente conexo).

Este caminho é depois encontrado seguindo uma pesquisa do tipo *breadth-first search*, sendo que o caminho encontrado é sempre o menor possível.

Em suma o *breadth-first search* é um algoritmo de pesquisa que procura a informação por "níveis" da estrutura de dados proposta.

Nesta implementação, o algoritmo começa por listar os *nodes* de primeiro "nível", ou seja, que estão diretamente ligados à palavra origem (*from*).

Após verificar que nenhum é a palavra destino (*to*), o algoritmo prossegue por todas as listas dos *nodes* já listados, seguindo cada lista, uma por uma.



```
static void list_connected_component(hash_table_t *hash_table, const char *word, int numSpaces)
{
    hash_table_node_t *node = find_word(&hash_table, word, 0);
    // Caso a palavra não exista
    if (node == NULL) {
        printf("                ERRO!!!                \n");
        printf("                Essa palavra não existe    \n");
        printf("                no ficheiro selecionado!    \n");
        return;
    }
    // Caso tenha-mos chegado ao fim da lista de links da palavra
    if (node->head == NULL) {
        return;
    }

    // Marcar o node atual como visitado
    node->visited = 1;
    // Iterar sobre todas os nodes ligados ao original
    for(adjacency_node_t *link = node->head; link != NULL; link = link->next) {
        // Não listar nodes visitados
        if (link->vertex->visited == 1) {
            continue;
        }
        // Imprimir o node
        printf("                Nível %4i | %14s | \n", numSpaces, link->vertex->word);
        // Recursivamente ler o próximo node e os seus links
        list_connected_component(hash_table, link->vertex->word, numSpaces+1);
    }
}
```



```

static int breadth_first_search(int maximum_number_of_vertices, hash_table_node_t *list_of_vertices)
{
    int n = 0; // end of the node "level"
    int i = 0; // tail of the list
    int found = 0;

    origin->visited = 1;
    list_of_vertices[0] = origin;

    while (found == 0 && n < maximum_number_of_vertices) {
        // Iterate through every adjacency node connected to the current node
        for(adjacency_node_t *link = list_of_vertices[n]->head; link != NULL; link = link->next) {
            // Dont check the path that is already traveled by
            if( link->vertex->visited == 1 ){
                continue;
            }

            i++;
            link->vertex->visited = 1;
            // Link the new node to its respective "parent" (current node)
            link->vertex->previous = list_of_vertices[n];
            // Add the new node to the tail of the list
            list_of_vertices[i] = link->vertex;

            // If we find the pretended node
            if (link->vertex == goal) {
                found = 1;
                return n;
            }
        }
        n++;
    }
    return -1;
}

```

Teste com as palavras “*tudo*” como origem e “*nada*” como destino:



# DISPLAY HASH TABLE

Imprime a *hash table* completa, com os *hash values* à esquerda e as respectivas *linked lists* contendo as palavras.

```
else if(command == 3) {
    printf("          ");
    // Iterar sobre a tabela toda
    for (int x = 0u; x < hash_table->hash_table_size; x++) {
        printf(" | [%7i] |", x);
        // Iterar sobre as listas
        for(node = hash_table->heads[x]; node != NULL; node = node->next) {
            printf(" -> %s", node->word);
        }
        printf("\n");
    }
    printf("          ");
}
```

Teste com o ficheiro "wordlist\_four\_letter.txt":



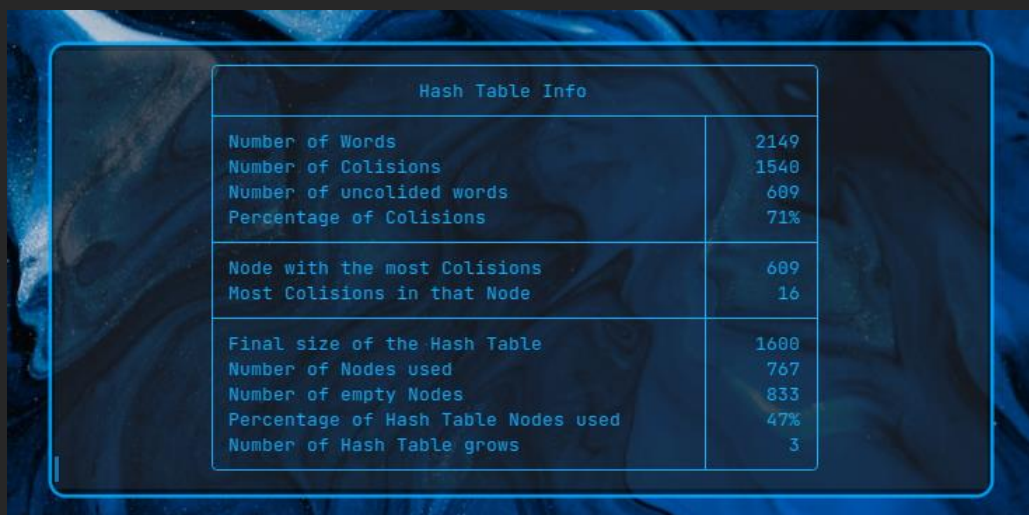
```
[ 668] → Temé
[ 669]
[ 670]
[ 671] → alio → alva → area → arfe → auto → aval → aves
[ 672]
[ 673]
[ 674] → caca → caou → cora → cose
[ 675]
[ 676] → doou → dose → faça
[ 677]
[ 678] → faca → fade → fora
[ 679] → gora → guie
[ 680] → hora
[ 681]
[ 682] → jade → jazi
[ 683]
[ 684] → laca → lesa → lhos → luto
[ 685] → maca → mesa → mete → mija → mora
[ 686] → nade → nazi → nora
[ 687] → oval → oves
[ 688] → fuça → pesa → pose → puto
[ 689]
[ 690] → grés → rija
[ 691] → saca → sete → soou
[ 692] → tesa → touu → tora
[ 693] → uivo → uval
[ 694] → vaca → vete → vivo → voou
[ 695]
[ 696]
[ 697] → viço → yuan
[ 698] → zoou
[ 699]
[ 700]
[ 701]
[ 702]
[ 703]
[ 704]
[ 705]
[ 706] → lerá → sóis
[ 707]
[ 708]
[ 709]
[ 710]
[ 711]
[ 712]
[ 713]
[ 714]
[ 715]
[ 716]
[ 717]
[ 718]
[ 719]
[ 720] → Pera

[ 1582] → nana
[ 1583] → ocos → olha → onze → ousa → ovel
[ 1584] → pana → pies → poda
[ 1585] → iman
[ 1586] → alço → rali → roda → rufo → ruir
[ 1587] → sana → soda
[ 1588] → tec1 → tens → tive → toda → topo → tufo
[ 1589] → caço → uive → unis
[ 1590] → vali → vazo → vens → vive
[ 1591]
[ 1592]
[ 1593]
[ 1594] → zebu
[ 1595] → coço
[ 1596]
[ 1597]
[ 1598]
[ 1599]
```

# DISPLAY HASH TABLE INFO

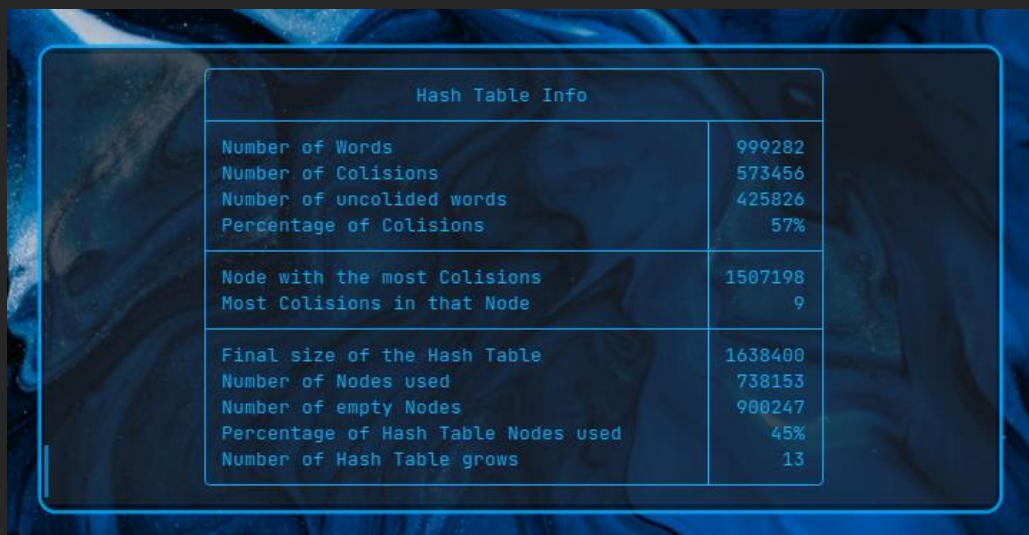
Esta opção imprime no ecrã várias informações sobre a *hash table* e os seus *nodes*. Estas informações foram calculadas ao longo da inicialização do código e da criação das estruturas principais, logo, são simplesmente impressas no terminal.

Teste com o ficheiro "wordlist\_four\_letter.txt":



Hash Table Info	
Number of Words	2149
Number of Colisions	1540
Number of uncolided words	609
Percentage of Colisions	71%
Node with the most Colisions	609
Most Colisions in that Node	16
Final size of the Hash Table	1600
Number of Nodes used	767
Number of empty Nodes	833
Percentage of Hash Table Nodes used	47%
Number of Hash Table grows	3

Teste com o ficheiro "wordlist\_big\_latest.txt":



Hash Table Info	
Number of Words	999282
Number of Colisions	573456
Number of uncolided words	425826
Percentage of Colisions	57%
Node with the most Colisions	1507198
Most Colisions in that Node	9
Final size of the Hash Table	1638400
Number of Nodes used	738153
Number of empty Nodes	900247
Percentage of Hash Table Nodes used	45%
Number of Hash Table grows	13

É de notar que:

- > A percentagem de *nodes* usados nunca é maior que 50%, pois a *hash table* é aumentada quando isso acontece;
- > Como existem menos colisões para o ficheiro com mais palavras, podemos concluir que o tamanho da *hash table* afeta bastante o número de colisões, mesmo quando inserimos um número elevado de *nodes*.

# DISPLAY GRAPH

A opção imprime no ecrã todos os componentes conexos.

Após a seleção da opção, pode também ser introduzido o tamanho mínimo do componente conexo a mostrar visto que muito deles têm tamanho 1 (sem ligações). A opção imprime também o número de grafos relevantes que encontrou.

```
else if(command == 5) {
    int minConnCompSize;
    int numConnCompShown = 0;
    char* tempStr[100];
    fprintf(stderr, "          Tamanho mínimo do grafo -> ");
    scanf("%99s", tempStr) != 1;
    minConnCompSize = atoi(tempStr);
    // Print the nodes of all the representatives
    for (int x = 0; x <= hash_table->hash_table_size; x++) {
        if(hash_table->heads[x] == NULL) {
            continue;
        }
        for(node = hash_table->heads[x]; node != NULL; node = node->next) {
            hash_table_node_t* representative = find_representative(node);
            if (representative->visited == 0 && representative->number_of_vertices >= minConnCompSize) {
                printf("\n
                Representative: | \n");
                printf("-> %-18s | \n", representative->word);
                printf("
                | \n");
                list_connected_component(hash_table, representative->word, 0);
                printf("
                | \n");
                numConnCompShown++;
                representative->visited = 1;
            }
        }
    }

    printf("\n
    Número de componentes | \n");
    printf("conexos com tamanho igual | \n");
    printf("ou maior do que %-4i | \n", minConnCompSize);
    printf("
    | \n");
    printf("-> %9i | \n", numConnCompShown);
    printf("
    | \n");
}
```

A função começa por iterar por cada palavra da *hash table* e listar o componente conexo de cada uma a partir do seu representante.

Para não repetir componentes conexos, o representante da tabela é marcado como visitado e será ignorado se for encontrado outra vez.

A função contém um marcador que aumenta com o número de componentes conexos apresentados (dependente do ficheiro utilizado e do tamanho mínimo de grafos inserido).



Teste com o ficheiro "wordlist\_four\_letter.txt":

Nível	168	robô
Nível	155	poli
Nível	122	vêus
Nível	100	Elsa

Representative:  
→ Leal

Nível	0	Real
Nível	1	leal
Nível	2	real

Representative:  
→ Juan

Nível	0	yuan
Nível	0	Joan
Nível	1	John
Nível	1	Jean
Nível	2	jean

Número de componentes  
conexos com tamanho igual  
ou maior do que 4

→ 3

Teste com o ficheiro "wordlist\_big\_latest.txt":

Nível	2	enceleirais
-------	---	-------------

Representative:  
→ estagiais

Nível	0	estagnais
Nível	1	estagneis
Nível	2	estagieis

Representative:  
→ segredar-nos

Nível	0	segregar-nos
Nível	1	segregar-vos
Nível	2	segredar-vos

Representative:  
→ prendais

Nível	0	prensais
Nível	1	preNSEis
Nível	2	prendeis

Número de componentes  
conexos com tamanho igual  
ou maior do que 4

→ 45743



# DISPLAY GRAPH INFO

Esta opção imprime informações gerais sobre o conjunto final dos componentes conexos criados, mostrando o número de conexões, vértices e o número total de componentes conexos.

Também são mostrados quantos componentes conexos não têm ligações, ou seja, quantas palavras não tem nenhuma ligação com outras e o número de vértices no maior componente conexo.

Estas informações foram calculadas ao longo da inicialização do código e da criação das estruturas principais, pelo que são simplesmente impressas no terminal.

Teste com o ficheiro "wordlist\_four\_letter.txt":

A terminal window with a dark background and light blue text. It displays a table titled "Graph Info" with five rows of statistics.

Graph Info	
Number of Edges	9267
Number of Vertices	2149
Number of connected components (graphs)	187
Number of Words with no connections	164
Size of the largest connected component	1931

Teste com o ficheiro "wordlist\_big\_latest.txt":

A terminal window with a dark background and light blue text. It displays a table titled "Graph Info" with five rows of statistics.

Graph Info	
Number of Edges	1060534
Number of Vertices	999282
Number of connected components (graphs)	377234
Number of Words with no connections	184869
Size of the largest connected component	16698

O facto de o ficheiro menor ter uma percentagem maior de componentes conexos sem ligações (tamanho 1) deve-se apenas ao facto de as palavras deste estarem quase todas acumuladas num só componente conexo devido à sua semelhança.

# TERMINATE

A opção final **TERMINATE** é uma das mais importantes, visto que esta tem de limpar completamente toda a memória alocada ao longo da execução do código para que não causemos problemas à máquina do utilizador, nomeadamente através de *Memory Leaks* (espaços de memória alocados sem qualquer forma de os alcançar). Isto também faz com que programas maliciosos não consigam ir buscar variáveis deixadas para trás em memória, após a execução o código.

Para isto implementamos a função `free_hash_table()`.

Dado que todos os alocações de memória estão associados à estrutura da *hash table*, e como esta é uma estruturação com um elevado grau de abstração, é necessário percorrer por todos os valores alocados da *hash table* e limpá-los.

Portanto a função `hash_table_free()` :

- 1 → Começa por iterar pelos valores na *hash table*, caso esta entrada não seja nula, começa a percorrer a lista ligada associada ao *hash code* da iteração atual;
- 2 → Para cada entrada não nula, é iterada a lista desta entrada num *while loop*, que termina sua iteração se encontrar o fim (NULL) da lista ligada, e chama o método `free_hash_table_node()` para liberar o node;
- 3 → Por fim, é chamada a função de libertação de memória alocada nativa da linguagem C – `free()` – para libertar os espaços do array da *struct* da tabela, `hash_table->heads`.

```
static void hash_table_free(hash_table_t *hash_table)
{
    hash_table_node_t *crawler;
    hash_table_node_t *node_before;
    // Percorrer a hash table toda
    for (unsigned int i = 0; i < hash_table->hash_table_size; i++) {
        if (hash_table->heads[i] == NULL) {
            continue;
        }
        // Percorrer pelas listas até ao final
        crawler = hash_table->heads[i];
        while (crawler->next != NULL) {
            node_before = crawler;
            crawler = crawler->next;
            // Libertar o node anterior ao crawler
            free_hash_table_node(node_before);
        }
        // Libertar o node deixado para trás pelo crawler
        free_hash_table_node(crawler);
    }
    // Libertar o resto da hash table
    free(hash_table->heads);
    free(hash_table);

    return;
}
```

Similarmente ao `free_hash_table()`, o `free_hash_table_node()` itera sobre todos os *links* alocados de cada *node* e liberta-os na memória.

```
static void free_hash_table_node(hash_table_node_t *node)
{
    adjacency_node_t *adj_crawler = node->head;
    adjacency_node_t *adj_before;
    while(adj_crawler != NULL)
    {
        adj_before = adj_crawler;
        adj_crawler = adj_crawler->next;
        free_adjacency_node(adj_before);
    }
    free_adjacency_node(adj_crawler);

    free(node);
}
```

Após correr o programa com a ferramenta “*valgrind*” (um programa que deteta o uso de memória de um processo), podemos verificar que, de facto, a nossa implementação não sofre de quaisquer *ES memory leaks*, e que o nosso número de alocações é igual ao número de libertações:

```

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 DISPLAY HASH TABLE                 display the hash table and appropriate info
4 DISPLAY HASH TABLE INFO            display appropriate info about the hash table
5 DISPLAY GRAPH                       display the graph
6 DISPLAY GRAPH INFO                  display appropriate info about the graph
7                                     terminate

→ 7

==187984==
==187984== HEAP SUMMARY:
==187984==    in use at exit: 0 bytes in 0 blocks
==187984==   total heap usage: 3,120,369 allocs, 3,120,369 frees, 140,099,088 bytes allocated
==187984==
==187984== All heap blocks were freed -- no leaks are possible
==187984==
==187984== For lists of detected and suppressed errors, rerun with: -s
```

# FUNÇÕES SECUNDÁRIAS

Implementamos também algumas funções que não são vitais para o objetivo final do projeto, mas facilitam a implementação de outras funcionalidades do projeto ou alteram a estética do programa em si.

## setnNodesVisitedTo0

Coloca a *flag* "visited" de todos os *nodes* de volta a 0.

É essencial correr esta função sempre que se acaba de utilizar alguma outra que altere esta *flag*.

```
void setNodesVisitedTo0(hash_table_t *hash_table) {
    hash_table_node_t *node;
    // Set all the node's "visited" flag back to 0
    for (int x = 0u; x <= hash_table->hash_table_size; x++) {
        if(hash_table->heads[x] == NULL) {
            continue;
        }
        for(node = hash_table->heads[x]; node != NULL; node = node->next) {
            node->visited = 0;
        }
    }
}
```

# progressBar

Imprime no terminal uma barra de progresso que informa visualmente o utilizador do progresso na inicialização do algoritmo.

Teste com o ficheiro "wordlist\_four\_letter.txt":



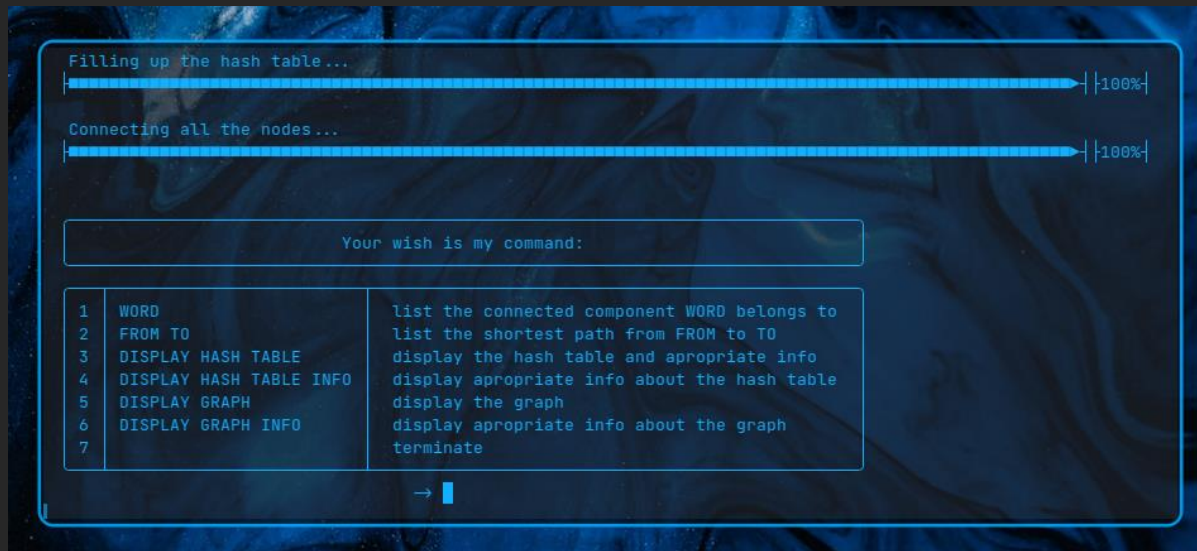
```
//  
void progressBar(int percent) {  
    printf("\r |");  
    for (int x = 0; x < percent-1; x++) {  
        printf("■");  
    }  
    printf(" ▶");  
    for (int x = percent; x < 100; x++) {  
        printf("-");  
    }  
    printf("| |%3i%| ", percent);  
    fflush(stdout);  
}
```

```
percent = (int) (hash_table->number_of_entries * 100 / (hash_table->hash_table_size / 2));  
progressBar(percent);
```



# RESULTADOS

Ao correr o programa, obtemos o seguinte resultado:



A partir deste menu podemos aceder e visualizar qualquer detalhe que esteja relacionado com a *hash table* ou com o grafo criado, utilizando as funções previamente exploradas.

Todas as funções implementadas funcionam como esperado, como pudemos ver anteriormente nos testes relativos a cada opção.

## ALOCAÇÕES E MEMORY LEAKS

Como também mencionado anteriormente, o nosso programa não sofre de *memory leaks*, alcançando assim um dos objetivos mais importantes deste projeto, sendo o número de alocações igual ao número de libertações.



# CONCLUSÃO

Concluindo este projeto, podemos afirmar que a nossa percepção de estruturas e interações de variáveis avançou enormemente, assim como a nossa capacidade de escrever códigos eficientes e otimizados a nível de recursos (visto que a nossa implementação gasta apenas cerca de 140Mb para o maior ficheiro de texto, "*wordlist\_big\_latest*").

Este projeto envolveu também a criação de algoritmos de manipulação e de arquivo de dados, e pelo que pudemos apurar, será algo que nos será útil em projetos futuros desta dimensão.

O grupo reconhece que o nosso algoritmo, por mais que já se encontre bastante eficiente e otimizado, carecerá sempre de algumas adaptações que o tornem, ainda, mais eficiente e dinâmico.

Futuramente, com mais experiência em desenvolvimento de algoritmos, poderá ser possível aperfeiçoar este código de modo a ser ainda mais rápido e/ou desenvolver uma outra solução baseada em novos conceitos e ideias.



# WEBGRAFIA

- > <https://askubuntu.com/>
- > <https://aur.archlinux.org/>
- > <https://doc.ic.ac.uk/>
- > <https://geeksforgeeks.org>
- > <https://groups.google.com/g>
- > <https://log2base2.com>
- > <https://link.springer.com>
- > <https://programiz.com/>
- > <https://stackoverflow.com>
- > <https://users.cs.cf.ac.uk/>
- > <https://web.mit.edu/>

# APÊNDICE

## CÓDIGO EM C

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

//
// static configuration
//

#define _max_word_size_ 32

int totalWords = 0;
int totalColisions = 0;
int mostColHashNode = 0;
int mostColisions = 0;
int totalGrows = 0;
int numUsedHashNodes = 0;

int totalEdges = 0;
int numConnectedComponents = 0;
int numSeperatedComponents = 0;
int largestComponent = 0;

//
// 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)
};

struct hash_table_s
{
    unsigned int hash_table_size;   // the size of the hash table array
    unsigned int number_of_entries; // the number of entries in the hash table
    unsigned int number_of_edges;   // number of edges (for information purposes only)
    hash_table_node_t **heads;      // the heads of the linked lists
};

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

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

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

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

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

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

```



```

}

static void free_hash_table_node(hash_table_node_t *node)
{
    adjacency_node_t *adj_crawler = node->head;
    adjacency_node_t *adj_before;
    while(adj_crawler != NULL)
    {
        adj_before = adj_crawler;
        adj_crawler = adj_crawler->next;
        free_adjacency_node(adj_before);
    }
    free_adjacency_node(adj_crawler);

    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;

    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);
    }

    hash_table->hash_table_size = 200;
    hash_table->number_of_entries = 0;

```

```

hash_table->number_of_edges = 0;

hash_table->heads = (hash_table_node_t**) malloc(hash_table->hash_table_size*sizeof(hash_table_node_t));
memset(hash_table->heads, 0, hash_table->hash_table_size*sizeof(hash_table_node_t));

return hash_table;
}

static hash_table_node_t *find_word(hash_table_t **hash_table, const char *word, int insert_if_not_found);
static void hash_table_free(hash_table_t *hash_table);

static void hash_table_grow(hash_table_t *hash_table)
{
    unsigned int hashVal;
    hash_table_node_t *node;
    hash_table_node_t *next_node;

    hash_table->hash_table_size = hash_table->hash_table_size * 2;
    totalGrows++;
    // Alocar o novo array de hash table
    hash_table_node_t **new_heads = (hash_table_node_t**) malloc(hash_table->hash_table_size*sizeof(hash_table_node_t));
    memset(new_heads, 0, hash_table->hash_table_size*sizeof(hash_table_node_t));

    // Iterar sobre a hash table antiga
    for (unsigned int i = 0; i < hash_table->hash_table_size/2; i++) {
        if (hash_table->heads[i] != NULL) {
            node = hash_table->heads[i];
            // Iterar sobre as linked lists
            while(node != NULL) {
                next_node = node->next;
                hashVal = crc32(node->word) % hash_table->hash_table_size;

                // Colocar na nova hash table
                if (new_heads[hashVal] == NULL) {
                    node->next = NULL;
                    new_heads[hashVal] = node;
                }
                else {
                    hash_table_node_t *last_node;

                    last_node = new_heads[hashVal];
                    while (last_node->next != NULL) {
                        last_node = last_node->next;
                    }
                    last_node->next = node;
                    node->next = NULL;
                }
            }

            node = next_node;
        }
    }
}

```

```

}
// Trocar o array na variável hash_table_t pelo novo array
hash_table_node_t **old_heads = hash_table->heads;
hash_table->heads = new_heads;
free(old_heads);
}

static void hash_table_free(hash_table_t *hash_table)
{
    hash_table_node_t *crawler;
    hash_table_node_t *node_before;
    // Percorrer a hash table toda
    for (unsigned int i = 0u; i < hash_table->hash_table_size; i++) {
        if (hash_table->heads[i] == NULL) {
            continue;
        }
        // Percorrer pelas listas até ao final
        crawler = hash_table->heads[i];
        while (crawler->next != NULL) {
            node_before = crawler;
            crawler = crawler->next;
            // Libertar o node anterior ao crawler
            free_hash_table_node(node_before);
        }
        // Libertar o node deixado para trás pelo crawler
        free_hash_table_node(crawler);
    }
    // Libertar o resto da hash table
    free(hash_table->heads);
    free(hash_table);

    return;
}

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

    // Verificar o preenchimento da hash table
    if ((*hash_table)->number_of_entries >= (*hash_table)->hash_table_size / 2) {
        hash_table_grow(*hash_table);
    }
    // Obter o hash code da palavra
    hashVal = crc32(word) % (*hash_table)->hash_table_size;

    // Se a operação for de insert
    if (insert_if_not_found == 1) {
        // Criar o Node
        hash_table_node_t *node = allocate_hash_table_node();
        node->next = NULL;
        node->head = NULL;
        node->visited = 0;
        node->previous = NULL;
    }

```

```

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

    // Se não existir um Node nesse hash value
    if ((*hash_table)->heads[hashVal] == NULL) {
        strcpy(node->word, word);

        (*hash_table)->heads[hashVal] = node;
        (*hash_table)->number_of_entries++;
    }
    // Se já existir um Node nesse hash value
    else {
        totalColisions++;
        hash_table_node_t *last_node;
        // Percorrer a lista até ao fim
        last_node = (*hash_table)->heads[hashVal];
        while (last_node->next != NULL)
        {
            last_node = last_node->next;
        }
        last_node->next = node;
        strcpy(node->word, word);
    }
    return NULL;
}

// Caso não seja para inserir nodes novos
else {
    hash_table_node_t *node;
    // Caso o hash value corresponda a algum node
    if ((*hash_table)->heads[hashVal] != NULL) {
        node = (*hash_table)->heads[hashVal];
        // Percorrer a lista de nodes
        // até encontrar o pretendido
        while (node != NULL) {
            if (strcmp(node->word, word)==0) {
                return node;
            }
            else {
                node = node->next;
            }
        }
    }
    return NULL;
}
}

//
// 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 *representative;
representative = node;
// Mover para o node representativo até um deles apontar para si próprio
while (representative->representative != representative) {
    representative = representative->representative;
}
// Optimizar o node, apontando diretaente para o node representativo real
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 *link;
    to = find_word(&hash_table, word, 0);

    if (to == NULL) {
        return;
    }

    from->number_of_edges++;
    to->number_of_edges++;

    // Encontrar o representativo de cada node
    from_representative = find_representative(from);
    to_representative = find_representative(to);
    totalEdges++;

    if (from_representative != to_representative) {
        // Comparar os componentes conexos de cada node para decidir qual
        // componente conexo prevalece na sua junção
        if (from_representative->number_of_vertices > to_representative->number_of_vertices) {
            from_representative->number_of_vertices += to_representative->number_of_vertices;
            to_representative->representative = from_representative;
            to->representative = from_representative;
        }
        // Se forem menores ou iguais, vai para o node menor ou o que tiver mais valor
        // em strcmp(sempre o to)
        else if (from_representative->number_of_vertices < to_representative->
number_of_vertices) {
            to_representative->number_of_vertices += from_representative->number_of_vertices;
            from_representative->representative = to_representative;
            from->representative = to_representative;
        }
        // Automaticamente atribui a prioridade à palavra com menor valor no strcmp,
        // visto que só essas são usadas nesta função
        else {
            from_representative->number_of_vertices += to_representative->number_of_vertices;
            to_representative->representative = from_representative;
            to->representative = from_representative;
        }
    }
}

```



```

// Adicionar link ao node from
adjacency_node_t *new_link0 = allocate_adjacency_node();
new_link0->vertex = to;
new_link0->next = NULL;
link = from->head;
to->visited = 1;
// Colocar na lista de links do node
if(link == NULL) {
    from->head = new_link0;
}
else {
    while(link->next != NULL) {
        link = link->next;
    }
    link->next = new_link0;
}

// Adicionar link ao node to
adjacency_node_t *new_link1 = allocate_adjacency_node();
new_link1->vertex = from;
new_link1->next = NULL;
link = to->head;
// Colocar na lista de links do node
if(link == NULL) {
    to->head = new_link1;
}
else {
    while(link->next != NULL) {
        link = link->next;
    }
    link->next = new_link1;
}

return;
}

//
// 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;
}
if (from->head == NULL) {
    return;
}
for(adjacency_node_t *link = from->head; link->next != NULL; link = link->next) {
    link->vertex->visited = 0;
}
}

//
// 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 breadth_first_search(int maximum_number_of_vertices,hash_table_node_t
*list_of_vertices[],hash_table_node_t *origin,hash_table_node_t *goal)
{
    int n = 0; // end of the node "level"
    int i = 0; // tail of the list
    int found = 0;

    origin->visited = 1;
    list_of_vertices[0] = origin;

    while (found == 0 && n < maximum_number_of_vertices) {
        // Iterate through every adjacency node connected to the current node
        for(adjacency_node_t *link = list_of_vertices[n]->head; link != NULL; link = link-
>next) {
            // Dont check the path that is already traveled by
            if( link->vertex->visited == 1 ){
                continue;
            }

```

```

    i++;
    link->vertex->visited = 1;
    // Link the new node to its respective "parent" (current node)
    link->vertex->previous = list_of_vertices[n];
    // Add the new node to the tail of the list
    list_of_vertices[i] = link->vertex;

    // If we find the pretended node
    if (link->vertex == goal) {
        found = 1;
        return n;
    }
}
n++;
}
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, int
numSpaces)
{
    hash_table_node_t *node = find_word(&hash_table, word, 0);
    // Caso a palavra não exista
    if (node == NULL) {
        printf("                |          ERRO!!!          |\n");
        printf("                |    Essa palavra não existe    |\n");
        printf("                |    no ficheiro selecionado!      |\n");
        return;
    }
    // Caso tenha-mos chegado ao fim da lista de links da palavra
    if (node->head == NULL) {
        return;
    }

    // Marcar o node atual como visitado
    node->visited = 1;
    // Iterar sobre todas os nodes ligados ao original
    for(adjacency_node_t *link = node->head; link != NULL; link = link->next) {
        // Não listar nodes visitados
        if (link->vertex->visited == 1) {
            continue;
        }
        // Imprimir o node
        printf("                | Nível %4i |   %14s   |\n", numSpaces, link->vertex->word);
        // Recursivamente ler o próximo node e os seus links
        list_connected_component(hash_table, link->vertex->word, numSpaces+1);
    }
}

```

```

//
// compute the diameter of a connected component (optional) 2 rosnes Mónica
//
//static int largest_diameter = 0;
//static hash_table_node_t **largest_diameter_example;

//static int connected_component_diameter(hash_table_t *hash_table, hash_table_node_t
*source)
//{
//  hash_table_node_t *node;
//  int diameter = 0;
//  int distance;
//  const hash_table_node_t *sourceRepresentative = find_representative(source);

//  for (unsigned int x = 0; x < hash_table->hash_table_size; x++) {
//    if(hash_table->heads[x] == NULL) {
//      continue;
//    }
//    // Iterar sobre as listas
//    for(node = hash_table->heads[x]; node != NULL; node = node->next) {
//      // Obter a distancia entre o node Source e todos os outros
//      if (sourceRepresentative == find_representative(node)) {
//        hash_table_node_t *currentPath[hash_table->hash_table_size + 1];
//        distance = breadth_first_search(hash_table->hash_table_size, currentPath, source,
node);
//        // Obter só o maior de todos
//        if(distance > diameter) {
//          diameter = distance;
//        }
//      }
//    }
//  }

//  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)
{
  hash_table_node_t *source = find_word(&hash_table, from_word, 0),
    *goal = find_word(&hash_table, to_word, 0);

  if (source == NULL) {
    printf("\nERRO! A palavra inicial não existe no ficheiro selecionado!!!\n");
    return;
  }
  if (goal == NULL) {
    printf("\nERRO! A palavra destino não existe no ficheiro selecionado!!!\n");
    return;
  }
}

```



```

hash_table_node_t *sourceRepresentative = find_representative(source),
                  *goalRepresentative = find_representative(goal);

if (sourceRepresentative != goalRepresentative) {
    printf("\nERRO! As palavras inseridas não tem ligações possíveis!!! (Componentes
Conexos diferentes)\n");
    return;
}

hash_table_node_t *currentPath[sourceRepresentative->number_of_vertices + 1];

(void)breadth_first_search(sourceRepresentative->number_of_vertices, currentPath, source,
goal);

int nivel = 0;
for (hash_table_node_t* parentNode = goal; parentNode != NULL; parentNode = parentNode-
>previous) {
    printf("\n--> %s", parentNode->word);
    nivel++;
}
printf("\n-Número de palavras percorridas > %i", nivel);

return;
}

//
// some graph information (optional)
//
static void graph_info()
{
    printf("
    printf("
    printf("
    printf("
    printf("
    printf("
numConnectedComponents);
    printf("
numSeperatedComponents);
    printf("
largestComponent);
    printf("
}

//
// some hash table information
//
static void hash_table_info(hash_table_t *hash_table)
{
    printf("
    printf("

```

```

printf("
printf("
printf("
totalColisions);
printf("
totalColisions);
printf("
(100*totalColisions/totalWords));
printf("
printf("
mostColHashNode);
printf("
mostColisions);
printf("
printf("
>hash_table_size);
printf("
numUsedHashNodes);
printf("
>hash_table_size - numUsedHashNodes);
printf("
(100*numUsedHashNodes/hash_table->hash_table_size));
printf("
printf("
}

```

```

void setNodesVisitedTo0(hash_table_t *hash_table) {
    hash_table_node_t *node;
    // Set all the node's "visited" flag back to 0
    for (unsigned int x = 0; x < hash_table->hash_table_size; x++) {
        if(hash_table->heads[x] == NULL) {
            continue;
        }
        for(node = hash_table->heads[x]; node != NULL; node = node->next) {
            node->visited = 0;
        }
    }
}

```

```

void setNodesPreviousToNULL(hash_table_t *hash_table) {
    hash_table_node_t *node;
    // Set all the node's "previous" flag back to NULL
    for (unsigned int x = 0; x < hash_table->hash_table_size; x++) {
        if(hash_table->heads[x] == NULL) {
            continue;
        }
        for(node = hash_table->heads[x]; node != NULL; node = node->next) {
            node->previous = NULL;
        }
    }
}

```

```
//
```

```

// calculate the hash table info, the Hash Node with the most colisions, number of nodes
used, etc
//
static void calculateInfo(hash_table_t *hash_table, int* mostColHashNode, int*
mostColisions, int* numUsedHashNodes, int* numConnectedComponents, int*
numSeperatedComponents, int* largestComponent)
{
    hash_table_node_t *node;

    // Run through every hash table head
    for (unsigned int x = 0; x <= hash_table->hash_table_size; x++) {
        if(hash_table->heads[x] == NULL) {
            continue;
        }

        // Increment number of used nodes
        *numUsedHashNodes = *numUsedHashNodes + 1;

        // Calculate how many nodes the head has
        int nodesInX = 0;

        for(node = hash_table->heads[x]; node != NULL; node = node->next) {
            nodesInX++;

            // Calculate the number of representatives (same as number of connected components)
            hash_table_node_t* representative = find_representative(node);
            if (representative->visited == 0) {
                *numConnectedComponents = *numConnectedComponents + 1;
                representative->visited = 1;
                // Calculate the largest component (most vertices)
                //printf("\nLargestComp > %i", representative->number_of_vertices);
                if (representative->number_of_vertices >= *largestComponent) {
                    *largestComponent = representative->number_of_vertices;
                }
                // Calculate number of connected components with only one word (unconnected)
                if (representative->number_of_vertices < 2) {
                    *numSeperatedComponents = *numSeperatedComponents + 1;
                }
            }
        }
    }

    // Compare the number of nodes to the maximum known
    if (nodesInX > *mostColisions) {
        *mostColHashNode = x;
        *mostColisions = nodesInX;
    }
}

setNodesVisitedTo0(hash_table);
//int biggestDiameter;
//for (int x = 0u; x < hash_table->hash_table_size; x++) {
//    if(hash_table->heads[x] == NULL) {
//        continue;
//    }
//    for(node = hash_table->heads[x]; node != NULL; node = node->next) {

```

```

// // Obter a distancia entre o node Source e todos os outros
// // Calcula a distância para todos os outros nodes
// biggestDiameter = connected_component_diameter(hash_table, node);
// if (biggestDiameter > largest_diameter) {
//     largest_diameter = biggestDiameter;
// }
// }
// }
//}
// Set all the nodes's "visited" flag to 0
//setNodesVisitedTo0(hash_table);
return;
}

//
// main program
//
void progressBar(int percent) {
    printf("\r |");
    for (int x = 0; x < percent-1; x++) {
        printf("■");
    }
    printf("▶");
    for (int x = percent; x < 100; x++) {
        printf("-");
    }
    printf("| |%3i%%| ", percent);
    fflush(stdout);
}

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);
    }

    int percent=0;

    printf("\n Filling up the hash table...\n");
    while(fscanf(fp, "%99s", word) == 1) {

```

```

(void)find_word(&hash_table,word,1);
totalWords++;
percent = (int) (hash_table->number_of_entries * 100 / (hash_table->hash_table_size /
2));
progressBar(percent);
}
fclose(fp);

percent = 100;
progressBar(percent);
printf("\n");
percent = 0;

printf("\n Connecting all the nodes...\n");
// Iterar sobre todos os nodes
for(i = 0u;i < hash_table->hash_table_size;i++) {
    percent = (int) (i * 100 / hash_table->hash_table_size);
    progressBar(percent);
    for(node = hash_table->heads[i];node != NULL;node = node->next) {
        similar_words(hash_table,node);
    }
}

// Set all the nodes's "visited" flag to 0
setNodesVisitedTo0(hash_table);
calculateInfo(hash_table, &mostColHashNode, &mostColisions, &numUsedHashNodes,
&numConnectedComponents, &numSeperatedComponents, &largestComponent);
setNodesVisitedTo0(hash_table);

percent = 100;
progressBar(percent);
printf("\n");

// ask what to do
for(;;)
{
    printf("\n\n");
    fprintf(stderr,"
    _____\n");
    fprintf(stderr," |                               Your wish is my
command:           | \n");
    fprintf(stderr,"
    _____\n");
    fprintf(stderr,"
    _____\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 | DISPLAY HASH TABLE                               | display the hash table and aproprate
info        | \n");
    fprintf(stderr," | 4 | DISPLAY HASH TABLE INFO | display aproprate info about the
hash table  | \n");
}

```



```

    fprintf(stderr," | 5 | DISPLAY GRAPH          | display the
graph                                     |\n");
    fprintf(stderr," | 6 | DISPLAY GRAPH INFO      | display aproppriate info about the
graph                                     |\n");
    fprintf(stderr," | 7
|                                     | terminate                                     |\n");
    fprintf(stderr,"
|                                     |                                     |\n");
    fprintf(stderr,"                                     -> ");
    if(scanf("%99s",word) != 1)
        break;
    command = atoi(word);
    //system("clear");

    if(command == 1) {
        if(scanf("%99s",word) != 1)
            break;
        printf("\n
        printf("                                     -> %-18s |\n", word);
        printf("                                     |\n");
        printf("                                     |\n");
        list_connected_component(hash_table,word, 0);
        printf("                                     |\n");
    }

    else if(command == 2) {
        if(scanf("%99s",from) != 1)
            break;
        if(scanf("%99s",to) != 1)
            break;
        path_finder(hash_table,from,to);
        setNodesPreviousToNULL(hash_table);
    }

    else if(command == 3) {
        printf("                                     ");
        // Iterar sobre a tabela toda
        for (unsigned int x = 0; x < hash_table->hash_table_size; x++) {
            printf(" | [%7i] |", x);
            // Iterar sobre as listas
            for(node = hash_table->heads[x];node != NULL;node = node->next) {
                printf(" -> %s", node->word);
            }
            printf("\n");
        }
        printf("                                     ");
    }

    else if(command == 4) {
        hash_table_info(hash_table);
    }

    else if(command == 5) {
        int minConnCompSize;

```

