|  |  |  |  |
| --- | --- | --- | --- |
|  |  | WORD LADDDER  ALGORITMOS E ESTRUTURAS  DE DADOS 2022  Prof. Tomás Oliveira e Silva  André Oliveira / 107637 33.(3)%  Duarte Cruz / 107359 33.(3)%  Rodrigo Graça / 107634 33.(3)% |  |
|  |  | Index - Free interface icons  Índice  [Introdução 3](#_Toc124258086)  [Funções 4](#_Toc124258087)  [Função *hash\_table\_create()* 4](#_Toc124258088)  [Função *hash\_table\_grow()* 6](#_Toc124258089)  [Função *hash\_table\_free()* 8](#_Toc124258090)  [Função *find\_word()* 10](#_Toc124258091)  [Função *find\_representative()* 11](#_Toc124258092)  [Função *add\_edge()* 12](#_Toc124258093)  [Função *breadh\_first\_search()* 14](#_Toc124258094)  [Função *list\_connected\_component()* 15](#_Toc124258095)  [Função *connected\_component\_diameter()* 16](#_Toc124258096)  [Função *path\_finder()* 18](#_Toc124258097)  [Função *find\_connected\_component\_representatives()* 19](#_Toc124258098)  [Função *graph\_info()* 20](#_Toc124258099)  [Função *hash\_table\_info()* 22](#_Toc124258100)  [Estatísticas da *Hash Table* 24](#_Toc124258101)  [Crescimento 24](#_Toc124258102)  [Atribuições de “lugares” 25](#_Toc124258103)  [Curiosidades 26](#_Toc124258104)  [Grafos 26](#_Toc124258105)  [Informações dos grafos 26](#_Toc124258106)  [Informações das *hash tables* 28](#_Toc124258107)  [Cadeias interessantes de palavras 28](#_Toc124258108)  [Confirmações de *memory leaks* 29](#_Toc124258109)  [Conclusão 30](#_Toc124258110)  [Apêndice 31](#_Toc124258111)  [Código c 31](#_Toc124258112)  [Código Matlab 38](#_Toc124258113)  [*Printscreens* resultados mostrados anteriormente 41](#_Toc124258114) |  |

# **Introdução**

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

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

Uma imagem com mesa

Descrição gerada automaticamente



# Funções

## Função *hash\_table\_create()*

Figura 1 | Função hash\_table\_create()

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

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

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

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

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

## Uma imagem com texto Descrição gerada automaticamenteFunção *hash\_table\_grow()*

Figura 2 | Função hash\_table\_grow()

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

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

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

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

## Uma imagem com texto Descrição gerada automaticamenteFunção *hash\_table\_free()*

Figura 3 | Função hash\_table\_free()

A função *hash\_table\_free* tem como objetivo liberar toda a memória alocada para uma *Hash Table*. Começa por declarar quatro variáveis: dois ponteiros para nós da *Hash Table* (*node* e *temp*) e dois ponteiros para nós de adjacência (*adj\_node* e *temp\_adj*).

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

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

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

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

## Uma imagem com texto Descrição gerada automaticamenteFunção *find\_word()*

Figura 4 | Função find\_word()

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

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

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

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

## Uma imagem com texto Descrição gerada automaticamenteFunção *find\_representative()*

Figura 5 | Função find\_representative()

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

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

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

## Uma imagem com texto Descrição gerada automaticamenteFunção *add\_edge()*

Figura 6 | Função add\_edge()

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

Os vértices são representados por elementos da estrutura *hash\_table\_node\_t* e as arestas são representadas por elementos da estrutura *adjacency\_node\_t*, que contêm um ponteiro para um vértice e um ponteiro para o próximo elemento na lista de adjacências do vértice.

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

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

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

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

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

## Uma imagem com texto Descrição gerada automaticamenteFunção *breadh\_first\_search()*

Figura 7 | Função breadh\_first\_search()

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

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

## Uma imagem com texto Descrição gerada automaticamenteFunção *list\_connected\_component()*

Figura 8 | Função list\_connected\_component()

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

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

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

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

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

## Text Description automatically generatedText Description automatically generatedFunção *connected\_component\_diameter()*

Figura 9 | Função connected\_component\_diameter()

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

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

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

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

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

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

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

## Uma imagem com texto Descrição gerada automaticamenteFunção *path\_finder()*

Figura 10 | Função path\_finder()

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

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

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

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

## Text Description automatically generatedFunção *find\_connected\_component\_representatives()*

Figura 11 | Função find\_connected\_component\_representative()

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

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

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

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

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

## Uma imagem com texto Descrição gerada automaticamenteFunção *graph\_info()*

Figura 12 | Função graph\_info()

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

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

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

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

## Uma imagem com texto Descrição gerada automaticamenteFunção *hash\_table\_info()­­­­*

Figura 13 | Função hash\_table\_info()

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

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

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

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

# Estatísticas da *Hash Table*

## Crescimento

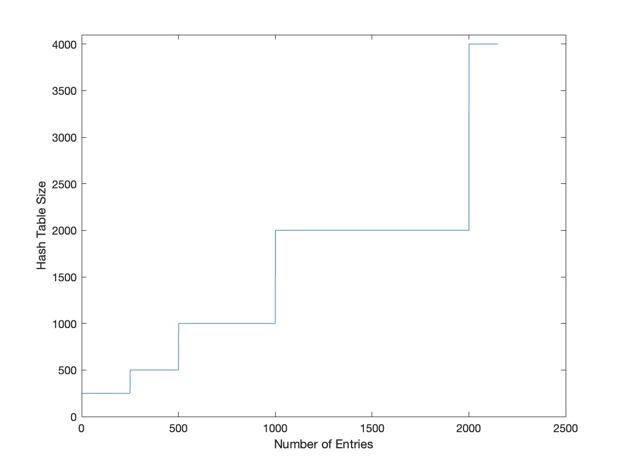
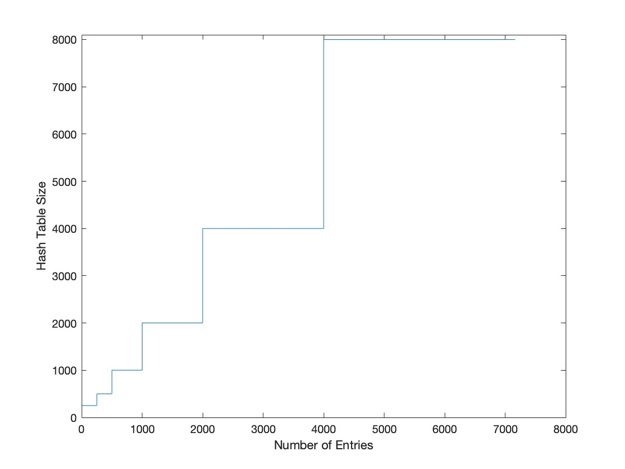


Figura 14 | Crescimento com ficheiro de 6 letras

Figura 15 | Crescimento com ficheiro de 4 letras

Figura 16 | Crescimento com ficheiro de 5 letras

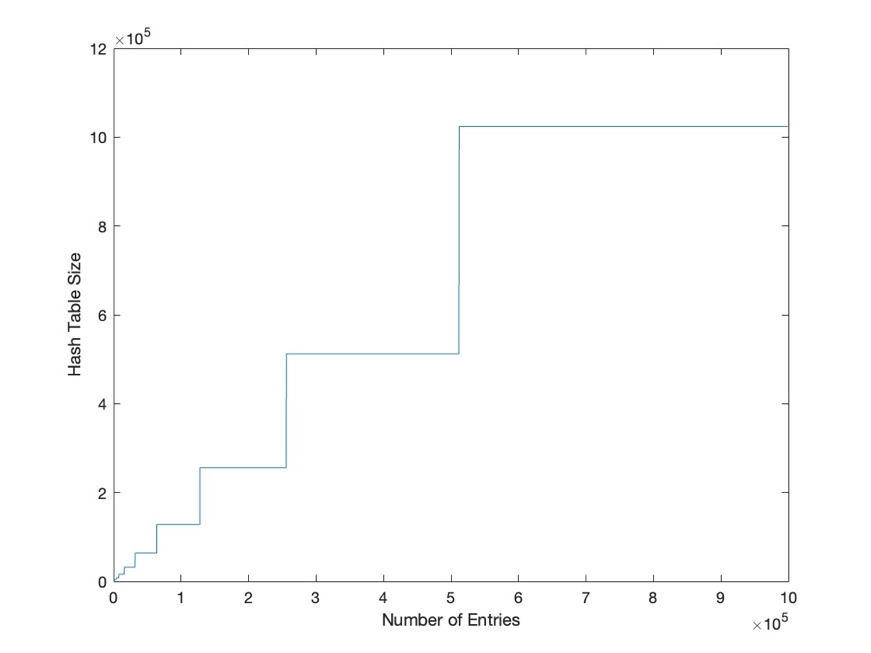


Figura 17 | Crescimento com ficheiro default

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

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

## Atribuições de “lugares”

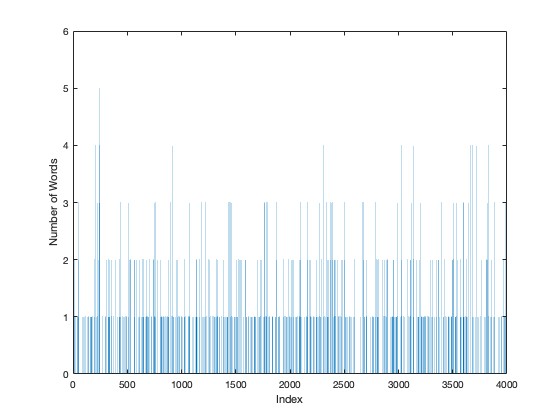
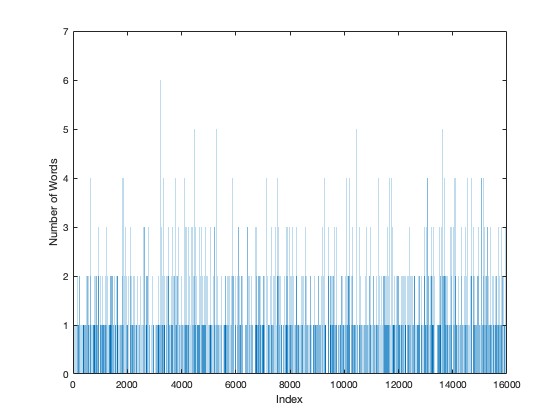
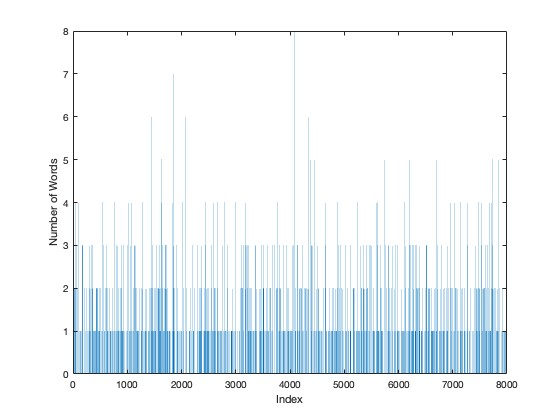


Figura 18 | Atribuição de "lugares" com ficheiro de 6 letras

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

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

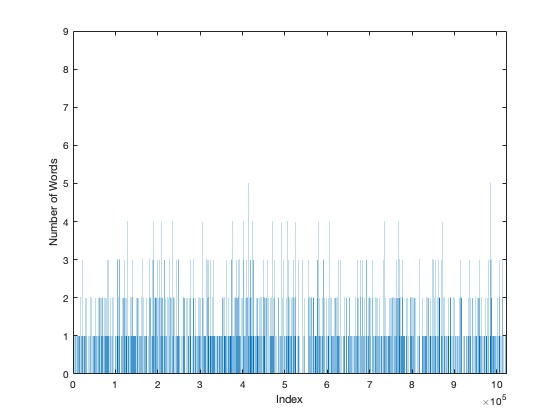


Figura 21 | Atribuição de "lugares" com ficheiro default

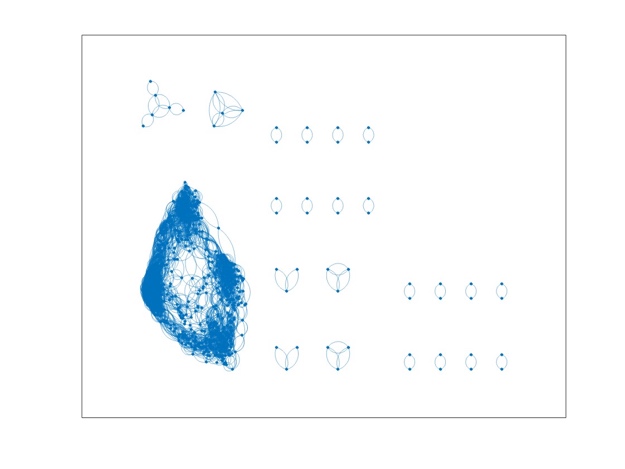
Estes gráficos têm o intuito de mostrar a distribuição das palavras do ficheiro pela *hash table.*

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

# Curiosidades

## Grafos

Uma imagem com texto

Descrição gerada automaticamenteUma imagem com texto

Descrição gerada automaticamente

Figura 24 | Grafo do ficheiro de 6 letras

Figura 22 | Grafo do ficheiro de 5 letras

Figura 22 | Grafo do ficheiro de 4 letras

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

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

## Informações dos grafos

**Ficheiro de 4 letras:**

**Exemplo de maior diâmetro:**

* Number of edges: 9267

[0] coai

[1] ceai

[2] coas

[3] coar

[4] coam

[5] cozi

[6] cosi

[7] comi

[8] coei

[9] zoai

[10] voai

[11] toai

[12] soai

[13] doai

[14] cear

[15] ceei

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

F**icheiro de 5 letras:**

**Exemplo de maior diâmetro:**

[30] mamal

[31] gamar

[32] mamam

[33] mamai

[15] datam

[16] danar

[17] datai

[18] datal

[19] manas

[20] minar

[21] sanar

[22] panar

[23] nanar

[24] lanar

[25] manam

[26] fanar

[27] manai

[28] mamas

[29] mimar

[0] matar

[1] datar

[2] manar

[3] mamar

[4] malar

[5] matam

[6] Catar

[7] matai

[8] catar

[9] matas

[10] maçar

[11] ratar

[12] datas

[13] dotar

[14] ditar

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

**Exemplo de maior diâmetro:**

**Ficheiro de 6 letras:**

* Number of edges: 36204

[15] voltes

[16] volvas

[17] volvem

[18] voavam

[19] soltei

[20] saltai

[21] soldai

[22] solhas

[23] soldas

[24] saltas

[25] soltos

[26] soltes

[27] solvas

[28] saltar

[29] soldar

[45] valvas

[46] voavas

[47] vulvas

[48] volveu

[49] volver

[50] voaram

[51] coavam

[52] toavam

[53] doavam

[54] zoavam

[55] solvei

[56] soldei

[57] saltei

[30] solvem

[31] soldem

[32] saltem

[33] soavam

[34] sorvam

[35] salvam

[36] silvam

[37] faltam

[38] salgam

[39] saldam

[40] sondam

[41] toldam

[42] moldam

[43] volvei

[44] volves

[0] voltam

[1] soltam

[2] voltai

[3] voltas

[4] voltar

[5] voltem

[6] volvam

[7] soltai

[8] soltas

[9] soltar

[10] soltem

[11] solvam

[12] saltam

[13] soldam

[14] voltei

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

**Ficheiro *default*:**

**Exemplo de maior diâmetro:**

* Number of edges: 1060534

[75] garramos

[76] torrados

[77] jorrados

[78] forrados

[79] bordados

[80] borrador

[81] mirrados

[82] berrador

[83] beirados

[84] serrados

[85] ferrados

[86] cerrados

[87] narrador

[88] mareados

[89] marcados

90] borravam

[91] birravam

[92] berravam

[50] marcadas

[51] mareadas

[52] carraças

[53] cardadas

[54] cariadas

[55] barbamos

[56] barbudos

[57] barbavam

[58] barbaram

[59] barbarás

[60] barbares

[61] bárbaras

[62] barrocos

[63] barricai

[64] barricar

[65] barricam

[66] barrigas

[67] barrámos

[68] barremos

[69] borramos

[70] birramos

[71] berramos

[72] varramos

[73] narramos

[74] marramos

[25] barravam

[26] borravas

[27] birravas

[28] berravas

[29] narravas

[30] marravas

[31] barraram

[32] barrarás

[33] barrares

[34] borraras

[35] birraras

[36] berraras

[37] narraras

[38] marraras

[39] borratas

[40] torradas

[41] porradas

[42] jorradas

[43] forradas

[44] bordadas

[45] mirradas

[46] beiradas

[47] serradas

[48] ferradas

[49] cerradas

[0] barradas

[1] barbadas

[2] barracas

[3] barrados

[4] barravas

[5] barraras

[6] borradas

[7] birradas

[8] berradas

[9] narradas

[10] marradas

[11] carradas

[12] barbados

[13] barbavas

[14] barbaras

[15] barbudas

[16] barracos

[17] barrocas

[18] barricas

[19] barramos

[20] borrados

[21] birrados

[22] berrados

[23] narrados

[24] marrados

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

## Informações das *hash tables*

**Ficheirode 5 letras:**

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

**Ficheirode 4 letras:**

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

**Ficheiro *default*:**

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

**Ficheirode 6 letras:**

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

## Cadeias interessantes de palavras

[0] pintor

[1] pintar

[2] pintas

[3] tintas

[0] rabo

[1] cabo

[2] caio

[3] cais

[4] caos

[5] cios

[6] aios

[7] aços

[8] açor

[9] apor

[10] opor

[11] odor

[0] cadeira

[1] caleira

[2] raleira

[3] raleara

[4] raleada

[5] ralhada

[6] rolhada

[7] rolhado

[8] folhado

[9] folhedo

[10] folheio

[11] folheis

[12] falheis

[13] falíeis

[14] faríeis

[15] feríeis

[16] geríeis

[17] gemíeis

[18] remíeis

[19] regíeis

[20] regreis

[21] regrais

[22] retrais

[23] retraia

[24] retrava

[25] recrava

[26] recuava

[27] recurva

[28] recurvo

[29] recurso

[0] subir

[1] subia

[2] sabia

[3] cabia

[4] caria

[5] carta

[6] casta

[7] cesta

[8] desta

[9] deste

[10] desce

[0] mãe

[1] mie

[2] fie

[3] fiz

[4] faz

[5] paz

[6] pai

[0] cebola

[1] rebola

[2] rebela

[3] rebeca

[4] rabeca

[5] rabeia

[6] rareia

[7] pareia

[8] pareis

[9] partis

[10] partas

[11] cartas

[12] cortas

[13] coutas

[14] chutas

[15] chuvas

[16] chovas

[17] choras

[18] chorar

[0] triste

[1] traste

[2] araste

[3] ataste

[4] ateste

[5] atente

[6] atende

[7] acende

[8] acendi

[9] acenai

[10] acenas

[11] arenas

[12] areias

[13] creias

[14] cheias

[15] checas

[16] chocas

[17] chocar

[18] chorar

[0] risada

[1] pisada

[2] pirada

[3] parada

[4] para-a

[5] pare-a

[6] pareia

[7] mareia

[8] moreia

[9] morria

[10] sorria

[11] sorrir

[0] musica

[1] musico

[2] mésico

[3] médico

[4] medico

[5] medi-o

[6] mede-o

[7] mede-a

[8] medeia

[9] meneia

[10] mentia

[11] sentia

[12] sentis

[13] sentas

[14] santas

[15] cantas

[16] cantos

[17] cantor

[0] palcos

[1] parcos

[2] partos

[3] pastos

[4] castos

[5] cantos

[6] cantor

[0] bebe

[1] bibe

[2] bise

[3] vise

[4] vive

[5] vivi

[6] viii

[7] xiii

[8] xixi

[1] chumbar

[2] chumbam

[3] chumbas

[4] chumbai

[5] chumbem

[6] chumbos

[7] chumbes

[8] chumbei

[9] chumbou

[0] bem

[1] tem

[2] teu

[3] meu

[4] mau

[5] mal

[0] bolo

[1] bola

[2] boca

[3] doca

[4] doce

## Confirmações de *memory leaks*

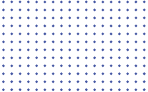
Uma imagem com texto

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

Figura 26 | Execução do programa através do valgrind

# 

# Conclusão



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

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

# 

# Apêndice

## Código c

//

// AED, November 2022 (Tomás Oliveira e Silva)

//

// Second practical assignement (word ladder)

//

// Place your student numbers and names here

// N.Mec. 107637 Name: André Oliveira

// N.Mec. 107634 Name: Duarte Cruz

// N.Mec. 107359 Name: Rodrigo Graça

//

// Do as much as you can

// 1) MANDATORY: complete the hash table code

// \*) hash\_table\_create

// \*) hash\_table\_grow

// \*) hash\_table\_free

// \*) find\_word

// +) add code to get some statistical data about the hash table

// 2) HIGHLY RECOMMENDED: build the graph (including union-find data) -- use the similar\_words function...

// \*) find\_representative

// \*) add\_edge

// 3) RECOMMENDED: implement breadth-first search in the graph

// \*) breadh\_first\_search

// 4) RECOMMENDED: list all words belonginh to a connected component

// \*) breadh\_first\_search

// \*) list\_connected\_component

// 5) RECOMMENDED: find the shortest path between to words

// \*) breadh\_first\_search

// \*) path\_finder

// \*) test the smallest path from bem to mal

// [ 0] bem

// [ 1] tem

// [ 2] teu

// [ 3] meu

// [ 4] mau

// [ 5] mal

// \*) find other interesting word ladders

// 6) OPTIONAL: compute the diameter of a connected component and list the longest word chain

// \*) breadh\_first\_search

// \*) connected\_component\_diameter

// 7) OPTIONAL: print some statistics about the graph

// \*) graph\_info

// 8) OPTIONAL: test for memory leaks

//

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

//

// static configuration

//

#define \_max\_word\_size\_ 32

//

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

//

typedef struct adjacency\_node\_s adjacency\_node\_t;

typedef struct hash\_table\_node\_s hash\_table\_node\_t;

typedef struct hash\_table\_s hash\_table\_t;

struct adjacency\_node\_s

{

adjacency\_node\_t \*next; // link to th enext adjacency list node

hash\_table\_node\_t \*vertex; // the other vertex

};

struct hash\_table\_node\_s

{

// the hash table data

char word[\_max\_word\_size\_]; // the word

hash\_table\_node\_t \*next; // next hash table linked list node

// the vertex data

adjacency\_node\_t \*head; // head of the linked list of adjancency edges

int visited; // visited status (while not in use, keep it at 0)

hash\_table\_node\_t \*previous; // breadth-first search parent

// the union find data

hash\_table\_node\_t \*representative; // the representative of the connected component this vertex belongs to

int number\_of\_vertices; // number of vertices of the conected component (only correct for the representative of each connected component)

int number\_of\_edges; // number of edges of the conected component (only correct for the representative of each connected component)

};

struct hash\_table\_s

{

unsigned int hash\_table\_size; // the size of the hash table array

unsigned int number\_of\_entries; // the number of entries in the hash table

unsigned int number\_of\_edges; // number of edges (for information purposes only)

hash\_table\_node\_t \*\*heads; // the heads of the linked lists

float average\_list\_length; // the average length of the linked lists

unsigned int max\_list\_length; // the maximum length of a linked list

unsigned int min\_list\_length; // the minimum length of a linked list

unsigned int empty\_lists; // the number of empty linked lists

};

//

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

//

static adjacency\_node\_t \*allocate\_adjacency\_node(void)

{

adjacency\_node\_t \*node;

node = (adjacency\_node\_t \*)malloc(sizeof(adjacency\_node\_t));

if (node == NULL)

{

fprintf(stderr, "allocate\_adjacency\_node: out of memory\n");

exit(1);

}

return node;

}

static hash\_table\_node\_t \*allocate\_hash\_table\_node(void)

{

hash\_table\_node\_t \*node;

node = (hash\_table\_node\_t \*)malloc(sizeof(hash\_table\_node\_t));

if (node == NULL)

{

fprintf(stderr, "allocate\_hash\_table\_node: out of memory\n");

exit(1);

}

return node;

}

//

// hash table stuff (mostly to be done)

//

unsigned int crc32(const char \*str)

{

static unsigned int table[256];

unsigned int crc;

if (table[1] == 0u) // do we need to initialize the table[] array?

{

unsigned int i, j;

for (i = 0u; i < 256u; i++)

for (table[i] = i, j = 0u; j < 8u; j++)

if (table[i] & 1u)

table[i] = (table[i] >> 1) ^ 0xAED00022u; // "magic" constant

else

table[i] >>= 1;

}

crc = 0xAED02022u; // initial value (chosen arbitrarily)

while (\*str != '\0')

crc = (crc >> 8) ^ table[crc & 0xFFu] ^ ((unsigned int)\*str++ << 24);

return crc;

}

static hash\_table\_t \*hash\_table\_create(void)

{

// create a new hash table

hash\_table\_t \*hash\_table;

// allocate the hash table

hash\_table = (hash\_table\_t \*)malloc(sizeof(hash\_table\_t));

// check for allocation errors

if (hash\_table == NULL)

{

fprintf(stderr, "create\_hash\_table: out of memory\n");

return -1;

}

// initialize the hash table

hash\_table->hash\_table\_size = 250;

hash\_table->number\_of\_entries = 0;

hash\_table->number\_of\_edges = 0;

// allocate the array of linked list heads

hash\_table->heads = (hash\_table\_node\_t \*\*)malloc(hash\_table->hash\_table\_size \* sizeof(hash\_table\_node\_t \*));

// check for allocation errors

if (hash\_table->heads == NULL)

{

fprintf(stderr, "create\_hash\_table: out of memory\n");

return -1;

}

// Fill the array of linked list heads with NULL

for (int i = 0; i < hash\_table->hash\_table\_size; i++)

{

hash\_table->heads[i] = NULL;

}

return hash\_table;

}

static void hash\_table\_grow(hash\_table\_t \*hash\_table)

{

// the old and new array of linked list heads

hash\_table\_node\_t \*\*old\_heads, \*\*new\_heads, \*node, \*next;

unsigned int old\_size, i;

// save a pointer to the old array of linked list heads and its size

old\_heads = hash\_table->heads;

old\_size = hash\_table->hash\_table\_size;

// create a new hash table with a larger size

hash\_table->hash\_table\_size \*= 2;

new\_heads = (hash\_table\_node\_t \*\*)malloc(hash\_table->hash\_table\_size \* sizeof(hash\_table\_node\_t \*));

// Fill the array of linked list heads with NULL

for (i = 0; i < hash\_table->hash\_table\_size; i++)

new\_heads[i] = NULL;

if (new\_heads == NULL)

{

fprintf(stderr, "Error: out of memory");

exit(1);

}

// rehash the entries from the old array to the new one

for (i = 0; i < old\_size; i++)

{

node = old\_heads[i];

while (node != NULL)

{

next = node->next;

size\_t value = crc32(node->word) % hash\_table->hash\_table\_size;

node->next = new\_heads[value];

new\_heads[value] = node;

node = next;

}

}

free(old\_heads);

hash\_table->heads = new\_heads;

}

static void hash\_table\_free(hash\_table\_t \*hash\_table)

{

hash\_table\_node\_t \*node;

hash\_table\_node\_t \*temp;

adjacency\_node\_t \*adj\_node;

adjacency\_node\_t \*temp\_adj;

unsigned int i;

for (i = 0; i < hash\_table->hash\_table\_size; i++)

{

node = hash\_table->heads[i];

while (node != NULL)

{

temp = node;

adj\_node = node->head;

while (adj\_node != NULL)

{

temp\_adj = adj\_node;

adj\_node = adj\_node->next;

free(temp\_adj);

}

node = node->next;

free(temp);

}

}

// Free the memory used by the array of linked list heads

free(hash\_table->heads);

// Free the memory used by the hash table

free(hash\_table);

}

static hash\_table\_node\_t \*find\_word(hash\_table\_t \*hash\_table, const char \*word, int insert\_if\_not\_found)

{

hash\_table\_node\_t \*node;

unsigned int i;

// find the word in the hash table

i = crc32(word) % hash\_table->hash\_table\_size;

node = hash\_table->heads[i];

while (node != NULL)

{

if (strcmp(node->word, word) == 0)

return node;

node = node->next;

}

// if the word is not found, insert it into the hash table

if (insert\_if\_not\_found && strlen(word) < \_max\_word\_size\_)

{

node = allocate\_hash\_table\_node(); // allocate a new node

strncpy(node->word, word, \_max\_word\_size\_); // copy the word into the node

node->representative = node; // the representative of a node is itself

node->next = hash\_table->heads[i]; // insert the node at the head of the linked list

node->previous = NULL;

node->number\_of\_edges = 0;

node->number\_of\_vertices = 1;

node->visited = 0;

node->head = NULL;

hash\_table->heads[i] = node;

hash\_table->number\_of\_entries++;

// if the number of entries exceeds the size of the hash table, grow the hash table

if (hash\_table->number\_of\_entries > hash\_table->hash\_table\_size)

hash\_table\_grow(hash\_table);

return node;

}

return NULL;

}

//

// add edges to the word ladder graph

//

hash\_table\_node\_t \*find\_representative(hash\_table\_node\_t \*node)

{

hash\_table\_node\_t \*representative, \*next\_node, \*actual\_node;

// Find the representative element by following the chain of representatives until we reach an element that points to itself

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

;

// Do a second pass to apply path compression, which flattens the tree representation of the disjoint-set and improves the efficiency of future find operations

for (next\_node = node; next\_node != representative; next\_node = next\_node->representative)

{

actual\_node = next\_node->representative;

next\_node->representative = representative;

}

return representative;

}

static void add\_edge(hash\_table\_t \*hash\_table, hash\_table\_node\_t \*from, const char \*word)

{

hash\_table\_node\_t \*to, \*from\_representative, \*to\_representative;

adjacency\_node\_t \*linkfrom, \*linkto;

// Find the word in the hash table

from\_representative = find\_representative(from);

to = find\_word(hash\_table, word, 0);

// If the word is not in the hash table, or if the two vertices are the same, return without adding an edge

if (to == NULL || to == from)

return;

// Find the representatives of the two connected components

to\_representative = find\_representative(to);

// If the vertices are already in the same connected component, increment the number of vertices in the component

if (from\_representative == to\_representative)

{

from\_representative->number\_of\_vertices++;

}

// If the vertices are not in the same connected component, merge the two connected components

if (from\_representative != to\_representative)

{

if (from\_representative->number\_of\_vertices < to\_representative->number\_of\_vertices)

{

from\_representative->representative = to\_representative;

to\_representative->number\_of\_vertices += from\_representative->number\_of\_vertices;

to\_representative->number\_of\_edges += from\_representative->number\_of\_edges;

}

else

{

to\_representative->representative = from\_representative;

from\_representative->number\_of\_vertices += to\_representative->number\_of\_vertices;

from\_representative->number\_of\_edges += to\_representative->number\_of\_edges;

}

}

// Allocate two adjacency nodes to represent the edge between the vertices

linkfrom = allocate\_adjacency\_node();

linkto = allocate\_adjacency\_node();

// If allocation fails, print an error message and exit

if (linkfrom == NULL || linkto == NULL)

{

fprintf(stderr, "add\_edge: out of memory\n");

exit(1);

}

// Update the linked lists of adjacency nodes for each vertex to include the new nodes

linkfrom->vertex = to;

linkfrom->next = from->head;

from->head = linkfrom;

linkto->vertex = from;

linkto->next = to->head;

to->head = linkto;

from\_representative->number\_of\_edges++;

to\_representative->number\_of\_edges++;

hash\_table->number\_of\_edges++;

return;

}

//

// generates a list of similar words and calls the function add\_edge for each one

//

// man utf8 for details on the uft8 encoding

//

static void break\_utf8\_string(const char \*word, int \*individual\_characters)

{

int byte0, byte1;

while (\*word != '\0')

{

byte0 = (int)(\*(word++)) & 0xFF;

if (byte0 < 0x80)

\*(individual\_characters++) = byte0; // plain ASCII character

else

{

byte1 = (int)(\*(word++)) & 0xFF;

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

{

fprintf(stderr, "break\_utf8\_string: unexpected UFT-8 character\n");

exit(1);

}

\*(individual\_characters++) = ((byte0 & 0b00011111) << 6) | (byte1 & 0b00111111); // utf8 -> unicode

}

}

\*individual\_characters = 0; // mark the end!

}

static void make\_utf8\_string(const int \*individual\_characters, char word[\_max\_word\_size\_])

{

int code;

while (\*individual\_characters != 0)

{

code = \*(individual\_characters++);

if (code < 0x80)

\*(word++) = (char)code;

else if (code < (1 << 11))

{ // unicode -> utf8

\*(word++) = 0b11000000 | (code >> 6);

\*(word++) = 0b10000000 | (code & 0b00111111);

}

else

{

fprintf(stderr, "make\_utf8\_string: unexpected UFT-8 character\n");

exit(1);

}

}

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

}

static void similar\_words(hash\_table\_t \*hash\_table, hash\_table\_node\_t \*from)

{

static const int valid\_characters[] =

{ // 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

//

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

//

static int breadh\_first\_search(int maximum\_number\_of\_vertices, hash\_table\_node\_t \*\*list\_of\_vertices, hash\_table\_node\_t \*origin, hash\_table\_node\_t \*goal)

{

// Indices for the read and write positions of the queue

int read = 0, write = 1;

// Initialize the first position of the queue with the origin vertex [origin, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL]

list\_of\_vertices[0] = origin;

// Initialize the previous vertex of the origin as NULL

origin->previous = NULL;

// Mark the origin vertex as visited

origin->visited = 1;

int found = 0;

// While the queue is not empty

while (read != write)

{

// node is set to the head of the origin vertex to begin traversing the linked list

adjacency\_node\_t \*node = list\_of\_vertices[read++]->head;

if (found == 1)

{

break;

}

while (node != NULL) // Traverse the linked list

{

// Check if the vertex has already been visited

if (node->vertex->visited == 0)

{

node->vertex->visited = 1;

// The previous vertex of this vertex is the vertex before it in the queue

node->vertex->previous = list\_of\_vertices[read - 1];

// Add the vertex to the queue using the write position

list\_of\_vertices[write++] = node->vertex;

if (node->vertex == goal)

{

found = 1;

break;

}

}

node = node->next;

}

}

for (int i = 0; i < write; i++)

{

list\_of\_vertices[i]->visited = 0;

}

return write;

}

//

// list all vertices belonging to a connected component

//

static int list\_connected\_component(hash\_table\_t \*hash\_table, const char \*word, int option)

{

// Find the given word in the hash table

hash\_table\_node\_t \*node = find\_word(hash\_table, word, 0);

int vertices = 0;

if (node == NULL)

{

fprintf(stderr, "Word not found\n");

return -1;

}

// Get the representative node of the connected component containing the word

hash\_table\_node\_t \*representative = find\_representative(node);

// Allocate memory for a list of vertices in the connected component

hash\_table\_node\_t \*\*list\_of\_vertices = (hash\_table\_node\_t \*\*)malloc(representative->number\_of\_vertices \* sizeof(hash\_table\_node\_t \*));

// Perform a breadth-first search to find all the vertices in the connected component

int num\_vertices = breadh\_first\_search(hash\_table->number\_of\_entries, list\_of\_vertices, node, NULL);

// Print all the vertices in the connected component

if (option == 1)

{

printf("Connected component:\n");

}

for (int i = 0; i < num\_vertices; i++)

{

vertices++;

if (option == 1)

{

printf("[%d] %s\n", vertices, list\_of\_vertices[i]->word);

}

}

free(list\_of\_vertices);

return vertices;

}

//

// compute the diameter of a connected component

//

static int largestDiameter, smallestDiameter, diametersSum, numDiameters;

static hash\_table\_node\_t \*\*largestDiameterList;

static int maxNumVertices;

static connected\_component\_diameter(hash\_table\_node\_t \*node)

{

int diameter = -1;

int numVertices;

hash\_table\_node\_t \*\*vertices, \*\*temporaryList;

maxNumVertices = find\_representative(node)->number\_of\_vertices;

vertices = (hash\_table\_node\_t \*\*)malloc(maxNumVertices \* sizeof(hash\_table\_node\_t \*));

temporaryList = (hash\_table\_node\_t \*\*)malloc(maxNumVertices \* sizeof(hash\_table\_node\_t \*));

if (vertices == NULL || temporaryList == NULL)

{

fprintf(stderr, "connected\_component\_diameter: out of memory\n");

exit(1);

}

numVertices = breadh\_first\_search(maxNumVertices, vertices, node, NULL);

for (int i = 0; i < numVertices; i++)

{

int temporaryNumVertices = breadh\_first\_search(maxNumVertices, temporaryList, vertices[i], vertices[i]);

int temporaryDiameter = 0;

hash\_table\_node\_t \*palavras = temporaryList[temporaryNumVertices - 1];

while (palavras != NULL)

{

temporaryDiameter++;

palavras = palavras->previous;

}

if (temporaryDiameter > diameter)

{

diameter = temporaryDiameter - 1;

}

}

if (diameter > largestDiameter)

{

largestDiameter = diameter;

largestDiameterList = (hash\_table\_node\_t \*\*)malloc(numVertices \* sizeof(hash\_table\_node\_t \*));

if (largestDiameterList == NULL)

{

fprintf(stderr, "connected\_component\_diameter: out of memory\n");

exit(1);

}

largestDiameterList = vertices;

}

if (diameter < smallestDiameter)

{

smallestDiameter = diameter;

}

if (diameter == -1)

{

printf("connected\_component\_diameter: diameter not found\n");

return -1;

}

numDiameters++;

diametersSum += diameter;

free(temporaryList);

free(vertices);

}

//

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

//

static void path\_finder(hash\_table\_t \*hash\_table, const char \*from\_word, const char \*to\_word)

{

// Find the hash table nodes corresponding to the given words.

hash\_table\_node\_t \*from, \*fromRepresentative, \*to, \*toRepresentative;

from = find\_word(hash\_table, from\_word, 0);

to = find\_word(hash\_table, to\_word, 0);

// Find the representatives of the connected components containing the words.

fromRepresentative = find\_representative(from);

toRepresentative = find\_representative(to);

// If one of the words doesn't exist in the hash table, print an error message and return.

if (from == NULL || to == NULL)

{

printf("One of the words doesn't exist\n");

return;

}

// If the words aren't in the same connected component, print an error message and return.

if (fromRepresentative != toRepresentative)

{

printf("The words aren't in the same connected component\n");

return;

}

// Allocate an array of pointers to hash table nodes to store the vertices of the shortest path.

hash\_table\_node\_t \*\*vertices = malloc(sizeof(hash\_table\_node\_t \*) \* fromRepresentative->number\_of\_vertices);

// Find the shortest path between the two words using a breadth-first search.

int path = breadh\_first\_search(fromRepresentative->number\_of\_vertices, vertices, to, from);

// Print the words on the shortest path in order, starting from the destination word and ending at the source word.

hash\_table\_node\_t \*palavras = vertices[path - 1];

int ordem = 0;

while (palavras != NULL)

{

printf("[%d] %s \n", ordem, palavras->word);

ordem++;

palavras = palavras->previous;

}

// Free the array of vertices.

free(vertices);

}

//

// some graph information (optional)

//

static int find\_connected\_component\_representatives(hash\_table\_t \*hash\_table, hash\_table\_node\_t \*\*representatives)

{

int nrRepresentatives = 0;

// Find the representatives of each connected component

for (int i = 0; i < hash\_table->hash\_table\_size; i++)

{

for (hash\_table\_node\_t \*vertex = hash\_table->heads[i]; vertex != NULL; vertex = vertex->next)

{

// Find the representative of the connected component

hash\_table\_node\_t \*representative = find\_representative(vertex);

// Add the representative to the array if it has not already been added

if (!representative->visited)

{

representatives[nrRepresentatives++] = representative->word;

representative->visited = 1;

}

}

}

// Reset the visited status of all vertices

for (int i = 0; i < hash\_table->hash\_table\_size; i++)

{

for (hash\_table\_node\_t \*vertex = hash\_table->heads[i]; vertex != NULL; vertex = vertex->next)

{

vertex->visited = 0;

}

}

return nrRepresentatives;

}

static void graph\_info(hash\_table\_t \*hash\_table)

{

// Allocate an array to store representatives (one representative per connected component)

hash\_table\_node\_t \*\*representatives = malloc(sizeof(hash\_table\_node\_t) \* hash\_table->hash\_table\_size);

// Find the representatives for each connected component in the hash table

int nrRepresentatives = find\_connected\_component\_representatives(hash\_table, representatives);

// Reset the global variables that track the largest and smallest diameters

largestDiameter = 0;

smallestDiameter = hash\_table->number\_of\_entries;

// For each representative node, find the diameter of its connected component

for (int i = 0; i < nrRepresentatives; i++)

{

connected\_component\_diameter(representatives[i]);

}

// calcula tamanho maximo de componente conexa

int min = hash\_table->number\_of\_entries;

int max = 0;

int total = 0;

for (int i = 0; i < nrRepresentatives; i++)

{

int size = list\_connected\_component(hash\_table, representatives[i]->word, 2);

if (size > max)

{

max = size;

}

if (size < min)

{

min = size;

}

total += size;

}

// Print out various statistics about the graph

printf("\nNumber of edges: %u\n", hash\_table->number\_of\_edges);

printf("Number of vertices: %u\n", hash\_table->number\_of\_entries);

printf("Number of different representatives: %d\n", numDiameters);

printf("Largest connected component: %u\n", max);

printf("Smallest connected component: %u\n", min);

printf("Average connected component: %.2f\n", (float)total / nrRepresentatives);

printf("Largest diameter: %d\n", largestDiameter);

printf("Smallest diameter: %d\n", smallestDiameter);

printf("Average of diameters: %.2f\n\n", (float)diametersSum / numDiameters);

// Print an example of a word chain in the largest connected component

printf("Largest diameter example: \n");

for (int i = 0; i < largestDiameter + 1; i++)

{

printf("[%d] %s \n", i, largestDiameterList[i]->word);

}

printf("\n");

free(representatives);

}

static void hash\_table\_info(hash\_table\_t \*hash\_table)

{

unsigned int total\_list\_length = 0;

unsigned int list\_length;

hash\_table\_node\_t \*node;

unsigned int i;

// Initialize the statistical data

hash\_table->average\_list\_length = 0.0;

hash\_table->max\_list\_length = 0;

hash\_table->min\_list\_length = \_max\_word\_size\_;

hash\_table->empty\_lists = 0;

// Iterate through the array of linked list heads and calculate the statistical data

for (i = 0; i < hash\_table->hash\_table\_size; i++)

{

list\_length = 0;

node = hash\_table->heads[i];

while (node != NULL)

{

list\_length++;

node = node->next;

}

if (list\_length > hash\_table->max\_list\_length)

{

hash\_table->max\_list\_length = list\_length;

}

if (list\_length < hash\_table->min\_list\_length && list\_length != 0)

{

hash\_table->min\_list\_length = list\_length;

}

if (list\_length == 0)

{

hash\_table->empty\_lists++;

}

total\_list\_length += list\_length;

}

// Calculate the average list length

hash\_table->average\_list\_length = (float)total\_list\_length / hash\_table->hash\_table\_size;

// Print the statistical data

printf("\nNumber of entries: %u\n", hash\_table->number\_of\_entries);

printf("Hash table size: %u\n", hash\_table->hash\_table\_size);

printf("Average list length: %.2f\n", hash\_table->average\_list\_length);

printf("Max list length: %u\n", hash\_table->max\_list\_length);

printf("Min list length: %u\n", hash\_table->min\_list\_length);

printf("Number of empty lists: %u\n\n", hash\_table->empty\_lists);

}

//

// main program

//

int main(int argc, char \*\*argv)

{

char word[100], from[100], to[100];

hash\_table\_t \*hash\_table;

hash\_table\_node\_t \*node;

unsigned int i;

int command;

FILE \*fp;

printf("Inicialização do programa\n");

// initialize hash table

hash\_table = hash\_table\_create();

// read words

fp = fopen((argc < 2) ? "wordlist-big-latest.txt" : argv[1], "rb");

if (fp == NULL)

{

fprintf(stderr, "main: unable to open the words file\n");

exit(1);

}

while (fscanf(fp, "%99s", word) == 1)

{

(void)find\_word(hash\_table, word, 1);

// printf("%d %d\n", hash\_table->hash\_table\_size, hash\_table->number\_of\_entries);

}

fclose(fp);

// return 0;

// find all similar words

for (i = 0u; i < hash\_table->hash\_table\_size; i++)

{

for (node = hash\_table->heads[i]; node != NULL; node = node->next)

{

similar\_words(hash\_table, node);

}

}

/\* for (unsigned int i = 0; i < hash\_table->hash\_table\_size; i++) // loop through the hash table

{

hash\_table\_node\_t \*node = hash\_table->heads[i]; // set node to the first element of the hash table

while (node != NULL) // while the node has a next node

{

hash\_table\_node\_t \*temp = node; // set temp to the node

node = node->next; // set node to the next node // free the temp node

adjacency\_node\_t \*adj\_node = temp->head; // set adj\_node to the first element of the adjacency list

while (adj\_node != NULL) // while the adj\_node has a next node

{

adjacency\_node\_t \*temp\_adj = adj\_node; // set temp\_adj to the adj\_node

adj\_node = adj\_node->next; // set adj\_node to the next node

printf("%s %s\n", temp->word, temp\_adj->vertex->word); // print the word and the word in the adjacency list")

}

}

}

return 0; \*/

// ask what to do

for (;;)

{

fprintf(stderr, "Your wish is my command:\n");

fprintf(stderr, " 1 WORD (list the connected component WORD belongs to)\n");

fprintf(stderr, " 2 FROM TO (list the shortest path from FROM to TO)\n");

fprintf(stderr, " 3 (hash info)\n");

fprintf(stderr, " 4 (graph info)\n");

fprintf(stderr, " 5 (print words)\n");

fprintf(stderr, " 6 (terminate)\n");

fprintf(stderr, "> ");

if (scanf("%99s", word) != 1)

break;

command = atoi(word);

if (command == 1)

{

if (scanf("%99s", word) != 1)

break;

list\_connected\_component(hash\_table, word, 1);

}

else if (command == 2)

{

if (scanf("%99s", from) != 1)

break;

if (scanf("%99s", to) != 1)

break;

path\_finder(hash\_table, from, to);

}

else if (command == 3)

{

hash\_table\_info(hash\_table);

}

else if (command == 4)

{

graph\_info(hash\_table);

}

else if (command == 5)

{

int numPalavras = 0;

for (i = 0; i < hash\_table->hash\_table\_size; i++)

{

for (node = hash\_table->heads[i]; node != NULL; node = node->next)

{

printf("indice = %u -> %s\n", hash\_table->heads[i], node->word);

numPalavras++;

}

}

printf("Número de palavras = %d\n", numPalavras);

}

else if (command == 6)

break;

else

{

fprintf(stderr, "Invalid command\n");

}

}

// clean up

hash\_table\_free(hash\_table);

return 0;

}

## Código Matlab

%% Hash Table Size Big

valores = load("hashTableSizeBig.txt");

hashTableSize = valores(:,1);

numEntries = valores(:,2);

plot(numEntries,hashTableSize);

axis([0,1000000,0,1200000]);

ylabel("Hash Table Size");

xlabel("Number of Entries");

%% Gráfico Barras Big

valores = load("histogramaBig.txt");

indexs = valores(:,1);

numWords = valores(:,2);

bar(indexs,numWords);

ylabel("Number of Words");

xlabel("Index");

%% Grafo Big

data = readtable("graphBig.txt");

G = graph;

words = unique(data(:,1));

for i= 1:height(words)

G = addnode(G,words{i,1});

end

for i= 1:height(data)

G = addedge(G,data{i,1},data{i,2});

end

plot(G)

%% Hash Table Size Four

valores = load("hashTableSizeFour.txt");

hashTableSize = valores(:,1);

numEntries = valores(:,2);

plot(numEntries,hashTableSize);

axis([0,2500,0,4100]);

ylabel("Hash Table Size");

xlabel("Number of Entries");

%% Gráfico Barras Four

valores = load("histogramaFour.txt");

indexs = valores(:,1);

numWords = valores(:,2);

bar(indexs,numWords);

ylabel("Number of Words");

xlabel("Index");

%% Grafo Four

data = readtable("graphFour.txt");

G = graph;

words = unique(data(:,1));

for i= 1:height(words)

G = addnode(G,words{i,1});

end

for i= 1:height(data)

G = addedge(G,data{i,1},data{i,2});

end

plot(G)

%% Hash Table Size Five

valores = load("hashTableSizeFive.txt");

hashTableSize = valores(:,1);

numEntries = valores(:,2);

plot(numEntries,hashTableSize);

axis([0,8000,0,8100]);

ylabel("Hash Table Size");

xlabel("Number of Entries");

%% Gráfico Barras Five

valores = load("histogramaFive.txt");

indexs = valores(:,1);

numWords = valores(:,2);

bar(indexs,numWords);

ylabel("Number of Words");

xlabel("Index");

%% Grafo Five

data = readtable("graphFive.txt");

G = graph;

words = unique(data(:,1));

for i= 1:height(words)

G = addnode(G,words{i,1});

end

for i= 1:height(data)

G = addedge(G,data{i,1},data{i,2});

end

plot(G)

%% Hash Table Size Six

valores = load("hashTableSizeSix.txt");

hashTableSize = valores(:,1);

numEntries = valores(:,2);

plot(numEntries,hashTableSize);

axis([0,16000,0,16100]);

ylabel("Hash Table Size");

xlabel("Number of Entries");

%% Gráfico Barras Six

valores = load("histogramaSix.txt");

indexs = valores(:,1);

numWords = valores(:,2);

bar(indexs,numWords);

ylabel("Number of Words");

xlabel("Index");

%% Grafo Six

data = readtable("graphSix.txt");

G = graph;

words = unique(data(:,1));

for i= 1:height(words)

G = addnode(G,words{i,1});

end

for i= 1:height(data)

G = addedge(G,data{i,1},data{i,2});

end

plot(G)

## *Printscreens* resultados mostrados anteriormente

Uma imagem com texto

Descrição gerada automaticamente**Ficheiro 4 letras**

Figura 28 | Hash Info 4 letras

**Uma imagem com texto

Descrição gerada automaticamente**

Figura 27 | Graph Info 4 letras

**Uma imagem com texto

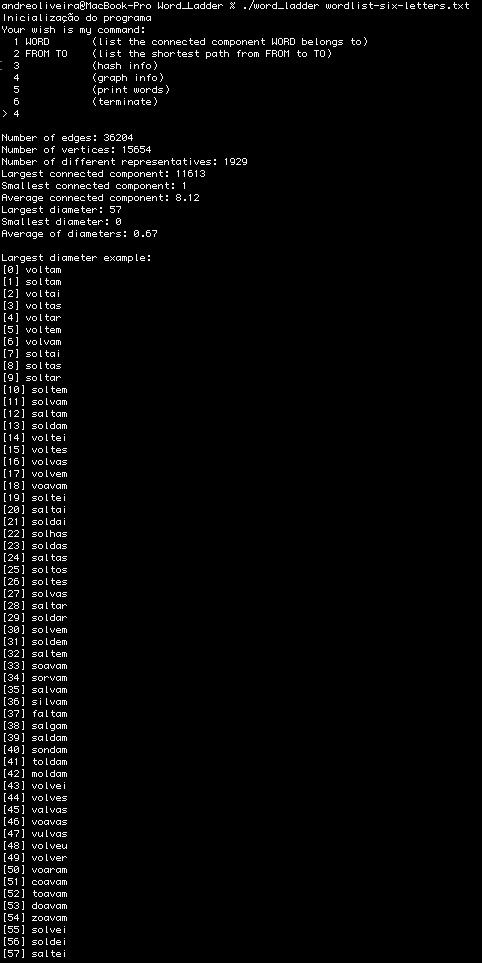
Descrição gerada automaticamenteFicheiro 5 letras**

Figura 30 | Hash Info 5 letras

**Uma imagem com texto

Descrição gerada automaticamente**

Figura 29 | Graph Info 5 letras

**Uma imagem com texto

Descrição gerada automaticamenteFicheiro 6 letras**

Figura 32 | Hash Info 6 letras

Figura 31 | Graph Info 6 letras

**Uma imagem com mesa

Descrição gerada automaticamenteUma imagem com texto, quadro de resultados, placa

Descrição gerada automaticamenteUma imagem com texto, quadro de resultados, placa

Descrição gerada automaticamenteUma imagem com texto

Descrição gerada automaticamenteFicheiro *default***

Figura 33 | Graph Info default

Figura 34 | Hash Info default

**Uma imagem com texto

Descrição gerada automaticamente**