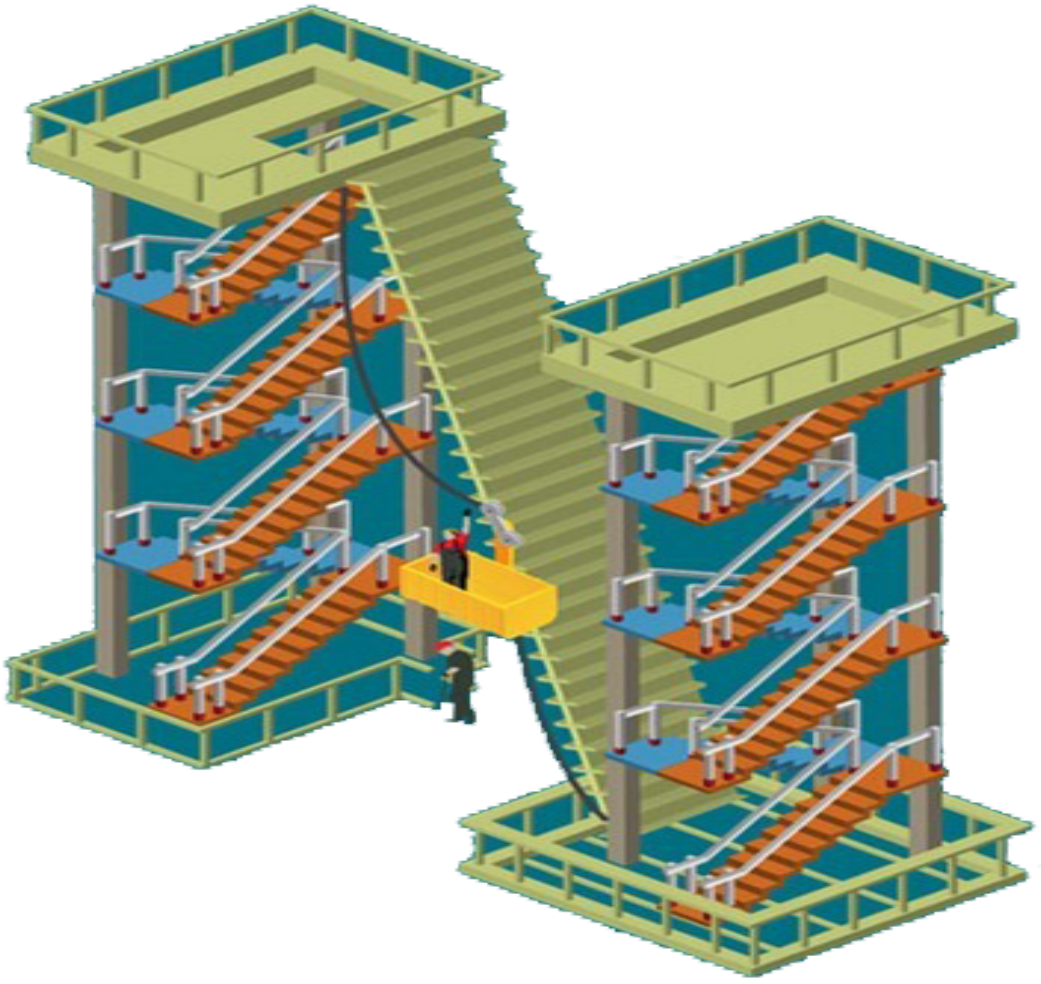
universidade de aveiro deti departamento de electrónica, telecomunicações e informática

Word Ladder



Algoritmos e Estruturas de Dados
Relatório Trabalho 02 - Turma P4
Prof. Pedro Lavrador
2022/23

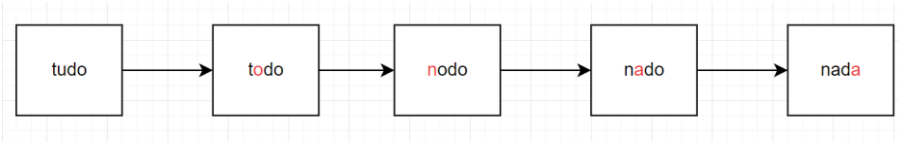
Maria Rafaela Alves Abrunhosa, 107658-50%
Matilde Moital Portugal Sampaio Teixeira, 108193-50%

Índice

Introdução

O segundo projeto de Algoritmos e Estruturas de Dados baseia-se no entendimento e bom funcionamento de *hash tables* bem como de algoritmos como *Breadth-first search* e outros. Nestes é necessário o controlo de diversas técnicas de resolução de algoritmos, *sorting*, *hash tables*, *structs*, estruturas de dados para grafos, grafos, saber trabalhar com *nodes* adjacentes de um grafo de forma a se encontrar um menor caminho de um vértice *a* a um vértice *b* e, ainda, saber lidar e ter atenção com o endereçamento de memória tal como saber realocá-la.

Mais concretamente, este projeto visa encontrar o menor percurso para se chegar a uma palavra *b*, começando na palavra *a* e apenas mudando uma letra de palavra a palavra até que cheguemos à palavra final.



O ficheiro com o nome *word_ladder.c* contém o código em C com diversas funções para a resolução do problema.

Metodologia

Contextualização do problema

Tal como o nome do ficheiro C *word_ladder.c* indica, uma *word ladder* é uma sequência de palavras na qual duas palavras adjacentes diferem, apenas, numa letra. Sendo possível ir de palavras a palavras como tudo -> nada.
O menor percurso deste exemplo requer quatro passos: tudo -> todo -> nodo -> noda -> nada.

Para a realização do projeto, será necessário ir completando funções ao longo do ficheiro *word_ladder.c*, como é o caso de funções que permitam o bom funcionamento de uma *hash table* que reajustará dinamicamente o seu tamanho para quando lhe forem adicionadas mais informação, esta que no seu estado inicial, não conseguiria albergar.
Será preciso, também, lidar com grafos unidirecionais, este que será o percurso pelo qual cada palavra chega a outra sem ter de voltar a ela mesma, o que criaria loops, ou laços em grafos.

Estruturas de dados internos

Antes de se completar as funções é necessário estudar e perceber as estruturas de dados declaradas essenciais à resolução do programa. Serão declaradas três *structs* principais onde estarão definidas uma lista de variáveis agrupadas num bloco de memória.

As três *structs* criadas são do tipo *adjacency_node_s*, *hash_table_node_s* e *hash_table_s*.

A primeira *struct*, *adjacency_node_s*, é constituída por dois parâmetros, uma ligação para o lista do nó adjacente seguinte do tipo *adjacency_node_t* e um *vertex* do tipo *hash_table_node_t*.

Na segunda *struct*, *hash_table_node_s*, guardam-se todos os dados referentes à *hash table*: uma palavra do tipo *char* que será a *key* da *hash table* e poderá ter um *_max_word_size_* definido inicialmente como 32 (*word*).

Esta também albergará o próximo nó da lista linkada da *hash table* (**next*), o nó anterior ou o parente do *breadth-first search* (**previous*) e um representante do componente conectado ao qual o *vertex* pertence (**representative*) do tipo *hash_table_node_t*.

Além disso, esta também apresenta a 'cabeça' da *linked list* das arestas adjacentes (**head*), *adjacency_node_s* e, por fim, alguns *int*, o *visited* que apresenta o estado do que já foi visitado, o *number_of_vertices* que conta o número de vértices do componente ligado e o *number_of_edges* que guarda o número de arestas do componente ligado.

Na terceira e última *struct*, *hash_table_s*, guardamos o tamanho do array da *hash table* (*hash_table_size*), o número de entradas da *hash table* (*number_of_entries*) e o número de arestas (*number_of_edges*) em três *unsigned int* e as *heads* das *linked lists* do tipo *hash_table_node_t* (***heads*).

```
// data structures

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 - keys
 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 -> if the representatives of
their connected componments are the same, they are connected
 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)
};
```

Funções

1. create

Esta função é usada para criar uma nova tabela de hash. Ela aloca memória dinamicamente usando a função malloc para criar um novo objeto *hash_table_t* e verifica se há memória suficiente.

Em seguida, ela inicializa o tamanho da tabela de hash como 103, o número de entradas e arestas como 0. E aloca memória para as "*heads*" da tabela de *hash*, que é um array de ponteiros para nós da tabela de *hash*.

Então, usa um loop para inicializar cada índice da tabela de hash como NULL. Por fim, retorna o ponteiro para a tabela de hash recém-criada.

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

 hash_table = (hash_table_t *)malloc(sizeof(hash_table_t)); //
malloc is used to dynamically allocate a single
 // large block of memory with the specified size
 if(hash_table == NULL)
 {
   fprintf(stderr,"create_hash_table: out of memory\n");
   exit(1);
 }
 //
 // complete this
 //
 hash_table->hash_table_size = 103;
 hash_table->number_of_entries = 0;
 hash_table->number_of_edges = 0;

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

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

2. grow

Esta função é usada para aumentar o tamanho da tabela de hash quando o número de entradas na tabela ultrapassa o tamanho atual da tabela.

Primeiro, ela salva a tabela de hash antiga e o seu tamanho em variáveis temporárias. Em seguida, ela duplica o tamanho da tabela de hash e aloca memória para uma nova tabela de hash com esse novo tamanho. Em seguida, ela inicializa o número de entradas e arestas como 0 e inicializa cada índice da nova tabela de hash como NULL.

Depois, usa um loop para percorrer cada índice da tabela de hash antiga e para cada nó na lista ligada desse índice. Calcula-se o novo índice para cada nó usando a função crc32 e adiciona-se o nó à lista ligada no novo índice.

Finalmente, liberta-se a memória alocada para a tabela de hash antiga.

```
static void hash_table_grow(hash_table_t *hash_table){
 hash_table_node_t **old_hash_table, *head_one, *next;
 unsigned int old_size, i, index;
 //
 // complete this
 //

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

 hash_table->hash_table_size = hash_table->hash_table_size*2+1; //
double the size of the hash table

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

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

```
  }

  // check for out of memory
  for (i = 0; i < hash_table->hash_table_size; i++) {
    hash_table->heads[i] = NULL;
  }

  for (i = 0; i < old_size; i++){
    //
    // complete this
    //
    head_one = old_hash_table[i];
    while(head_one != NULL){
      //
      // complete this
      //
      index = crc32(head_one->word) % hash_table->hash_table_size;
      next = head_one->next;
      // ...
      head_one->next = hash_table->heads[index];
      hash_table->heads[index] = head_one;
      head_one = next;
   }
  }
  free(old_hash_table);
}
```

3. free

Esta função liberta a memória alocada para uma tabela de *hash*.

Primeiro, define o tamanho da tabela, o número de entradas e o número de arestas como 0. Em seguida, usa um loop para percorrer cada índice da tabela de hash.

Para cada índice, a função percorre a lista ligada desse índice e para cada nó, percorre a sua lista de adjacência e liberta a memória alocada para cada nó adjacente. Depois disso, liberta o nó da lista ligada. Após esse processo, liberta a memória alocada para o array de listas ligadas e finalmente para a tabela de hash em si.

```
static void hash_table_free(hash_table_t *hash_table)
{
 hash_table_node_t *head_one, *next;
 unsigned int i;
```

```
    //
    // complete this
    //

    hash_table->hash_table_size = 103;
    hash_table->number_of_entries = 0;
    hash_table->number_of_edges = 0;

    for (i = 0; i < hash_table->hash_table_size; i++){
      //
      // complete this
      //
      head_one = hash_table->heads[i];
      while(head_one != NULL){
        adjacency_node_t *nodeadj = head_one->head, *nextAdjNode;
        while(nodeadj != NULL){
        // complete this
        nextAdjNode = nodeadj->next;
        free_adjacency_node(nodeadj);
        nodeadj=nextAdjNode;
        }
        next = head_one->next;
        free_hash_table_node(head_one);
        head_one = next;
     }
    }
    free(hash_table->heads);
    free(hash_table);
  }
```

4. find_word

Esta função procura por uma palavra específica numa tabela de hash e retorna um ponteiro para o nó correspondente, se encontrado.

Usa a função crc32 para calcular o índice da palavra na tabela e, em seguida, percorre a lista ligada no índice correspondente à procura da palavra. Se a palavra for encontrada, a função retorna o ponteiro para o nó.

Se a palavra não for encontrada e o parâmetro *insert_if_not_found* for verdadeiro e o tamanho da palavra for menor que o tamanho máximo permitido, a função aloca um novo nó, insere a palavra nele e adiciona-o à lista ligada no índice correspondente. Para além disso, verifica se o número de entradas na tabela

ultrapassou o tamanho da tabela e, nesse caso, usa a função *hash_table_grow* para aumentar o tamanho da tabela.

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

```
static hash_table_node_t *find_word(hash_table_t *hash_table,
const char *word, int insert_if_not_found)
{
hash_table_node_t *node1;
unsigned int index;
index = crc32(word) % hash_table->hash_table_size;
node1 = hash_table->heads[index];
while(node1 != NULL)
{
  if(strcmp(node1->word,word) == 0)
    return node1;
  node1 = node1->next;
}
if(insert_if_not_found && strlen(word) < _max_word_size_)
{
  node1 = allocate_hash_table_node();
  strncpy(node1->word,word,_max_word_size_);
  node1->representative = node1;
  node1->next = hash_table->heads[index];
  node1->previous = NULL;
  node1->number_of_edges = 0;
  node1->number_of_vertices = 1;
  node1-> visited = 0;
  node1->head = NULL;
  hash_table->heads[index] = node1;
  hash_table->number_of_entries++;
  if(hash_table->number_of_entries > hash_table->hash_table_size)
    hash_table_grow(hash_table);
  return node1;
}
 return NULL;
}
```

5. find_representative

A função *find_representative*() é usada para encontrar o nó representativo de uma componente conectada num grafo, representado por uma *hash table*.

Passa-se como parâmetro um nó específico do grafo (*node*).
Primeiramente, é criada uma variável chamada *representative* que é inicializada com o nó passado como parâmetro.
De seguida, é criado um *loop while* que continuará a executar até que *representative* aponte para si mesmo. Dentro do *loop*, a variável *representative* é atualizada para apontar para o nó apontado pelo seu ponteiro.
Isso é feito até que o nó apontado por *representative* seja o próprio nó, ou seja, o nó representativo da componente conectada.

Posteriormente, é criado outro *loop while*, que começa com o nó passado como parâmetro e continuará até que o nó atual seja o nó representativo encontrado anteriormente.
Dentro do *loop*, é criada uma variável temporária *temp_node* que é inicializada como o nó apontado pelo ponteiro do *representative* do nó atual.
O ponteiro de representante do nó atual é atualizado para apontar diretamente para o nó representativo encontrado anteriormente.
Por fim, o nó atual é atualizado para a variável temporária *temp_node*, para continuar a percorrer a cadeia de nós da componente conectada.

Essa implementação é conhecida como compressão de caminho (*path compression*) e é usada para otimizar o algoritmo de busca do nó representativo. Faz com que a busca do nó representativo seja mais rápida e evita que a árvore de representação fique muito desiquilibrada.

No fim, a função retorna o nó representativo encontrado.

```
static hash_table_node_t *find_representative(hash_table_node_t
*node)
{
   hash_table_node_t *Representative, *next_node, *temp_node;

   Representative = node;
   while (Representative != Representative->representative)
   {
       Representative = Representative->representative;
   }
   next_node = node;
   while (next_node != Representative)
```

```
        {
            temp_node = next_node->representative;
            next_node->representative = Representative;
            next_node = temp_node;
        }
        return Representative;
    }
```

6. add_edge

Este código é uma função em C que adiciona uma aresta entre dois vértices numa estrutura de dados de *hash table*. A tabela *hash* é representada pela estrutura *hash_table_t*, e os vértices são representados pela estrutura *hash_table_node_t*. A função recebe três parâmetros: um ponteiro para a tabela *hash*, um ponteiro para o vértice de origem e uma string que representa a palavra que será o vértice de destino.

A função começa por encontrar os representantes dos vértices de origem e destino usando a função *find_representative*.

Seguidamente, verifica se o vértice de destino já existe na tabela *hash*, e se o vértice de destino é o mesmo que o de origem. Se essa verificação retornar verdadeiro, a função retorna sem adicionar uma aresta.

Se os vértices de origem e destino são diferentes, a função verifica se eles já estão conectados (se os seus representantes são os mesmos). Se já estiverem conectados, o número de arestas do representante é incrementado. Caso contrário, os representantes são fundidos (*representative1* é atribuído a *representative2*), e o número de arestas e vértices dos representantes é atualizado.

Em seguida, aloca-se memória para duas novas estruturas *adjacency_node_t*, chamadas *link1* e *link2*. Essas estruturas representam as arestas entre os vértices de origem e destino. Se a alocação de memória falhar, a função exibe uma mensagem de erro e termina o programa.

Finalmente, os ponteiros dos nós adjacentes são atualizados, e o número de arestas na *hash table* é incrementado. A função não retornar nenhum valor.

```
    static void add_edge(hash_table_t *hash_table,hash_table_node_t
*from,const char *word)
    {
     hash_table_node_t *to_node,*representative1,*representative2;
     adjacency_node_t *link1, *link2;
```

```
        representative1 = find_representative(from);
        to_node = find_word(hash_table,word,0);
         if(to_node == NULL || to_node == from)
          return;

        representative2 = find_representative(to_node);
        if (representative1 == representative2){
          representative1->number_of_edges++;
        } else {
          if (representative1->number_of_vertices < representative2-
>number_of_vertices)
          {
            representative1->representative = representative2;
            representative2->number_of_vertices += representative1-
>number_of_vertices;
            representative2->number_of_edges += (representative1-
>number_of_edges)+1;
            representative1->number_of_edges = 0;
            representative1->number_of_vertices = 0;
          }
          else
          {
            representative2->representative = representative1;
            representative1->number_of_vertices += representative2-
>number_of_vertices;
            representative1->number_of_edges += (representative2-
>number_of_edges)+1;
            representative2->number_of_edges = 0;
            representative1->number_of_vertices = 0;
          }
        }

        link1 = allocate_adjacency_node();
        link2 = allocate_adjacency_node();

        if(link1 == NULL || link2 == NULL)
        {
          fprintf(stderr,"Out of memory\n");
          exit(1);
        }
         link1->vertex = to_node;
        link2->next = from->head;
```

```
    from->head = link1;

    link2->vertex = from;
    link2->next = to_node->head;
    to_node->head = link2;
     hash_table->number_of_edges++;
    return;
}
```

7. breadth-first-search

O algoritmo *Breadth First Search* (Busca em Largura) é um algoritmo de pesquisa usado para percorrer ou procurar num grafo ou árvore.

Começa pela raiz (ou num nó específico) e explora todos os nós adjacentes antes de explorar os nós mais distantes.

A busca é realizada em largura, ou seja, todos os nós de mesmo nível são visitados antes de serem visitados os nós dos níveis subsequentes.

A função *breadh_first_search*() implementa essa lógica para percorrer uma componente conectada de um grafo representado por uma *hash table*. Ela recebe como parâmetros o número máximo de vértices na componente conectada, um vetor de ponteiros para os vértices da componente conectada, um ponteiro para o vértice de origem da pesquisa e um ponteiro para o vértice alvo (opcional).

A função inicializa o vetor de vértices com o vértice de origem e marca-o como visitado.

Seguidamente, é criado um *loop while* que continua enquanto o número de vértices visitados for menor que o número máximo de vértices na componente conectada e o vértice alvo (se especificado) não tiver sido encontrado.

Dentro do *loop*, é criado outro *loop* que percorre a lista de adjacência do vértice atual. Para cada nó adjacente, se ele já tiver sido visitado, a iteração é saltada. Caso contrário, a função marca o nó como visitado e adiciona-o ao vetor de vértices da componente conectada.

A cada iteração, o índice do vetor de vértices é incrementado e a variável *n* é incrementada para contar o número de vértices visitados.

Quando o *loop while* termina, a função retorna *-1*, indicando que a pesquisa terminou sem encontrar o vértice alvo.

Em resumo, esta função implementa uma pesquisa em profundidade para percorrer uma componente conectada de um grafo representado por uma tabela de *hash* e armazena os vértices visitados num vetor.

```
      static int breadh_first_search(int
maximum_number_of_vertices,hash_table_node_t
*list_of_vertices[],hash_table_node_t *origin,hash_table_node_t *goal)
      {  //
       // complete this
       //
       list_of_vertices[0]=origin;
       origin->visited = 1;
       int n=0;

       int frente = 0;
       while(n<maximum_number_of_vertices && list_of_vertices[n] !=
goal) {
         /* hash_table_node_t node2 = list_of_vertices[n]; */
         for(adjacency_node_t *adjency_node = list_of_vertices[frente]-
>head; adjency_node != NULL; adjency_node = adjency_node->next ){
           if(adjency_node->vertex->visited==1){
             continue;
           }
           frente ++;
           adjency_node->vertex->visited=1;
           list_of_vertices[frente]=adjency_node->vertex;
           adjency_node->vertex->previous=list_of_vertices[frente];
         }
         n++;
       }
       return -1;
      }
```

8. list_connected_component

Esta função, *list_connected_component*(), é usada para listar todos os vértices da componente conectada de uma determinada palavra numa *hash table*. A função recebe dois parâmetros: um ponteiro para uma tabela de *hash* e uma palavra (*word*).

Primeiro, a função usa a função *find_word*() para encontrar o nó correspondente à palavra dada na tabela de *hash*. Se o nó não for encontrado, a função imprime uma mensagem de erro e retorna.

Em seguida, a função usa a função *find_representative*() para encontrar o nó representativo da componente conectada que contém o nó encontrado anteriormente.

A função aloca então um vetor de ponteiros para nós, chamado *list_of_vertices*, com o tamanho máximo de vértices da componente conectada. Depois, a função *breadh_first_search*() é chamada para preencher o vetor com os vértices da componente conectada.

Por fim, a função imprime cada vértice do vetor e liberta a memória alocada para o vetor antes de retornar.

Em resumo, esta função procura um vértice na *hash table* a partir de uma palavra específica e, de seguida, encontra o representante da componente conectada que contém o vértice. Aloca, ainda, um vetor de vértices da componente conectada e utiliza uma pesquisa em profundidade a partir do vértice encontrado. Por fim, a função imprime cada vértice do vetor e liberta a memória alocada para o vetor.

```
static void list_connected_component(hash_table_t *hash_table,const
char *word)
{
 //
 // complete this
 //
 hash_table_node_t *find_node, *rep_node;
 int n, max_vertices_number;
 hash_table_node_t **list_of_vertices;

 find_node = find_word(hash_table,word,0);
  if(find_node == NULL)
 {
   printf("list_connected_component: word not found\n");
   return;
 }
  rep_node = find_representative(find_node);
 max_vertices_number = rep_node->number_of_vertices;
 list_of_vertices = (hash_table_node_t **)malloc(max_vertices_number *
sizeof(hash_table_node_t *));
  if(list_of_vertices == NULL)
 {
   fprintf(stderr,"list_connected_component: out of memory\n");
   exit(1);
 }
```

```
 n =
breadh_first_search(max_vertices_number,list_of_vertices,find_node,NULL
);
  for(int i = 0;i < n;i++){
   printf("%d: %s\n",list_of_vertices[i]->word);
 }
 free(list_of_vertices);
}
```

9. path_finder

A função *path_finder* é usada para encontrar um caminho entre duas palavras dentro de um dicionário representado por uma *hash table*. Recebe como parâmetros a *hash table* e as palavras *from_word* e *to_word* para as quais deve ser encontrado o caminho.

Primeiro, a função usa a função *find_word* para procurar por ambas as palavras na tabela *hash* e verifica se elas existem no dicionário. Se uma ou ambas não forem encontradas, a função imprime uma mensagem de erro e retorna.

A seguir, a função usa a função *find_representative* para encontrar os representantes (ou raízes) das palavras na tabela *hash*. Se esses representantes não forem iguais, significa que as palavras não estão no mesmo componente conectado e, portanto, não há caminho entre elas. A função imprime uma mensagem de erro e retorna nesse caso.

Se as palavras estiverem no mesmo componente conectado, a função aloca espaço para uma lista de vértices do tamanho do número de vértices no componente conectado e usa a função *breadh_first_search* para preencher essa lista com todos os vértices no componente conectado e encontrar o índice final no qual a palavra *from_word* é encontrada na lista.

Finalizando, a função itera ao longo da lista começando do índice final e imprimindo cada palavra e o seu índice até chegar ao começo da lista.

Por fim, liberta o espaço alocado para a lista de vértices.

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

```
 //
 hash_table_node_t *nodeinit, *repinit, *nodefin, *repfin;
 nodeinit = find_word(hash_table, from_word, 0);
 nodefin = find_word(hash_table, to_word, 0);
  if(nodeinit == NULL || nodefin == NULL){
   printf("Word(s) not in dictionary\n");
   return;
 }

 repinit = find_representative(nodeinit);
 repfin = find_representative(nodefin);

 if(repinit != repfin){
   printf("Words not in the same connected component (No path between
them)\n");
   return;
 }

 hash_table_node_t **list_of_vertices = malloc(sizeof(hash_table_node_t
*) * repinit->number_of_vertices);

 if (list_of_vertices == NULL)
 {
   fprintf(stderr, "path_finder: malloc failed\n");
   exit(1);
 }

 int  final_index  =  breadh_first_search(repinit->number_of_vertices,
list_of_vertices, nodefin, nodeinit);
 int final_index2=final_index-1; //array começa na posição 0
 hash_table_node_t *pointer = list_of_vertices[final_index2];
  while (pointer != NULL)
 {
   printf("%d: %s \n", pointer->word);
   pointer = pointer->previous;
 }
  free(list_of_vertices);
}
```

Conclusão

Após algumas dificuldades, conseguimos colocar algumas de todas as funções presentes no ficheiro *word_ladder.c* funcionais. Este projeto, promoveu a nossa capacidade de organização entre dupla, a nossa capacidade de resolução de problemas e a de entendimento código e seguir as suas ideias embutidas.

Aprendemos a lidar com vários problemas de alocação de memória e incentivou-nos a pesquisar sobre alguns algoritmos existentes como o caso de algoritmos de pesquisa.

Com este trabalho, temos agora outras bases que serão aplicadas no futuro, na resolução de outros problemas, bem como nos ensinou a perceber melhor como pesquisar informações e entender as documentações da linguagem C.

Bibliografia

-Slides e Material Teórico de AED
-https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/
-https://www.programiz.com/dsa/graph-bfs
-https://www.digitalocean.com/community/tutorials/hash-table-in-c-plus-plus
-https://www.geeksforgeeks.org/breadth-first-search-without-using-queue/
-https://www.pngwing.com/pt

Anexo

Script word_ladder.c (a utilizar com wordlist.txt)

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

// static configuration

#define _max_word_size_  32

// data structures

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 - keys
 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 -> if the representatives of
their connected componments are the same, they are connected
 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
};

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

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

 node = (adjacency_node_t *)malloc(sizeof(adjacency_node_t));
 if(node == NULL)
 {
```

```
    fprintf(stderr,"allocate_adjacency_node: out of memory\n");
    exit(1);
  }
  return node;
}

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

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

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

static void free_hash_table_node(hash_table_node_t *node)
{
  free(node);
}

//
// hash table stuff (mostly to be done)
//
static void print_table(hash_table_t *hash_table){

  hash_table->hash_table_size = 103;
  hash_table->number_of_entries = 0;
  hash_table->number_of_edges = 0;

  printf("Starting hash table\n");
  for (int i = 0; i < hash_table->hash_table_size; i++){
    if ( hash_table->heads[i] == NULL){
      printf("\t%i\t---\n", i);
    } else {
```

```
      printf("\t%i\t%s\n", i, hash_table->heads[i]->word);
    }
  }
}

unsigned int crc32(const char *str)
{
 static unsigned int table[256];
 unsigned int crc;
 if(table[1] == 0u) // do we need to initialize the table[] array?
 {
   unsigned int i,j;

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

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

 hash_table = (hash_table_t *)malloc(sizeof(hash_table_t)); // malloc
is used to dynamically allocate a single
 // large block of memory with the specified size
 if(hash_table == NULL)
 {
   fprintf(stderr,"create_hash_table: out of memory\n");
   exit(1);
 }
 //
 // complete this
 //
```

```
 hash_table->hash_table_size = 103;
 hash_table->number_of_entries = 0;
 hash_table->number_of_edges = 0;

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

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

static void hash_table_grow(hash_table_t *hash_table) {
 hash_table_node_t **old_hash_table, *head_one, *next;
 unsigned int old_size, i, index;
 //
 // complete this
 //
 old_hash_table = hash_table->heads;
 old_size = hash_table->hash_table_size;

 hash_table->hash_table_size = hash_table->hash_table_size*2+1;
 //double the size of the hash table

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

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

 // check for out of memory
 for (i = 0; i < hash_table->hash_table_size; i++) {
```

```
    hash_table->heads[i] = NULL;
  }

  for (i = 0; i < old_size; i++){
    //
    // complete this
    //
    head_one = old_hash_table[i];
    while(head_one != NULL){
      //
      // complete this
      //
      index = crc32(head_one->word) % hash_table->hash_table_size;
      next = head_one->next;
      // ...
      head_one->next = hash_table->heads[index];
      hash_table->heads[index] = head_one;
      head_one = next;
   }
  }
  free(old_hash_table);
}

static void hash_table_free(hash_table_t *hash_table)
{
  hash_table_node_t *head_one, *next;
  unsigned int i;
  //
  // complete this
  //
  hash_table->hash_table_size = 103;
  hash_table->number_of_entries = 0;
  hash_table->number_of_edges = 0;

  for (i = 0; i < hash_table->hash_table_size; i++){
    //
    // complete this
    //
    head_one = hash_table->heads[i];
    while(head_one != NULL){
      adjacency_node_t *nodeadj = head_one->head, *nextAdjNode;
      while(nodeadj != NULL){
      // complete this
```

```
      nextAdjNode = nodeadj->next;
      free_adjacency_node(nodeadj);
      nodeadj=nextAdjNode;
      }
      next = head_one->next;
      free_hash_table_node(head_one);
      head_one = next;
   }
  }
  free(hash_table->heads);
  free(hash_table);
}

// find a word in the 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 *node1;
unsigned int index;
index = crc32(word) % hash_table->hash_table_size;
node1 = hash_table->heads[index];
while(node1 != NULL)
{
  if(strcmp(node1->word,word) == 0)
    return node1;
  node1 = node1->next;
}
if(insert_if_not_found && strlen(word) < _max_word_size_)
{
  node1 = allocate_hash_table_node();
  strncpy(node1->word,word,_max_word_size_);
  node1->representative = node1;
  node1->next = hash_table->heads[index];
  node1->previous = NULL;
  node1->number_of_edges = 0;
  node1->number_of_vertices = 1;
  node1-> visited = 0;
  node1->head = NULL;
  hash_table->heads[index] = node1;
  hash_table->number_of_entries++;
  if(hash_table->number_of_entries > hash_table->hash_table_size)
    hash_table_grow(hash_table);
  return node1;
```

```
}
 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, *next_node, *temp_node;

   Representative = node;
   while (Representative != Representative->representative)
   {
       Representative = Representative->representative;
   }
   next_node = node;
   while (next_node != Representative)
   {
       temp_node = next_node->representative;
       next_node->representative = Representative;
       next_node = temp_node;
   }
   return Representative;
}

static void add_edge(hash_table_t *hash_table,hash_table_node_t
*from,const char *word)
{
 hash_table_node_t *to_node,*representative1,*representative2;
 adjacency_node_t *link1, *link2;

 representative1 = find_representative(from);
 to_node = find_word(hash_table,word,0);
  if(to_node == NULL || to_node == from)
   return;

 representative2 = find_representative(to_node);
 if (representative1 == representative2){
   representative1->number_of_edges++;
 } else {
```

```
   if (representative1->number_of_vertices < representative2-
>number_of_vertices)
   {
     representative1->representative = representative2;
     representative2->number_of_vertices += representative1-
>number_of_vertices;
     representative2->number_of_edges += (representative1-
>number_of_edges)+1;
     representative1->number_of_edges = 0;
     representative1->number_of_vertices = 0;
   }
   else
   {
     representative2->representative = representative1;
     representative1->number_of_vertices += representative2-
>number_of_vertices;
     representative1->number_of_edges += (representative2-
>number_of_edges)+1;
     representative2->number_of_edges = 0;
     representative1->number_of_vertices = 0;
   }
 }

 link1 = allocate_adjacency_node();
 link2 = allocate_adjacency_node();

 if(link1 == NULL || link2 == NULL)
 {
   fprintf(stderr,"Out of memory\n");
   exit(1);
 }
  link1->vertex = to_node;
 link2->next = from->head;
 from->head = link1;

 link2->vertex = from;
 link2->next = to_node->head;
 to_node->head = link2;
  hash_table->number_of_edges++;
 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 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,0
xFC, // 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)
{  //
 // complete this
 //
 list_of_vertices[0]=origin;
 origin->visited = 1;
 int n=0;

 int frente = 0;
 while(n<maximum_number_of_vertices && list_of_vertices[n] != goal) {
   for(adjacency_node_t *adjency_node = list_of_vertices[frente]->head;
adjency_node != NULL; adjency_node = adjency_node->next ){
     if(adjency_node->vertex->visited==1){
       continue;
     }
     frente ++;
     adjency_node->vertex->visited=1;
     list_of_vertices[frente]=adjency_node->vertex;
     adjency_node->vertex->previous=list_of_vertices[frente];
   }
   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)
{
 //
 // complete this
 //
 hash_table_node_t *find_node, *rep_node;
 int n, max_vertices_number;
 hash_table_node_t **list_of_vertices;

 find_node = find_word(hash_table,word,0);
  if(find_node == NULL)
 {
   printf("list_connected_component: word not found\n");
   return;
 }
  rep_node = find_representative(find_node);
 max_vertices_number = rep_node->number_of_vertices;
 list_of_vertices = (hash_table_node_t **)malloc(max_vertices_number *
sizeof(hash_table_node_t *));
  if(list_of_vertices == NULL)
 {
   fprintf(stderr,"list_connected_component: out of memory\n");
   exit(1);
 }

 n =
breadh_first_search(max_vertices_number,list_of_vertices,find_node,NULL
);
  for(int i = 0;i < n;i++){
   printf("%d: %s\n",list_of_vertices[i]->word);
 }
 free(list_of_vertices);
}

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

```
static int largest_diameter;
static hash_table_node_t **largest_diameter_example;

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

static void path_finder(hash_table_t *hash_table,const char
*from_word,const char *to_word)
{
 //
 // complete this
 //
 hash_table_node_t *nodeinit, *repinit, *nodefin, *repfin;
 nodeinit = find_word(hash_table, from_word, 0);
 nodefin = find_word(hash_table, to_word, 0);
  if(nodeinit == NULL || nodefin == NULL){
   printf("Word(s) not in dictionary\n");
   return;
 }

 repinit = find_representative(nodeinit);
 repfin = find_representative(nodefin);

 if(repinit != repfin){
   printf("Words not in the same connected component (No path between
them)\n");
   return;
 }

 hash_table_node_t **list_of_vertices = malloc(sizeof(hash_table_node_t
*) * repinit->number_of_vertices);

 if (list_of_vertices == NULL)
 {
   fprintf(stderr, "path_finder: malloc failed\n");
   exit(1);
 }

 int final_index = breadh_first_search(repinit->number_of_vertices,
list_of_vertices, nodefin, nodeinit);
```

```
 int final_index2=final_index-1; //array começa na posição 0
 hash_table_node_t *pointer = list_of_vertices[final_index2];
  while (pointer != NULL)
 {
   printf("%d: %s \n", pointer->word);
   pointer = pointer->previous;
 }
  free(list_of_vertices);
}

//
// main program
//

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

 // initialize hash table
 hash_table = hash_table_create();
 // read words
 fp = fopen((argc < 2) ? "wordlist-big-latest.txt" : argv[1],"rb");
 if(fp == NULL)
 {
   fprintf(stderr,"main: unable to open the words file\n");
   exit(1);
 }
 while(fscanf(fp,"%99s",word) == 1)
   (void)find_word(hash_table,word,1);

 fclose(fp);
 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);
 // ask what to do
 for(;;)
 {
   fprintf(stderr,"Your wish is my command:\n");
```

```
    fprintf(stderr,"  1 WORD       (list the connected component WORD
belongs to)\n");
    fprintf(stderr,"  2 FROM TO    (list the shortest path from FROM to
TO)\n");
    fprintf(stderr,"  3            (terminate)\n");
    fprintf(stderr,"> ");
    if(scanf("%99s",word) != 1)
      break;
    command = atoi(word);
    if(command == 1)
    {
      if(scanf("%99s",word) != 1)
        break;

      list_connected_component(hash_table,word);
    }
    else if(command == 2)
    {
      if(scanf("%99s",from) != 1)
        break;
      if(scanf("%99s",to) != 1)
        break;
      path_finder(hash_table,from,to);
    }
    else if(command == 3)
      break;
  }
  // clean up
  hash_table_free(hash_table);
  return 0;
}
```