Algoritmos e Estrutura de Dados - Word Ladder

Universidade de Aveiro

HashTables, Words, LinkedList, UndirectedGraph

David Palricas, Eduardo Alves, Inês Santos

Algoritmos e Estrutura de Dados - Word Ladder

Universidade de Aveiro

(108780) David Palricas
davidpalricas@ua.pt

(104179) Eduardo Alves
eduardoalves@ua.pt

(103477) Inês Santos
ines.santos20@ua.pt

14 de janeiro de 2023

Índice

1 Contribuições de cada elemento **1**

2 Introdução **2**

3 Pequena descrição das funções desenvolvidas **4**
3.1 Hash Table 4
3.2 Implementação do Grafo 10
3.3 Implementação do algoritmo breadh-first-search 12

4 Resultados **17**

5 Conclusão **26**

6 Apêndice **27**

7 Bibliografia **43**

Capítulo 1

Contribuições de cada elemento

Percentagem	das	contribuições
David Palricas	Eduardo Alves	Inês Santos
33.33%	33.33%	33.33%

Todos os membros do grupo contribuiram igualmente para todas as componentes desta avaliação.

Capítulo 2

Introdução

Este segundo trabalho "Word Ladder"descreve-nos um problema cujo desafio se encontra em transformar uma palavra noutra, substituindo apenas uma letra de cada vez no minimo de passos possiveis.

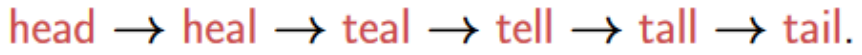


Figura 2.1: Exemplo em inglês de um Word Ladder

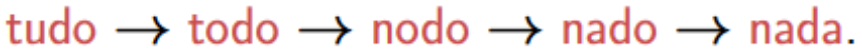


Figura 2.2: Exemplo em português de um Word Ladder

Para tal, é nos solicitado a implementação de uma "Hash Table", também designada por tabela de dispersão. Sendo necessário completar para este processo, as seguintes funções:

- create (Criação da tabela de disperção)
- grow (Permite o redimensionamento da tabela de dispersão)
- free (Esvazia a tabela de dispersão)
- find_Word(Procura por uma palavra na tabela)

Em seguinda teremos que concluir a implementação de um grafo através das funções:

- find_representative
- add_Edge (Permite adicionar vértices ao gráfico)
- similar_words

Após esta implementação, prosseguimos com a utilização de um algoritmo "(Breadth-first search)", também designado por busca em largura. Implementámos então a função:

- breadh_first_search

Após isto encontramos a lista de palavras pertencentes a um componente conexo e o caminho mais curto entre palavras através das funções:

- list_connected_component
- path_finder

Para concluir, entregaram-nos também a tarefa opcional de computar o diâmetro de uma componente conectada e listar a corrente de palavras mais longa, tal como capturar algumas estatísticas sobre o gráfico e testar por perdas de memória. Para isto usariamos algumas funções tais com:

- connected_component_diameter
- graph_info

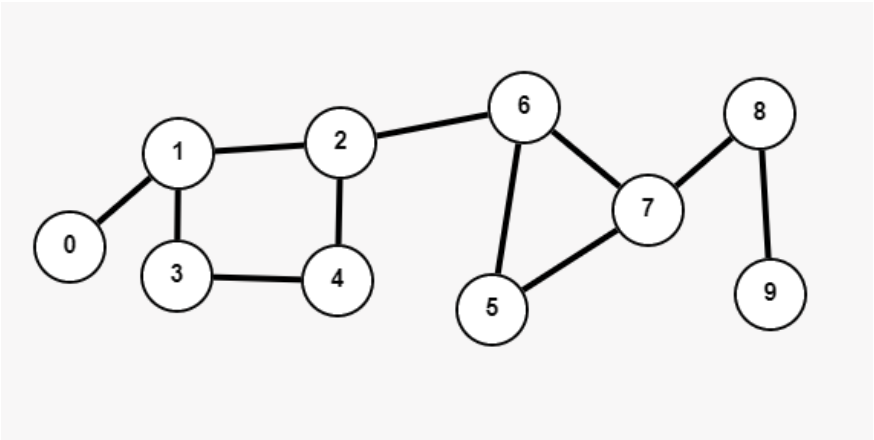


Figura 2.3: Exemplo de um Grafo para contexto

Capítulo 3

Pequena descrição das funções desenvolvidas

3.1 Hash Table

No começo do projeto é necessária a implementação de uma hashtable através das funções create, grow, free e find_word como já mencionado anteriormente. Na implementação da função create temos que atribuir várias variáveis entre as quais:

1. hash_table_size (Tamanho á qual decidimos que seria melhor o valor de 101)
2. number_of_entries (Número de entradas na tabela á qual atribuimos 0 inicialmente)
3. number_of_edges (Número de arestas, que servem apenas para efeitos de informação)
4. hash_table-> heads (Foi necessária alocação de memória consoante o hash_table_size através do uso de um malloc)

Por fim, nesta função criámos um ciclo for que atribui o valor Nulo a todos os valores da tabela. Como podemos através da seguinte imagem:

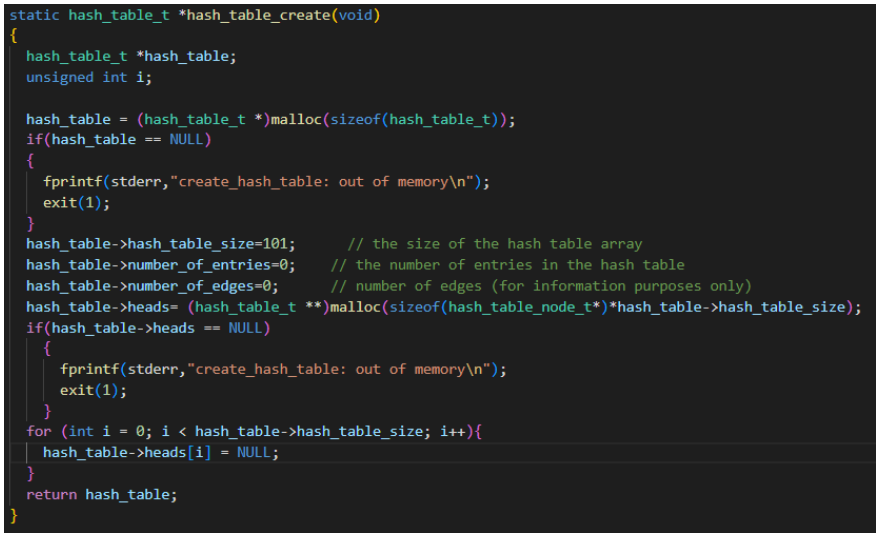


Figura 3.1: Função Create

Após finalizarmos a função create deparámo-nos com a função grow, esta função procura redimensionar a tabela de dispersão que inicialmente construimos de forma a impedir que esta fique sem espaço. Para este efeito, achamos melhor desenvolver a função de forma a esta construir uma nova hash_table com um tamanho novo ao qual designamos "newsize". O newsize corresponde ao dobro + 1 do tamanho da tabela original (hash_table_size).

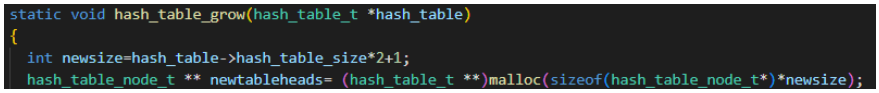


Figura 3.2: Criação de newsize

Com a nova tabela de dispersão com pouco mais que o dobro do espaço, era necessário transferir a informação da tabela ineficiente antiga para a nova. Para isto úsamos dois ciclos (um for e um while). O ciclo for que percorre a hashtable antiga e guarda os valores numa variável temporária "tmp"e o ciclo while que se encontra dentro do próprio ciclo for que transfere a informação para a nova tabela (Enquanto a variável temporária não for nula).

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


 while(tmp != NULL){
     hash_table_node_t* next = tmp->next;
     int index = crc32(tmp->word) % newsize;
     tmp->next = newtableheads[index];
     newtableheads[index] = tmp;
     tmp = next;
   }
 }
```

Para finalizar esta função, usamos a função free() para libertar a memória previamente alocada do array hash_table_heads e ajustamos os nomes para puder ser invocada a função grow outra vez, no caso de ser necessária uma tabela ainda maior.

Na terceira função presente no projeto encontramos a função hash_table_free, o objetivo dela é libertar toda a informação da tabela de dispersão. No entanto, para libertar toda a tabela, não se pode simplesmente eliminar o array hash_table->heads pois este contêm dentro dele blocos que continuariam com informação a ocupar espaço, por isso também será necessário libertar essa informação.

No entanto, dentro destes blocos encontram-se mais um campo que também tem informação a ser libertada (head), contêm informação vertex e next. Então para libertar na sua totalidade a tabela de dispersão é necessário libertar todas estas etapas. Para isto, começámos pelo uso de um ciclo for de forma a percorrer todas as entradas da hash_table, de seguida usamos um while loop para libertar a informação em cada entrada da hash_table (enquanto o Node não for Nulo, vai libertando a sua informação), de seguida fizemos o mesmo para os nós adjacentes, ou seja, a linked list que se encontra dentro dos blocos, garantindo assim a libertação de toda a tabela de disperção. Como podemos analisar na imagem seguinte

```c
static void hash_table_free(hash_table_t *hash_table){
  hash_table_node_t *node, *next_node;
  adjacency_node_t *node_adj, *next_node_adj;

  // Percorre as entradas na Hash_Table.
  for(unsigned int i = 0; i < hash_table->hash_table_size; i++){
    node = hash_table->heads[i];
  // Liberta a memória alocada em cada bloco
    while(node != NULL){
      next_node = node->next;
      node_adj = node->head;
       while (node_adj != NULL){
        next_node_adj = node_adj->next;
        free_adjacency_node(node_adj);
        node_adj = next_node_adj;
       }
      free_hash_table_node(node);
      node = next_node;
      // O mesmo que para os nós mas para os adjacentes

    }
  }

  // free the memory allocated for the heads array
  free(hash_table->heads);

  // free the memory allocated for the hash table
  free(hash_table);
}
```

Figura 3.3: Função Free

Por último na implementação da tabela temos a função find_words, cujo objetivo é percorrer a hash_table até encontrar a palavra pretendida. Para este processo o programa, começa por percorrer a lista ligada na posição do array á procura da palavra, compara as palavras de cada Node á desejada através da utilização da função já existente na linguagem c, strcmp(), função esta que compara duas strings caractér a caractér e retorna o resultado 0 se as palavras forem iguais.

Se a palavra for encontrada em algum dos Nodes, o programa dá break do loop, senão continua a percorrê-lo até encontrar a palavra. Se a palavra não for encontrada até ao final da hash_Table é nos atribuida a tarefa de a inserir na hash_table, para obter este resultado começamos por alocar espaço na memória na hash_table e atribuimos valores "default"a um novo Node e após isso copiamos a string da palavra desejada para dentro do Node através do uso da função strcpy.

```
node found =!strcmp(node->word,word);
```

(a) Utilização da função strcmp

```
strcpy(node->word, word);
```

(b) Utilização da função strcopy

No caso de não haver espaço ao inserir o novo Node, decidimos utilizar a função grow() implementada previamente de forma a prevenir esse possível erro.

Na imagem seguinte está demonstrada a função na sua totalidade:

```c
static hash_table_node_t *find_word(hash_table_t *hash_table,const char *word,int insert_if_not_found)
{
  // Se a HashTable for demasiado pequena, ajusta o tamanho com a função Grow.
  if(hash_table->hash_table_size < hash_table->number_of_entries){
    hash_table_grow(hash_table);
  }
  unsigned i = crc32(word) % hash_table->hash_table_size;
  bool node_found = false;
  hash_table_node_t *node = NULL;

  //Percorrer a lista ligada na posição do array à procura da palavra
  node = hash_table->heads[i];
  while(!node_found && node != NULL){
  // Compara as palavras entre o Node e a desejada e analisa se são iguais.
  node_found =!strcmp(node->word,word);
  if(node_found){
    break;
  }
  node = node->next;
  }

  if (!node_found && insert_if_not_found){

    // Completa os valores no novo Node.
    node = allocate_hash_table_node();
    node->previous = NULL;
    node->next = NULL;
    node->head = NULL;
    node->number_of_vertices = 1;
    node->number_of_edges = 0;
    node->visited = 0;
    node->representative = node;
    strcpy(node->word, word);

    // Insere o novo Nó no inicio da lista ligada.
    hash_table_node_t *next_node = hash_table->heads[i];
    hash_table->heads[i] = node;
    node->next = next_node;

    hash_table->number_of_entries + 1;
  }

  return node;
}
```

3.2 Implementação do Grafo

Na implementação do grafo temos alguns conceitos importantes a esclarecer antes de começarmos a pensar em funções. O que são representantes? Como se decide qual é o representante quando dois grafos se juntam? Primeiro, começemos por esclarecer essas questões. Mas primeiro, aqui se encontram algumas variáveis importantes dos grafos:

1. representative
2. arestas
3. vértices

Um representante é um vértice que permite saber se dois vértices estão conectados ou não, se um vértice A está conectado ao vértice representativo C e pretendemos saber se um vertice B também conectado ao vértice representativo C está connectado também a A, então a resposta é afirmativa. Vértices com o mesmo representante pertencem então ao mesmo componente conexo.

E se dois componentes conexos se pretenderem juntar, como se define o representante? Neste caso o representante do novo componente conexo (junção dos dois anteriores) passa a ser o do representante do componente com maior número de vértices (Isto ser-nos á útil na função add_edges).

Com o aumento do tamanho dos componentes conexos, o representante final vai mudando constantemente. O objetivo da primeira função "find_representative"que desenvolvemos é encontrar esse representante final. Para isso, usámos as variáveis que nos forneceram: *representative, *Node e *Next_Node. Decidimos primeiro usar um ciclo while para encontrar o representante final e em seguida utilizar outro ciclo while para colocar todos os representantes ao longo do caminho a apontar diretamente para o representante final (tornando bastante mais eficiente o processo nas futuras chamadas à função).

```
while(next_node != NULL){
  if(next_node == next_node->representative){
   representative = next_node;
   break;
  }else{
    next_node = next_node->representative;
  }
```

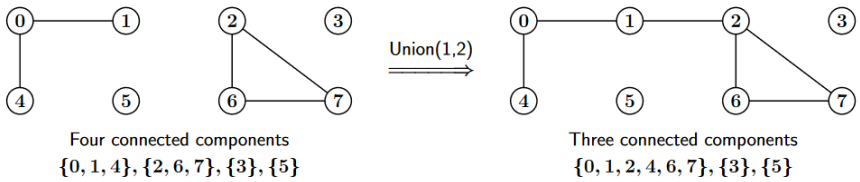
(a) Procura pelo resultante final

```
  while (node != NULL && node != representative){
     next_node=node->representative;
     node->representative = representative;
     node=next_node;
  }
 return representative;
 }
```

(b) Coloca os Nodes a apontar para o representante final diretamente

A segunda função da implementação dos grafos, que se designa add_edges permite adicionar uma aresta entre dois vértices, para isto usámos a estrutura Union-Find Data.

Um exemplo disto encontra-se na imagem seguinte em que se adiciona uma aresta entre o vértice 1 e 2.



Nesta segunda função, os valores de entrada fornecidos são a hash_table, um node from e uma constante word. A partir deste node criámos uma ligação através do uso das variáveis link e new_link que criámos, esta ligação é criada do valor "from"que no caso da imagem acima seria o valor 1 para um valor "to"que na imagem acima seria o 2 através da variável Link. No entanto é também criada uma ligação do "to"para o "from"ao qual úsamos a variável new_link. Mas antes disso é necessário verificar se a palavra para onde pretendemos ir realmente existe, sendo necessário o uso da função find_word no "to".

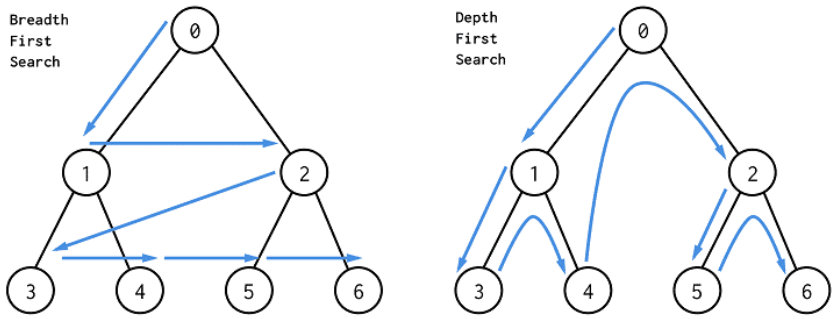
No fim desta função, utilizamos a função find_representative do "to"e do "from", de forma a se obter acesso ao seu representanto e ao número de vértices de cada um. Desta forma, permite uma comparação entre números de vértices. O que tiver um maior número de vértices permanece como representante do componente conexo novo e maior.

É também efetuada a soma de ambos os números de vértices dos componentes conexos antigos, atualizando assim no novo. De seguida é demonstrada a função completa:

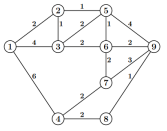
```c
static void add_edge(hash_table_t *hash_table,hash_table_node_t *from,const char *word)
{
  hash_table_node_t *to,*from_representative,*to_representative;
  adjacency_node_t *link, *new_link;

  to = find_word(hash_table,word,0);
  if(to == NULL){
    return;
  }
  //Do from para o to.
  link = allocate_adjacency_node(); //Cria um novo link
  link -> vertex = to; //do outro lado está o to.
  link -> next = from ->head; //Insere o na lista
  from->head = link;
  // Do to para o from.
  new_link = allocate_adjacency_node(); //Cria um novo link
  new_link -> vertex = from; //do outro lado está o from.
  new_link -> next = to ->head; //Insere o na lista
  to->head = new_link;

  //Union Find Data
  from_representative = find_representative(from);
  to_representative = find_representative(to);

  if(from_representative == to_representative){
    return;
  }else{
    //O representante do componente mais pequeno tomar o valor do representante maior e atualizar o respetivo number of vértices.
    if(from_representative-> number_of_vertices > to_representative -> number_of_vertices){
      to_representative->representative = from_representative;
      from_representative->number_of_vertices = from_representative->number_of_vertices + to_representative->number_of_vertices;
    }else{
      from_representative->representative = to_representative;
      to_representative->number_of_vertices = to_representative->number_of_vertices + from_representative->number_of_vertices;
    }
  }
}
```

3.3 Implementação do algoritmo breadh-first-search

O algoritmo breadth-first-search ou busca em largura realiza uma busca num grafo progredindo por todas as arestas e vértices do grafo. O algoritmo garante que nenhum vértice ou aresta será visitado mais do que uma vez. Para demonstrar como este algoritmo funciona escolhemos a imagem seguinte que compara o algoritmo em largura que usámos com um de busca em profundidade: O gráfico que encontramos na imagem à esquerda é o gráfico que explora o mesmo algoritmo que vamos explorar na nossa função.

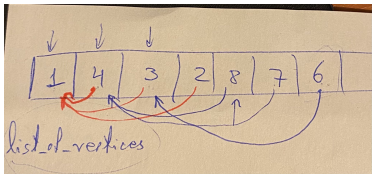


No começo da função criámos duas variáveis integers, uma out_index e uma in_index. Ambas estas variáveis irão percorrer a list_of_vertices que vai agir como um array onde vamos guardando os dados dos vértices. A função começa na origem e vai marcando os vértices por onde vai passando como visitados ou não. O in_index vai acrescentando ao array list_of_vertices os vértices adjacentes á medida que vai percorrendo o grafo, enquanto o out_index vai percorrendo entre os vértices guardados pelo in_index e procurar pelos vértices adjacentes não visitados pela ordem de entrada dos in_index. Isto enquanto out_index $<$ in_index, out_index $<$ número máximo de vértices e que o destino (goal) ainda não tenha sido atingido. No final da função breadth-first-search voltámos a colocar todos os nodes como não visitados para na próxima pesquisa se voltar a poder marcar as visitas.

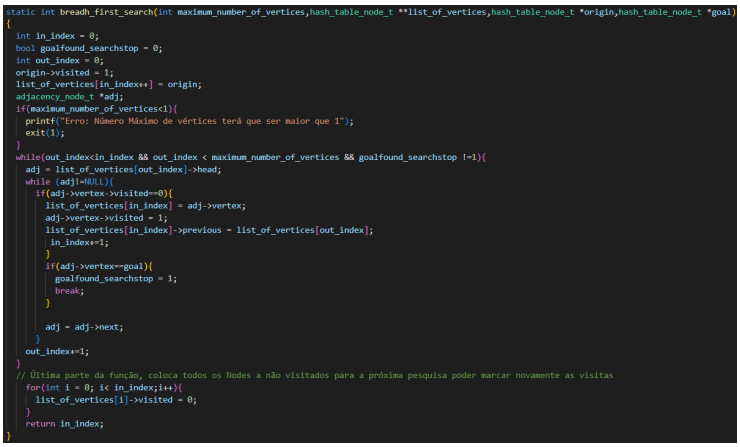


1	$(4,6) \to (3,4) \to (2,2)$
2	$(5,1) \to (3,1) \to (1,2)$
3	$(6,2) \to (5,2) \to (2,1) \to (1,4)$
4	$(8,2) \to (7,2) \to (1,6)$
5	$(9,4) \to (6,1) \to (3,2) \to (2,1)$
6	$(9,2) \to (7,2) \to (5,1) \to (3,2)$
7	$(9,3) \to (6,2) \to (4,2)$
8	$(9,1) \to (4,2)$
9	$(8,1) \to (7,3) \to (6,2) \to (5,4)$

(a) Lista de Adjacência



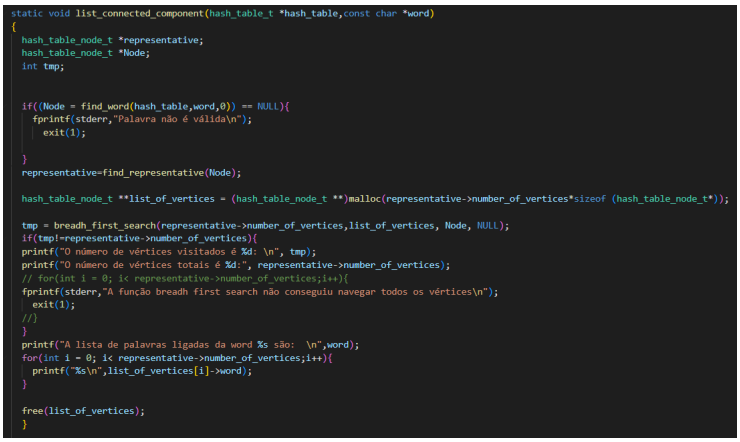
(b) Relação entre a lista de vértices, o in_index e o out_index



```
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)
{
  int in_index = 0;
  bool goalfound_searchstop = 0;
  int out_index = 0;
  origin->visited = 1;
  list_of_vertices[in_index++] = origin;
  adjacency_node_t *adj;
  if(maximum_number_of_vertices<1){
    printf("Erro: Número Máximo de vértices terá que ser maior que 1");
    exit(1);
  }
  while(out_index<in_index && out_index < maximum_number_of_vertices && goalfound_searchstop !=1){
    adj = list_of_vertices[out_index]->head;
    while (adj!=NULL){
      if(adj->vertex->visited==0){
        list_of_vertices[in_index] = adj->vertex;
        adj->vertex->visited = 1;
        list_of_vertices[in_index]->previous = list_of_vertices[out_index];
         in_index+=1;
        }
        if(adj->vertex==goal){
          goalfound_searchstop = 1;
          break;
        }

        adj = adj->next;
      }
    out_index+=1;
  }
  // Última parte da função, coloca todos os Nodes a não visitados para a próxima pesquisa poder marcar novamente as visitas
    for(int i = 0; i< in_index;i++){
      list_of_vertices[i]->visited = 0;
    }
    return in_index;
}
```

Figura 3.7: Imagem da função desenvolvida

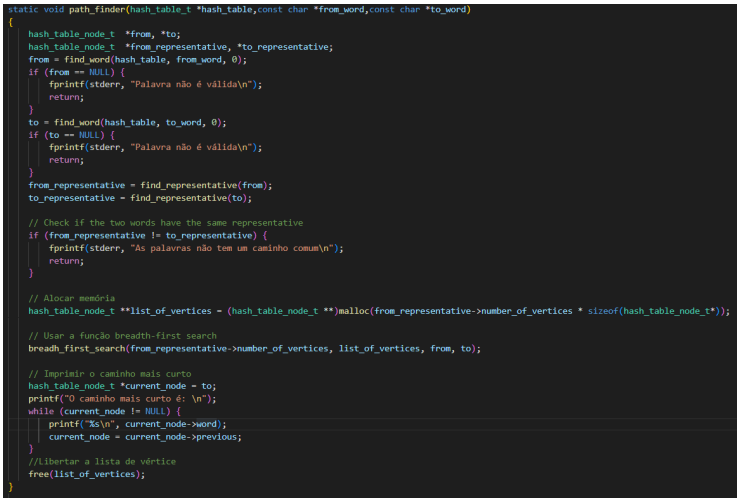
De seguida encontrámos a função list_connected_components cujo objetivo é listar todos os vértices pertencentes a um componente conexo. Para desenvolver esta função, necessitámos de usar a função find_word para saber se a palavra do componente conexo que queremos analisar é valida ou inválida(se for inválida envia uma mensagem de erro). Após isso, é necessário identificar o representativo utilizando a função find_representative(), alocar espaço de memória e iterar usando um for o conteúdo da "list_of_vertices". No fim da função foi necessário libertar a memória alocada previamente.
A imagem seguinte demonstra a função criada:



```c
static void list_connected_component(hash_table_t *hash_table,const char *word)
{
  hash_table_node_t *representative;
  hash_table_node_t *Node;
  int tmp;


  if((Node = find_word(hash_table,word,0)) == NULL){
    fprintf(stderr,"Palavra não é válida\n");
      exit(1);

  }
  representative=find_representative(Node);

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

  tmp = breadh_first_search(representative->number_of_vertices,list_of_vertices, Node, NULL);
  if(tmp!=representative->number_of_vertices){
  printf("O número de vértices visitados é %d: \n", tmp);
  printf("O número de vértices totais é %d:", representative->number_of_vertices);
  // for(int i = 0; i< representative->number_of_vertices;i++){
  fprintf(stderr,"A função breadh first search não conseguiu navegar todos os vértices\n");
    exit(1);
  //}
  }
  printf("A lista de palavras ligadas da word %s são:  \n",word);
  for(int i = 0; i< representative->number_of_vertices;i++){
    printf("%s\n",list_of_vertices[i]->word);
  }

  free(list_of_vertices);
  }
```

Figura 3.8: Imagem da função desenvolvida

Por fim, temos a função path_finder cujo objetivo é encontrar o caminho de palavras mais curto de uma palavra origem até uma palavra destino. Incluimos algumas condições que previnem possíveis erros, tais como a origem ou o destino serem palavras inválidas, ou as palavras não terem um caminho comum. Após isso, chamámos a função breadh-first-search e criámos um loop while de forma a imprimir as palavras dos nós saltando de previous em previous. Na página seguinte demonstramos a função desenvolvida:



```c
static void path_finder(hash_table_t *hash_table,const char *from_word,const char *to_word)
{
    hash_table_node_t  *from, *to;
    hash_table_node_t  *from_representative, *to_representative;
    from = find_word(hash_table, from_word, 0);
    if (from == NULL) {
        fprintf(stderr, "Palavra não é válida\n");
        return;
    }
    to = find_word(hash_table, to_word, 0);
    if (to == NULL) {
        fprintf(stderr, "Palavra não é válida\n");
        return;
    }
    from_representative = find_representative(from);
    to_representative = find_representative(to);

    // Check if the two words have the same representative
    if (from_representative != to_representative) {
        fprintf(stderr, "As palavras não tem um caminho comum\n");
        return;
    }

    // Alocar memória
    hash_table_node_t **list_of_vertices = (hash_table_node_t **)malloc(from_representative->number_of_vertices * sizeof(hash_table_node_t*));

    // Usar a função breadth-first search
    breadh_first_search(from_representative->number_of_vertices, list_of_vertices, from, to);

    // Imprimir o caminho mais curto
    hash_table_node_t *current_node = to;
    printf("O caminho mais curto é: \n");
    while (current_node != NULL) {
        printf("%s\n", current_node->word);
        current_node = current_node->previous;
    }
    //Libertar a lista de vértice
    free(list_of_vertices);
}
```

Figura 3.9: Imagem da função desenvolvida

Capítulo 4

Resultados

Após concluirmos as funções e concluirmos a etapa de debugging testámos algumas palavras para ambos os objetivos pedidos no projeto. Objetivos estes de listar todas as palavras pertencentes a um componente conexo (List Connected Component) e testar o caminho mais curto entre palavras á escolha. Para ambas estas tarefas foram-nos fornecidos alguns ficheiros de texto com palavras desde quatro letras (wordlist-four-letters) a uma lista com várias variadades de tamanhos (wordlist-big-latest-txt).

Primeiro vamos mostrar alguns resultados obtidos no terminal para a primeira tarefa (List Connected Components). É de notar que ainda faltam palavras em cada uma destas imagens mas não couberam na captura.

Para finalizar apresentamos os resultados do caminho mais curto (path_finder).Para esta última função escolhemos a combinação de palavras "vaca-sala", "pesa-vara", "golo-taco".

Incluimos também, o desafio proposto no código fornecido.

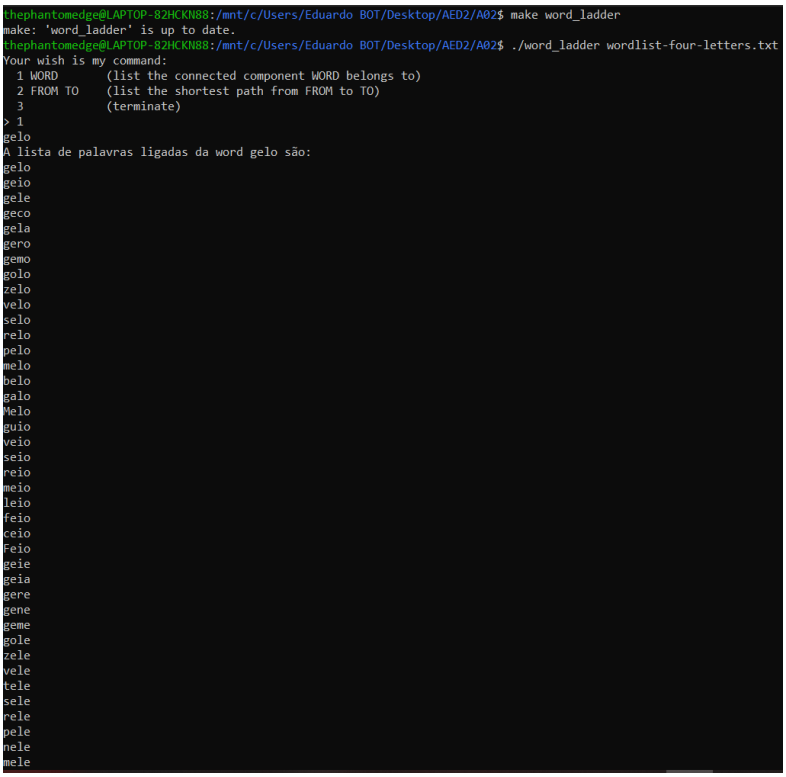


Figura 4.1: ListConnectedComponent (gelo)

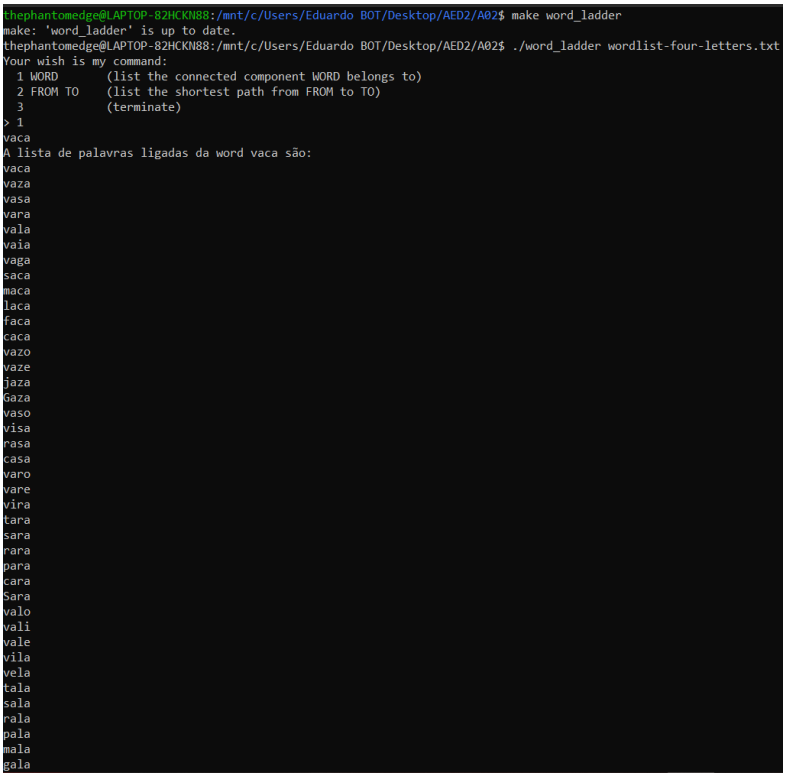


Figura 4.2: ListConnectedComponent (vaca)

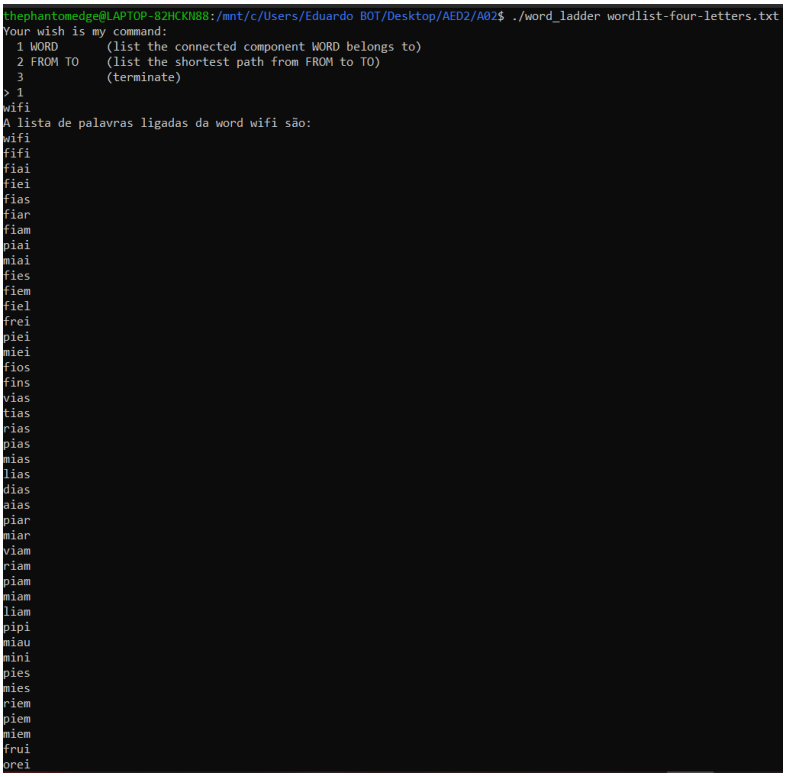


Figura 4.3: ListConnectedComponent (wifi)

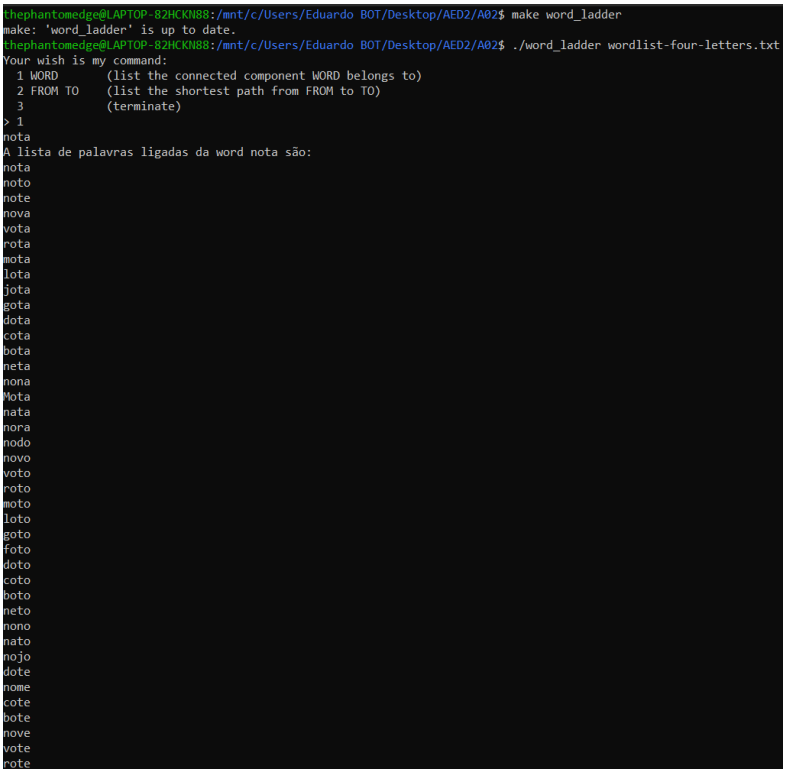


Figura 4.4: ListConnectedComponent (nota)

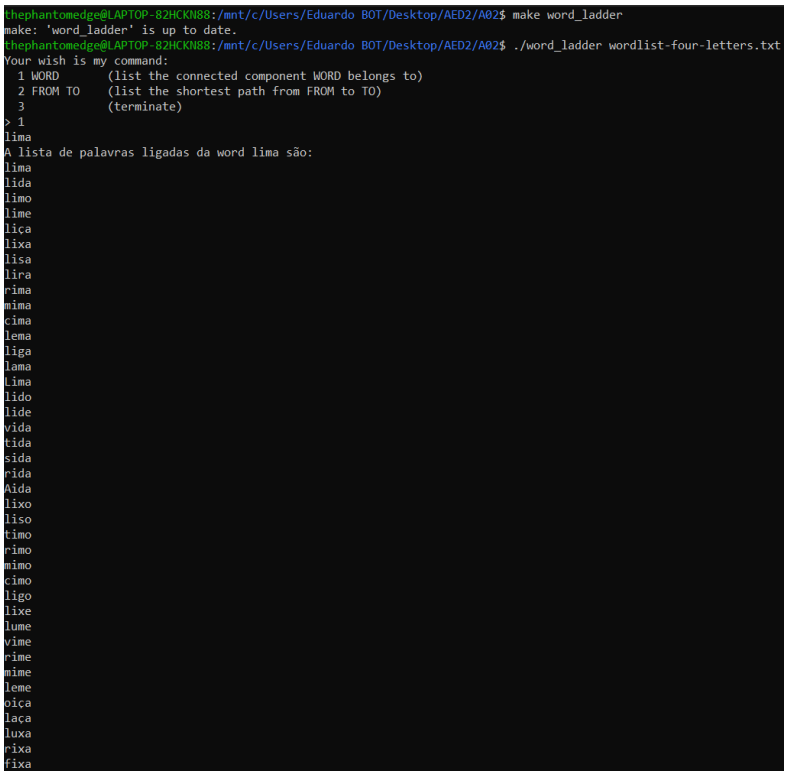


Figura 4.5: ListConnectedComponent (lima)

```
thephantomedge@LAPTOP-82HCKN88:/mnt/c/Users/Eduardo BOT/Desktop/AED2/A02$ make word_ladder
make: 'word_ladder' is up to date.
thephantomedge@LAPTOP-82HCKN88:/mnt/c/Users/Eduardo BOT/Desktop/AED2/A02$ ./word_ladder wordlist-four-letters.txt
Your wish is my command:
  1 WORD       (list the connected component WORD belongs to)
  2 FROM TO    (list the shortest path from FROM to TO)
  3            (terminate)
> 1
fogo
A lista de palavras ligadas da word fogo são:
fogo
foco
fodo
foge
fofo
foto
foro
vogo
togo
rogo
logo
jogo
figo
Togo
foca
toco
soco
loco
coco
fico
foda
todo
rodo
podo
nodo
modo
lodo
iodo
godo
fedo
fodi
fado
fode
fome
fole
fofa
mofo
voto
roto
noto
```

Figura 4.6: ListConnectedComponent (fogo)

```
thephantomedge@LAPTOP-82HCKN88:/mnt/c/Users/Eduardo BOT/Desktop/AED2/A02$ ./word_ladder wordlist-three-letters.txt
Your wish is my command:
  1 WORD       (list the connected component WORD belongs to)
  2 FROM TO    (list the shortest path from FROM to TO)
  3            (terminate)
> 2
mal
bem
O caminho mais curto é:
bem
bel
mel
mal
Your wish is my command:
  1 WORD       (list the connected component WORD belongs to)
  2 FROM TO    (list the shortest path from FROM to TO)
  3            (terminate)
>
```

Figura 4.7: Desafio: bem -> mal

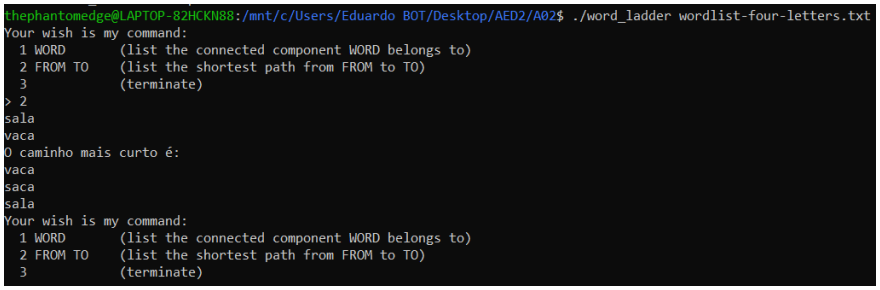


Figura 4.8: PathFinder: vaca-> sala

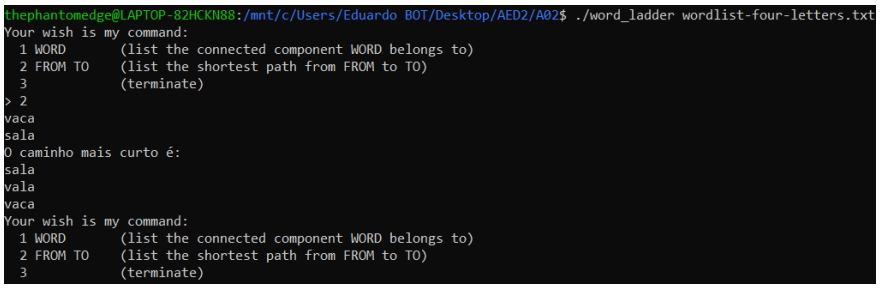


Figura 4.9: PathFinder: sala-> vaca

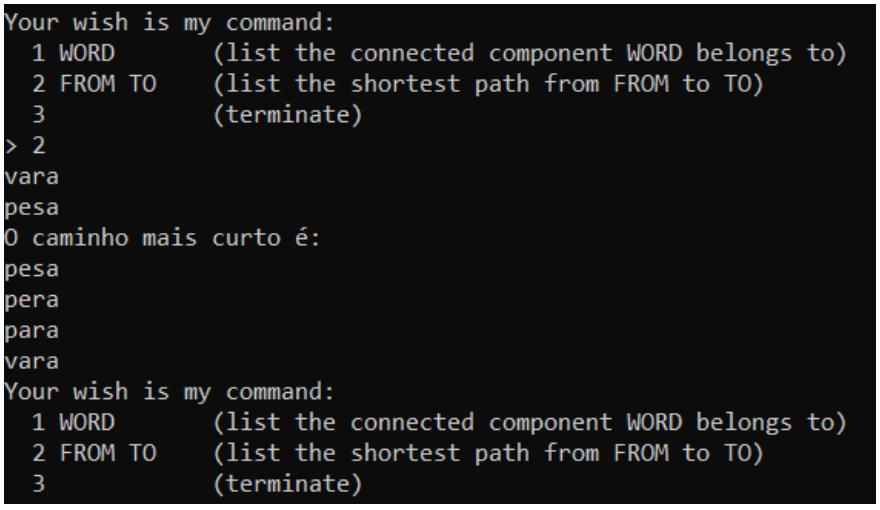


Figura 4.10: PathFinder: pesa -> vara

```
thephantomedge@LAPTOP-82HCKN88:/mnt/c/Users/Eduardo BOT/Desktop/AED2/A02$ ./word_ladder wordlist-four-letters.txt
Your wish is my command:
  1 WORD       (list the connected component WORD belongs to)
  2 FROM TO    (list the shortest path from FROM to TO)
  3            (terminate)
> 2
pesa
vara
O caminho mais curto é:
vara
para
pera
pesa
Your wish is my command:
  1 WORD       (list the connected component WORD belongs to)
  2 FROM TO    (list the shortest path from FROM to TO)
  3            (terminate)
```

Figura 4.11: PathFinder: vara -> pesa

```
Your wish is my command:
  1 WORD       (list the connected component WORD belongs to)
  2 FROM TO    (list the shortest path from FROM to TO)
  3            (terminate)
> 2
golo
taco
O caminho mais curto é:
taco
talo
tolo
golo
Your wish is my command:
  1 WORD       (list the connected component WORD belongs to)
  2 FROM TO    (list the shortest path from FROM to TO)
  3            (terminate)
```

Figura 4.12: PathFinder: taco -> golo

```
thephantomedge@LAPTOP-82HCKN88:/mnt/c/Users/Eduardo BOT/Desktop/AED2/A02$ ./word_ladder wordlist-four-letters.txt
Your wish is my command:
  1 WORD       (list the connected component WORD belongs to)
  2 FROM TO    (list the shortest path from FROM to TO)
  3            (terminate)
> 2
taco
golo
O caminho mais curto é:
golo
tolo
talo
taco
Your wish is my command:
  1 WORD       (list the connected component WORD belongs to)
  2 FROM TO    (list the shortest path from FROM to TO)
  3            (terminate)
```

Figura 4.13: PathFinder: golo -> taco

Capítulo 5

Conclusão

Este projeto "Word_Ladder"ensinou-nos a trabalhar e implementar Hash_Tables, tal como os benefícios e uso do algoritmo de pesquisa em largura, promoveu a nossa aprendizagem de linguagem C, na construção de funções e na procura do desconhecido, melhorou o nosso conhecimento de grafos e de componentes conexos na programação, tal como nos incentivou a utilização de ferramentas externas como latex.

No código fornecido do projeto encontrámos um desafio de encontrar o caminho mais curto entre as palavras bem e mal, no entanto, como não encontrámos um ficheiro de texto de palavras de três letras disponibilizado, decidimos ser nós a acrescentar (deixámos o link da fonte onde retirámos a lista de palavras na bibliografia deste relatório), o resultado aparenta ser diferente pois a lista de palavras é diferente.

O nosso principal objetivo era a implementação da tabela de dispersão, grafo, algoritmo de pesquisa em largura e a implementação das funções List Connected Component e Path Finder. Para concluir. ambas as funções principais funcionam, disponibilizámos nos resultados em cima, algumas combinações de palavras que achámos interessantes.

Capítulo 6

Apêndice

```
//
// AED, November 2022 (Tomás Oliveira e Silva)
//
// Second practical assignement (speed run)
//
// Place your student numbers and names here
//   N.Mec. 104179  Name: Eduardo Alves
//   N.Mec. 108780  Name: David Palricas
//   N.Mec. 103477  Name: Inês Santos
// 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 <stdbool.h>
#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
};


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

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

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

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

static hash_table_node_t *allocate_hash_table_node(void)
{
```

```
  hash_table_node_t *node;

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

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


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

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

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

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

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

```
  unsigned int i;

  hash_table = (hash_table_t *)malloc(sizeof(hash_table_t));
  if(hash_table == NULL)
  {
    fprintf(stderr,"create_hash_table: out of memory\n");
    exit(1);
  }
  hash_table->hash_table_size=101;      // the size of the hash table array
  hash_table->number_of_entries=0;    // the number of entries in the hash table
  hash_table->number_of_edges=0;      // number of edges (for information purposes only)
  hash_table->heads=
  (hash_table_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)
{
  int newsize=hash_table->hash_table_size*2+1;
  hash_table_node_t ** newtableheads=
  (hash_table_t **)malloc(sizeof(hash_table_node_t*)*newsize);
  if(newtableheads == NULL){
      fprintf(stderr,"create_new_hash_table: out of memory\n");
      exit(1);
    }
 for(unsigned int i = 0; i < hash_table->hash_table_size;i++){
  hash_table_node_t* tmp = hash_table->heads[i];


  while(tmp != NULL){
      hash_table_node_t* next = tmp->next;
      int index = crc32(tmp->word) % newsize;
      tmp->next = newtableheads[index];
      newtableheads[index] = tmp;
      tmp = next;
    }
  }
  free(hash_table->heads);
```

```
  hash_table->heads = newtableheads;
  hash_table->hash_table_size = newsize;
  }

static void hash_table_free(hash_table_t *hash_table){
  hash_table_node_t *node, *next_node;
  adjacency_node_t *node_adj, *next_node_adj;

  // Percorre as entradas na Hash_Table.
  for(unsigned int i = 0; i < hash_table->hash_table_size; i++){
    node = hash_table->heads[i];
  // Liberta a memória alocada em cada bloco
    while(node != NULL){
      next_node = node->next;
      node_adj = node->head;
       while (node_adj != NULL){
        next_node_adj = node_adj->next;
        free_adjacency_node(node_adj);
        node_adj = next_node_adj;
      }
      free_hash_table_node(node);
      node = next_node;
      // O mesmo que para os nós mas para os adjacentes

    }
  }

  // free the memory allocated for the heads array
  free(hash_table->heads);

  // free the memory allocated for 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)
{
  // Se a HashTable for demasiado pequena, ajusta o tamanho com a função Grow.
  if(hash_table->hash_table_size < hash_table->number_of_entries){
    hash_table_grow(hash_table);
  }
  unsigned i = crc32(word) % hash_table->hash_table_size;
  bool node_found = false;
  hash_table_node_t *node = NULL;

  //Percorrer a lista ligada na posição do array à procura da palavra
```

```
  node = hash_table->heads[i];
  while(!node_found && node != NULL){
  // Compara as palavras entre o Node e a desejada e analisa se são iguais.
  node_found =!strcmp(node->word,word);
  if(node_found){
    break;
  }
  node = node->next;
  }

  if (!node_found && insert_if_not_found){

    // Completa os valores no novo Node.
    node = allocate_hash_table_node();
    node->previous = NULL;
    node->next = NULL;
    node->head = NULL;
    node->number_of_vertices = 1;
    node->number_of_edges = 0;
    node->visited = 0;
    node->representative = node;
    strcpy(node->word, word);

    // Insere o novo Nó no inicio da lista ligada.
    hash_table_node_t *next_node = hash_table->heads[i];
    hash_table->heads[i] = node;
    node->next = next_node;

    hash_table->number_of_entries + 1;
  }

  return node;
}

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

static hash_table_node_t *find_representative(hash_table_node_t *node)
{
  hash_table_node_t *representative,*next_node;
  next_node = node;
  // Encontrar o representante final.
  while(next_node != NULL){
    if(next_node == next_node->representative){
     representative = next_node;
```

```
     break;
    }else{
      next_node = next_node->representative;
    }
  }
  // Apontar os vários Nodes para o representante final.
   while (node != NULL && node != representative){
      next_node=node->representative;
      node->representative = representative;
      node=next_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,*from_representative,*to_representative;
  adjacency_node_t *link, *new_link;

  to = find_word(hash_table,word,0);
  if(to == NULL){
    return;
  }
  //Do from para o to.
  link = allocate_adjacency_node(); //Cria um novo link
  link -> vertex = to; //do outro lado está o to.
  link -> next = from ->head; //Insere o na lista
  from->head = link;
  // Do to para o from.
  new_link = allocate_adjacency_node(); //Cria um novo link
  new_link -> vertex = from; //do outro lado está o from.
  new_link -> next = to ->head; //Insere o na lista
  to->head = new_link;

  //Union Find Data
  from_representative = find_representative(from);
  to_representative = find_representative(to);

  if(from_representative == to_representative){
    return;
  }else{
    //O representante do componente mais pequeno tomar o valor do representante maior
    //e atualizar o respetivo number of vértices.
    if(from_representative-> number_of_vertices > to_representative -> number_of_vertices){
      to_representative->representative = from_representative;
```

```
      from_representative->number_of_vertices = from_representative->number_of_vertices +
      to_representative->number_of_vertices;
    }else{
      from_representative->representative = to_representative;
      to_representative->number_of_vertices =
      to_representative->number_of_vertices + from_representative->number_of_vertices;
    }
  }
}

//
// 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 Ő 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;
  }
}

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)
{
  int in_index = 0;
  bool goalfound_searchstop = 0;
  int out_index = 0;
  origin->visited = 1;
  list_of_vertices[in_index++] = origin;
  adjacency_node_t *adj;
  if(maximum_number_of_vertices<1){
    printf("Erro: Número Máximo de vértices terá que ser maior que 1");
    exit(1);
  }
  while(out_index<in_index && out_index < maximum_number_of_vertices &&
  goalfound_searchstop !=1){
    adj = list_of_vertices[out_index]->head;
    while (adj!=NULL){
      if(adj->vertex->visited==0){
        list_of_vertices[in_index] = adj->vertex;
        adj->vertex->visited = 1;
        list_of_vertices[in_index]->previous = list_of_vertices[out_index];
         in_index+=1;
        }
        if(adj->vertex==goal){
          goalfound_searchstop = 1;
          break;
        }

        adj = adj->next;
      }
    out_index+=1;
  }
  // Última parte da função, coloca todos os Nodes a não visitados
  //para a próxima pesquisa poder marcar novamente as visitas
    for(int i = 0; i< in_index;i++){
      list_of_vertices[i]->visited = 0;
```

```
    }
    return in_index;
}

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

static void list_connected_component(hash_table_t *hash_table,const char *word)
{
  hash_table_node_t *representative;
  hash_table_node_t *Node;
  int tmp;


  if((Node = find_word(hash_table,word,0)) == NULL){
    fprintf(stderr,"Palavra não é válida\n");
      exit(1);

  }
  representative=find_representative(Node);

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

  tmp = breadh_first_search(representative->number_of_vertices,list_of_vertices, Node, NULL)
  if(tmp!=representative->number_of_vertices){
  printf("O número de vértices visitados é %d: \n", tmp);
  printf("O número de vértices totais é %d:", representative->number_of_vertices);
  // for(int i = 0; i< representative->number_of_vertices;i++){
  fprintf(stderr,"A função breadh first search não conseguiu navegar todos os vértices\n");
    exit(1);
  //}
  }
  printf("A lista de palavras ligadas da word %s são:  \n",word);
  for(int i = 0; i< representative->number_of_vertices;i++){
    printf("%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;

// {
//   int diameter;

//   //
//   // complete this
//   //
//   return diameter;
// }

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

static void path_finder(hash_table_t *hash_table,const char *from_word,const char *to_word)
{
    hash_table_node_t  *from, *to;
    hash_table_node_t  *from_representative, *to_representative;
    from = find_word(hash_table, from_word, 0);
    if (from == NULL) {
        fprintf(stderr, "Palavra não é válida\n");
        return;
    }
    to = find_word(hash_table, to_word, 0);
    if (to == NULL) {
        fprintf(stderr, "Palavra não é válida\n");
        return;
    }
    from_representative = find_representative(from);
    to_representative = find_representative(to);

    // Check if the two words have the same representative
    if (from_representative != to_representative){
        fprintf(stderr, "As palavras não tem um caminho comum\n");
        return;
    }
    // Alocar memória
    hash_table_node_t **list_of_vertices =
    (hash_table_node_t **)malloc
    (from_representative->number_of_vertices*sizeof(hash_table_node_t*));

    // Usar a função breadth-first search
    breadh_first_search(from_representative->number_of_vertices, list_of_vertices, from, to)
```

```
    // Imprimir o caminho mais curto
    hash_table_node_t *current_node = to;
    printf("O caminho mais curto é: \n");
    while (current_node != NULL) {
        printf("%s\n", current_node->word);
        current_node = current_node->previous;
    }
    //Libertar a lista de vértice
    free(list_of_vertices);
}

  //verificar se as palavras existem from e tu, senao existirem acabou ->dá erro
  //verificar se elas proprias sao iguais
  //verificar se os representantes sáo diferentes
  //Se forem diferentes não há caminho -> erro (nao há caminho)
  //Se houver caminho chamar função breath first
  //usar list connected
  //no final seguir caminho dos previous e imprimir os previous
  //
  //}

//
// some graph information (optional)
//
static void graph_info(hash_table_t *hash_table)
{
  //
  // complete this
  //
}
//
// 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);
  // 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);
  graph_info(hash_table);
  // ask what to do
  for(;;)
  {
    fprintf(stderr,"Your wish is my command:\n");
    fprintf(stderr,"  1 WORD       (list the connected component WORD belongs to)\n");
    fprintf(stderr,"  2 FROM TO    (list the shortest path from FROM to TO)\n");
    fprintf(stderr,"  3            (terminate)\n");
    fprintf(stderr,"> ");
    if(scanf("%99s",word) != 1)
      break;
    command = atoi(word);
    if(command == 1)
    {
      if(scanf("%99s",word) != 1)
        break;
      list_connected_component(hash_table,word);
    }
    else if(command == 2)
    {
      if(scanf("%99s",from) != 1)
        break;
      if(scanf("%99s",to) != 1)
        break;
      path_finder(hash_table,from,to);
    }
    else if(command == 3)
      break;
  }
  // clean up
  hash_table_free(hash_table);
  return 0;
}
```

Capítulo 7

Bibliografia

- https://www.programiz.com/c-programming/library-function/string.h/strcmp
- https://www.programiz.com/c-programming/library-function/string.h/strcpy
- https://stackoverflow.com/questions/71639226/searching-a-hash-table-with-singly-linked-list-in-c
- https://www.geeksforgeeks.org/connected-components-in-an-undirected-graph/
- https://www.dicio.com.br/palavras-com-tres-letras/
- https://en.wikipedia.org/wiki/Breadth-first_search
- https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/
- https://www.tutorialspoint.com/c_standard_library/c_function_free.htm