deti universidade de aveiro
departamento de eletrónica,
telecomunicações e informática

Word Ladder

Algoritmos e Estruturas de Dados 2022

Professor Tomás Silva e Professor Pedro Lavrador

Trabalho realizado por:

João Nuno da Silva Luís (107403) | 50%

Diana Raquel Rodrigues Miranda (107457) | 50%

Índice

Índice de figuras

Introdução

Este trabalho é realizado no âmbito da disciplina de Algoritmos e Estruturas de Dados do 2º ano da Licenciatura em Engenharia Informática.

Foi-nos proposto desenvolver um programa em C capaz de produzir um word ladder, isto é, que fosse capaz de formar uma sequência de palavras em que duas palavras adjacentes diferem apenas por uma letra. Este trabalho foi dividido em partes (obrigatórias, altamente recomendadas, recomendadas e opcionais).

Como obrigatório temos: Apresentar uma implementação completamente funcional de uma hash table que atualiza dinamicamente o seu tamanho sempre que necessário e apresentar alguns dados estatísticos sobre a hash table.

Como altamente recomendado temos: Construir o grafo já incluindo o union-find.

Como recomendado temos: Implementar a pesquisa no grafo usando o breadth-first search, listar todas as palavras pertencentes a um componente conectado e encontrar o caminho mais curto entre duas palavras;

Como opcional temos: Calcular o diâmetro de um componente conectado e listar a cadeia de palavras mais longa, mostrar alguns dados estatísticos sobre o grafo e confirmar se existe memory leaks.

Com isto em mente, temos como objetivo principal realizar todos os pontos descritos acima, tendo principal foco nos pontos obrigatórios e nos altamente recomendados.

Implementação da hash table

- **Função hash_table_create**

```
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 = 2000;
  hash_table->number_of_entries = 0;
  hash_table->number_of_edges = 0;
  hash_table->heads = (hash_table_node_t **)malloc((size_t)hash_table->hash_table_size * sizeof(hash_table_node_t *));
  if (hash_table->heads == NULL)
  {
    fprintf(stderr, "allocate hash_table->heads: out of memory\n");
    exit(1);
  }

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

  return hash_table;
}
```

Figura 1 - Criação da hash table.

Para a criação da hash table começámos por inicializar os valores de alguns dados relativos à mesma. Definimos então um tamanho inicial de 2000, o número de entradas a zero e o número de arestas também a zero. Em seguida alocámos o espaço necessário para essa hash table. Por fim, inicializamos todos os nós da hash table a NULL.

- **Função hash_table_grow**

```c
static void hash_table_grow(hash_table_t *hash_table)
{

  hash_table_node_t **old_heads = hash_table->heads;
  unsigned int old_hash_table_size = hash_table->hash_table_size;
  unsigned int i;

  hash_table->hash_table_size = hash_table->hash_table_size * 2;
  hash_table->heads = (hash_table_node_t **)malloc((size_t)hash_table->hash_table_size * sizeof(hash_table_node_t *));

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

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

  for (i = 0u; i < old_hash_table_size; i++)
  {
    hash_table_node_t *node = old_heads[i];
    while (node != NULL)
    {
      hash_table_node_t *next = node->next;
      unsigned int j = crc32(node->word) % hash_table->hash_table_size;
      node->next = hash_table->heads[j];
      hash_table->heads[j] = node;
      node = next;
    }
  }

  printf("hash_table_grow: old hash table size = %u new hash table size = %u\n", old_hash_table_size, hash_table->hash_table_size);

  free(old_heads);
}
```

Figura 2 - Ajuste dinâmico do tamanho da hash table.

Esta função é o que vai permitir que a hash table ajuste dinamicamente o seu tamanho sempre que necessário.

Para isso, começámos por copiar todos os nós da hash table para uma nova variável (old_heads) e guardámos também, numa variável (old_hash_table_size), o tamanho da hash table atual. Seguidamente, escolhemos aumentar o tamanho da hash table para o dobro e alocámos esse espaço. Após este aumento da tabela colocámos todos os nós a null para iniciar essa hash table como o tamanho novo. Depois voltámos a colocar todos os nós que foram guardados inicialmente, na variável old_heads, na hash table com tamanho atualizado.

Por fim, decidimos fazer um print com a informação da atualização do tamanho para uma melhor perceção do funcionamento desta função quando corremos o programa. Também libertámos os antigos heads guardados inicialmente, uma vez que não iriamos precisar mais deles.

- **Função hash_table_free**

```c
static void hash_table_free(hash_table_t *hash_table)
{
  unsigned int i;
  hash_table_node_t *node;
  hash_table_node_t *next;
  adjacency_node_t *adjacency_node;
  adjacency_node_t *next_adjacency_node;

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

      while (adjacency_node != NULL)
      {
        next_adjacency_node = adjacency_node->next;
        free_adjacency_node(adjacency_node);
        adjacency_node = next_adjacency_node;
      }

      free_hash_table_node(node);
      node = next;
    }
  }

  free(hash_table->heads);
  free(hash_table);
}
```

Figura 3 - Libertação de todo o espaço alocado pela hash table.

O objetivo desta função é libertar todo o espaço alocado na utilização da hash table evitando assim que existam memory leaks.

Nesta função começámos por percorrer todos os nós da hash table e em cada nó verificámos se existia algum nó de adjacência, em caso afirmativo, todos eles eram percorridos e iam sendo libertados um a um. A mesma lógica foi aplicada a libertação de cada nó da hash table.

- **Função find_word**

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

  i = crc32(word) % hash_table->hash_table_size;

  if (insert_if_not_found)
  {

    node = allocate_hash_table_node();
    strcpy(node->word, word);

    if (hash_table->number_of_entries > hash_table->hash_table_size)
      hash_table_grow(hash_table);

    node->next = hash_table->heads[i];
    hash_table->heads[i] = node;

    hash_table->number_of_entries++;
    node->representative = node;
    node->number_of_vertices = 1;
    node->visited = 0;
    node->head = NULL;
    node->last_word = NULL;
  }
  else
  {
    node = hash_table->heads[i];
    while (node != NULL)
    {
      if (strcmp(node->word, word) == 0)
      {
        return node;
      }
      node = node->next;
    }
  }

  return NULL;
}
```

Figura 4 - Procura ou inserção das palavras na hash table.

Esta função tem dois papéis. Pode ser usada para inserir uma palavra na hash table ou então para procurar uma palavra nela. Para distinguir qual o papel que a função irá exercer é passado como argumento a variável insert_if_not_find.

Se a variável insert_if_not_find for passada com o valor 1, então significa que a intenção é inserir a palavra na hash table. Para isso, fazemos a alocação de um novo nó e caso o número de entradas na hash table já seja superior ao tamanho da hash table, fazemos o aumento dela e só depois inserimos a nova palavra. Depois da inserção feita atualizámos o número de entradas e já definimos o representante desse nó como ele próprio, o número de vértices como 1 e inicializamos as restantes variáveis.

Se a variável insert_if_not_find for passada com o valor 0, o papel da função será procurar uma palavra na hash table. Começando por ir buscar o nó que, em princípio, corresponde à palavra que se procura, e depois fazendo a comparação entre a palavra nesse nó e a palavra procurada, se

houver uma correspondência, o valor retornado é esse nó, senão o valor retornado é NULL, que significa que a palavra não existe na hash table.

- **Alguns dados sobre a hash table**

  - Impressão da hash table

Para conseguirmos ter uma melhor visão sobre o funcionamento da hash table e inserção de dados na mesma, criámos uma função que imprimisse no terminal a hash table completa, isto é, com todas as palavras já inseridas.

```
void print_table(hash_table_t *hash_table)
{
  for (unsigned 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", i);
      hash_table_node_t *tmp = hash_table->heads[i];
      while (tmp != NULL)
      {
        printf("%s ->", tmp->word);
        tmp = tmp->next;
      }
      printf("\n");
    }
  }
}
```

Figura 5 - Função para visualizar as palavras na hash table

  - Informação sobre as colisões da hash table

De modo a ser possível verificarmos quantas colisões existe na hash table, criámos uma função que conta quantos espaços da hash table guardam 0, 1,2 até 10 palavras. Isto permite-nos ter uma ideia sobre se a nossa hash table é eficiente, depois de ser aplicada a hash function.

```
void count_colisions(hash_table_t *hash_table)
{
  int colisions = 0;

  int array_numColisions[11] = {0};
  for (unsigned int i = 0; i < hash_table->hash_table_size; i++)
  {
    if (hash_table->heads[i] == NULL)
    {
      array_numColisions[0]++;
    }
    if (hash_table->heads[i] != NULL)
    {
      colisions = 1;
      hash_table_node_t *tmp = hash_table->heads[i];
      while (tmp->next != NULL)
      {
        colisions++;
        tmp = tmp->next;
      }
      array_numColisions[colisions]++;
    }
  }

  printf("Number of positions on hash table with x words: \n");
  int n_colisions = 0;
  for (int i = 0; i < 11; i++)
  {
    if (array_numColisions[i] != 0)
    {
      printf("  %i words: %i\n", i, array_numColisions[i]);
    }
    if (i > 1)
    {
      n_colisions = n_colisions + array_numColisions[i];
    }
  }
  printf("\nRate of colisions: %0.02f%%\n", (double)n_colisions / hash_table->hash_table_size * 100);
}
```

Figura 6 - Função para contar as colisões existentes na hash table

Construção do grafo

- **Função find_representative**

```
static hash_table_node_t *find_representative(hash_table_node_t *node)
{
  hash_table_node_t *representative, *next_node, *current_node;

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

  for (current_node = node; current_node != representative; current_node = next_node)
  {
    next_node = current_node->representative;
    current_node->representative = representative;
  }

  return representative;
}
```

Figura 7 - Função para encontrar o representante de um componente conexo

Esta função tem como objetivo encontrar o representante do componente conexo. Para isto usámos a estrutura de dados union-find, que é uma maneira eficiente de acompanhar os componentes conectados de um grafo não direcionado.

Posto isto, temos o primeiro "for" que serve para encontrar o link para o nó representativo e o segundo "for" que faz a atualização do representativo de todos os vértices do componente conexo para o mesmo representante, isto denomina-se de compressão do caminho.

- **Função add_edge**

Esta função tem como finalidade adicionar as arestas do grafo. Para isso, é lhe passada um nó da hash table (a palavra trom) e uma palavra final (to). Depois de verificar se "to" pertence à lista de palavras, isto é, se está na hash table é criado um link entre a palavra "to" e "from" e também da palavra "from" para "to".

Por fim, para manter os dados do union-find, verifica-se se o representativo das duas palavras é o mesmo, se não for faz-se a ligação dos dois componentes conexos.

```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;

  to = find_word(hash_table, word, 0);

  if (to == NULL)
  {
    return;
  }

  if (to != NULL)
  {

    link = allocate_adjacency_node();
    link->vertex = to;
    link->next = NULL;

    if (from->head == NULL)
    {
      from->head = link;
      from->number_of_edges++;
    }
    else
    {
      link->next = from->head;
      from->head = link;
      from->number_of_edges++;
    }

    link = allocate_adjacency_node();
    link->vertex = from;
    link->next = NULL;

    if (to->head == NULL)
    {
      to->head = link;
      to->number_of_edges++;
    }
    else
    {
      link->next = to->head;
      to->head = link;
      to->number_of_edges++;
    }

    hash_table->number_of_edges++;

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

    if (from_representative != to_representative)
    {
      if (from_representative->number_of_vertices > to_representative->number_of_vertices)
      {
        to_representative->representative = from_representative;
        from_representative->number_of_vertices += to_representative->number_of_vertices;
      }
      else
      {
        from_representative->representative = to_representative;
        to_representative->number_of_vertices += from_representative->number_of_vertices;
      }
    }
  }
}
```

Figura 8 - Função para adicionar aresta ao gráfico

- **Função breadh_first_search**

```c
static hash_table_node_t **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)
{

  list_of_vertices[0] = origin;
  origin->visited = 1;
  origin->previous = NULL;
  int j = 1;
  int i = 0;

  while (i < j && i < maximum_number_of_vertices)
  {

    hash_table_node_t *current = list_of_vertices[i];
    adjacency_node_t *adjacency_node = current->head;

    while (adjacency_node != NULL)
    {
      if (adjacency_node->vertex->visited == 0)
      {
        adjacency_node->vertex->visited = 1;
        adjacency_node->vertex->previous = current;
        list_of_vertices[j] = adjacency_node->vertex;
        j++;
      }

      adjacency_node = adjacency_node->next;
    }
    i++;
  }

  origin->last_word = list_of_vertices[j - 1];

  for (int i = 0; i < j; i++)
  {
    list_of_vertices[i]->visited = 0;
  }

  return list_of_vertices;
}
```

Figura 9 - Função Breadh First Search

Utilizando o método de Breath First Search, nesta função vai ser feita a procura do caminho mais curto entre duas palavras. Para esta procura, começamos por definir a primeira posição do array como a palavra de origem (vai ser a raiz do grafo), marca-se essa posição como visitada e sem valor anterior. A partir daí é feita a procura de todos os nós adjacentes, sempre que houver um nó adjacente que não tenha sido visitado é acrescentado ao array essa palavra, atualizando o valor da variável “visited” e do ponteiro “previous”. No geral o que esta função faz é usar um array duplamente ligado com todas as palavras às quais a palavra de origem consegue chegar e através do ponteiro “previous” consegue descobrir o menor caminho entre duas palavras.

Depois de ter o array completo guardámos a última palavra, esta informação vai ser utilizada posteriormente para ver o caminho com maior diâmetro. Por fim, voltasse a marcar todos os vértices como não visitados e é retornado a lista dos vértices.

Interação com o grafo

- **Função list_connected_component**

```c
static void list_connected_component(hash_table_t *hash_table, const char *word)
{

  hash_table_node_t *node = find_word(hash_table, word, 0);

  if (node == NULL)
  {
    printf("Word not found.\n");
    return;
  }

  hash_table_node_t **list_of_vertices;
  list_of_vertices = (hash_table_node_t **)malloc((size_t)node->representative->number_of_vertices * sizeof(hash_table_node_t *));
  if (list_of_vertices == NULL)
  {
    fprintf(stderr, "malloc failed: out of memory\n");
    exit(1);
  }

  for (unsigned int i = 0u; i < node->representative->number_of_vertices; i++)
    list_of_vertices[i] = NULL;

  printf("Connected component %s belgons to: \n", node->word);

  list_of_vertices = breadh_first_search(node->representative->number_of_vertices, list_of_vertices, node, NULL);

  for (int i = 0; i < node->representative->number_of_vertices; i++)
  {
    printf("[%d] %s \n", i, list_of_vertices[i]->word);
  }

  free(list_of_vertices);
}
```

Figura 10 - Função que lista vértices do componente ligado

Esta função tem como objetivo listar todas as palavras às quais se pode chegar partindo da palavra escolhida. Para isso, é alocado espaço para um array que vai ser utilizado pela função breadh_frist_search, esse array é inicializado com todas as posições a NULL. Faz-se a chamada da função breadh_frist_search e com o array, por esta, retornado faz-se a impressão no terminal de todas as palavras nele presentes. Por último, é feito a libertação da memória ocupada pelo array criado.

- **Função path_finder**

```c
static void path_finder(hash_table_t *hash_table, const char *from_word, const char *to_word)
{

  printf("path from %s to %s:\n", from_word, to_word);

  hash_table_node_t *from = find_word(hash_table, from_word, 0);
  hash_table_node_t *to = find_word(hash_table, to_word, 0);

  hash_table_node_t **list_of_vertices;
  list_of_vertices = (hash_table_node_t **)malloc((size_t)from->representative->number_of_vertices * sizeof(hash_table_node_t *));
  if (list_of_vertices == NULL)
  {
    fprintf(stderr, "malloc failed: out of memory\n");
    exit(1);
  }

  for (unsigned int i = 0u; i < from->representative->number_of_vertices; i++)
   list_of_vertices[i] = 0;

  list_of_vertices = breadh_first_search(from->representative->number_of_vertices, list_of_vertices, to, from);

  hash_table_node_t *current = from;
  int x = 0;
  while (current != NULL)
  {
    printf("[%d] %s \n", x, current->word);
    current = current->previous;
    x++;
  }

  free(list_of_vertices);
}
```

Figura 11 - Função para encontrar o caminho mais curto entre 2 palavras.

Esta função tem como finalidade mostrar o caminho mais curto entre duas palavras. O seu funcionamento é muito semelhante à função list_connected_component (descrita acima), a única diferença é que nesta função serão apenas impressas as palavras que formam o caminho mais curto entre a palavra “from” e a palavra “to”.

- **Função graph_info**

```c
static void graph_info(hash_table_t *hash_table)
{
  int number_of_connected_components = 0;

  hash_table_node_t *node;
  unsigned int i;

  for (i = 0u; i < hash_table->hash_table_size; i++)
  {
    for (node = hash_table->heads[i]; node != NULL; node = node->next)
    {
      if (node != NULL && node->representative == node)
      {
        number_of_connected_components++;
      }
    }
  }

  printf("Number of vertices in the graph: %d\n", hash_table->number_of_entries);
  printf("Number of edges in the graph: %d\n", hash_table->number_of_edges);
  printf("Number of connected components: %d\n", number_of_connected_components);

  unsigned int j = 0;
  int diameter = 0;
  int array[number_of_connected_components];

  for (i = 0u; i < hash_table->hash_table_size; i++)
  {
    for (node = hash_table->heads[i]; node != NULL; node = node->next)
    {
      if (node != NULL && node->representative == node)
      {
        diameter = connected_component_diameter(node);
        array[j] = diameter;
        j++;
      }
    }
  }

  int current = 0;
  for (i = 0; i < number_of_connected_components; i++)
  {
    for (j = i + 1; j < number_of_connected_components; j++)
    {
      if (array[i] > array[j])
      {
        current = array[i];
        array[i] = array[j];
        array[j] = current;
      }
    }
  }

  printf("\nNumber of connected components with a diameter of: \n");
  count = 1;
  for (i = 0; i < number_of_connected_components; i++)
  {
    if (i == number_of_connected_components - 1 && array[i] != array[i - 1])
    {
      printf("  %d : %d\n", array[i], count);
      break;
    }

    if (array[i] == array[i + 1])
    {
      count++;
    }
    else
    {
      printf("  %d : %d\n", array[i], count);
      count = 1;
    }
  }

  printf("\nLargest word ladder: \n");
  path_finder(hash_table, largest_diameter_node->word, largest_diameter_node->last_word->word);
}
```

Figura 12 - Função que apresenta as informações do grafo.

Esta função mostra algumas informações sobre o grafo, como o número total de vértices e arestas que este possui, o número de componentes ligadas com um determinado diâmetro e o caminho entre as duas palavras com o maior diâmetro.

Execução do programa

- Início

```
diana@diana-Legion-5-Pro-16ACH6:~/LEI/AED/Pojeto2/A02$ ./word_ladder
hash_table_grow: old hash table size = 2000 new hash table size = 4000
hash_table_grow: old hash table size = 4000 new hash table size = 8000
hash_table_grow: old hash table size = 8000 new hash table size = 16000
hash_table_grow: old hash table size = 16000 new hash table size = 32000
hash_table_grow: old hash table size = 32000 new hash table size = 64000
hash_table_grow: old hash table size = 64000 new hash table size = 128000
hash_table_grow: old hash table size = 128000 new hash table size = 256000
hash_table_grow: old hash table size = 256000 new hash table size = 512000
hash_table_grow: old hash table size = 512000 new hash table size = 1024000

Hash table construction time: 84.003 seconds

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 See Hash Table
  4 See Colisions
  5 See graph info
  6 (terminate)
>
```

Ao executar o programa sem nenhum parâmetro ele vai ler as palavras do ficheiro “wordlist-big-latest.txt”. Enquanto faz a leitura dessas palavras e as insere na hash table vai sendo impresso todos os ajustes feitos no tamanho da hash table. Depois de todos as palavras inseridas é apresentado no ecrã o tempo que demorou a completar a hash table e, seguidamente, é exibido o menu.

- Opção 1

```
> 1 consoada
Connected component consoada belgons to:
[0] consoada
[1] consoado
[2] consoava
[3] consoara
[4] constada
[5] constado
[6] constava
[7] consoará
[8] consogra
[9] constara
[10] constata
[11] conotada
[12] conotado
[13] constato
[14] conotava
[15] constará
[16] consagra
[17] conotara
[18] constate
[19] conotará
[20] consagro
[21] consagre

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 See Hash Table
  4 See Colisions
  5 See graph info
  6 (terminate)
>
```

Selecionando a opção 1, com por exemplo a palavra “consoada”, é impresso no terminal todas as palavras que estão ligadas a esta.

- Opção 2

```
> 2 tudo nada
path from tudo to nada:
[0] tudo
[1] todo
[2] nodo
[3] nado
[4] nada

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 See Hash Table
  4 See Colisions
  5 See graph info
  6 (terminate)
>
```

Selecionando a opção 2, com por exemplo a palavra “tudo” e “nada”, é impresso no terminal o caminho mais curto entre essas palavras. Ou seja, é impresso todas as palavras que permitem chegar de “tudo” para “nada” mais rapidamente.

Mais alguns exemplos de vários caminhos:

```
> 2 bem mal
path from bem to mal:
[0] bem
[1] tem
[2] teu
[3] meu
[4] mau
[5] mal
```

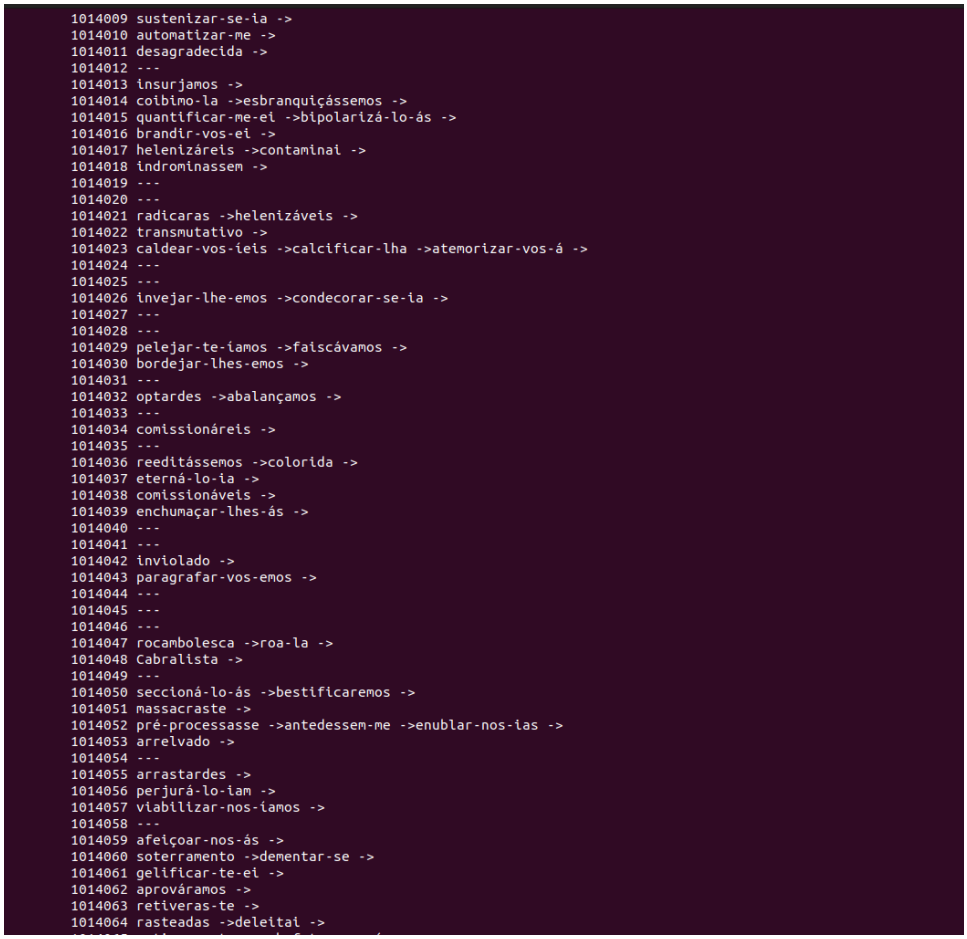
```
> 2 mãe pai
path from mãe to pai:
[0] mãe
[1] mie
[2] fie
[3] fiz
[4] faz
[5] paz
[6] pai
```

```
> 2 triste chorar
path from triste to chorar:
[0] triste
[1] traste
[2] araste
[3] ataste
[4] ateste
[5] atente
[6] atende
[7] acende
[8] acendi
[9] acenai
[10] acenas
[11] arenas
[12] areias
[13] creias
[14] cheias
[15] checas
[16] chocas
[17] chocar
[18] chorar
```

```
> 2 calma fúria
path from calma to fúria:
[0] calma
[1] calda
[2] carda
[3] caria
[4] faria
[5] fúria
```

- Opção 3

Nesta opção é impresso no terminal toda as posições da hash table, com as respetivas palavras.



Como o tamanho total da hash table é 1024000, no print acima só é apresentado uma parte dela.

- Opção 4

```
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 See Hash Table
  4 See Colisions
  5 See graph info
  6 (terminate)
> 4
Number of positions on hash table with x words:
  0 words: 386313
  1 words: 376605
  2 words: 183086
  3 words: 59676
  4 words: 14795
  5 words: 2947
  6 words: 494
  7 words: 75
  8 words: 8
  9 words: 1

Rate of colisions: 25.50%
```

Selecionando a opção 4 é impresso no terminal quantas posições tem um certo número de palavras guardadas e a percentagem de colisões da hash table.

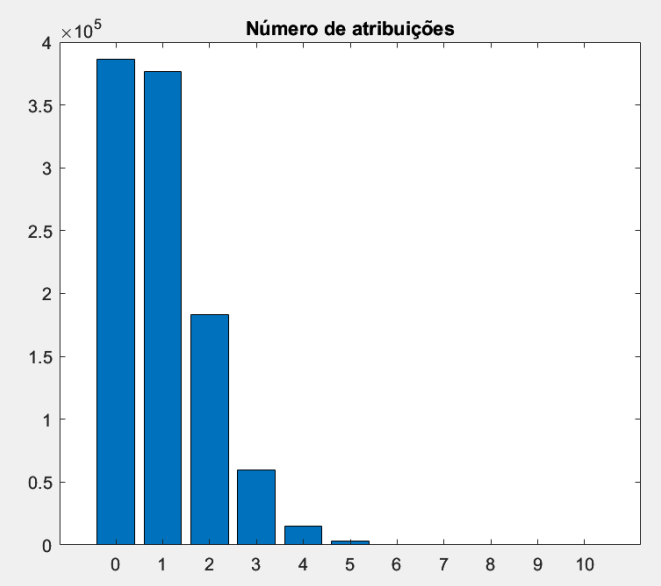


Figura 13 - Gráfico com número de atribuições da hash table

Através deste gráfico vemos que existem algumas colisões, no entanto como o ficheiro lido tem perto de um milhão de palavras é normal.

- Opção 5

Selecionando a opção 5 é impresso no terminal todas as informações relativamente ao grafo.

```
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 See Hash Table
  4 See Colisions
  5 See graph info
  6 (terminate)
> 5
Number of vertices in the graph: 999282
Number of edges in the graph: 1060534
Number of connected components: 377234

Number of connected components with a diameter of:
  1 : 184869
  2 : 141738
  3 : 36459
  4 : 6672
  5 : 3042
  6 : 2226
  7 : 1321
  8 : 251
  9 : 139
  10 : 114
  11 : 81
  12 : 58
  13 : 47
  14 : 44
  15 : 36
  16 : 34
  17 : 17
  18 : 13
  19 : 22
  20 : 26
  21 : 8
  22 : 4
  23 : 3
  24 : 1
  25 : 1
  27 : 1
  28 : 2
  35 : 1
  38 : 1
  43 : 1
  50 : 1
  52 : 1
```

```
Largest word ladder:
path from carradas to empaleou:
[0] carradas
[1] cerradas
[2] cerrados
[3] cerramos
[4] cerremos
[5] corremos
[6] coaremos
[7] toaremos
[8] traremos
[9] araremos
[10] afaremos
[11] afanemos
[12] afanamos
[13] afanados
[14] afinados
[15] afinadas
[16] afinaras
[17] afilaras
[18] afilares
[19] axilares
[20] exilares
[21] exalares
[22] exarares
[23] exararas
[24] exaradas
[25] exarados
[26] exaramos
[27] exaremos
[28] extremos
[29] entremos
[30] enteemos
[31] enfeemos
[32] enfermos
[33] enfermas
[34] enformas
[35] enfornas
[36] encornas
[37] encarnas
[38] escarnas
[39] escarpas
[40] escalpas
[41] escaldas
[42] escaldar
[43] espaldar
[44] espalhar
[45] espalhas
[46] espelhas
[47] espelhos
[48] espelhou
[49] espalhou
[50] empalhou
[51] empaleou
```

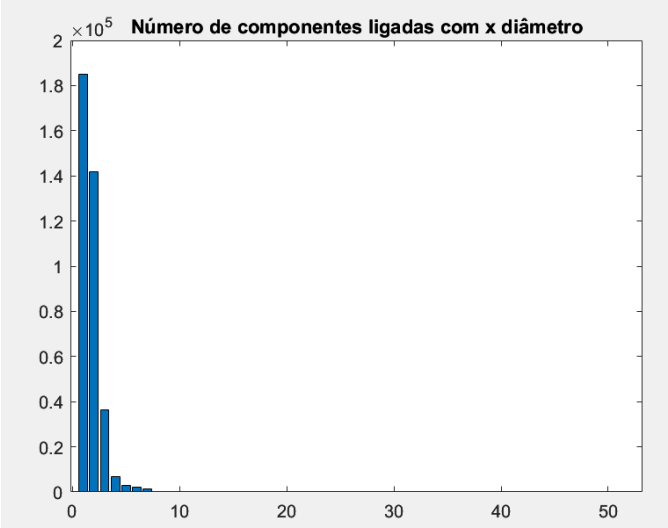


Figura 14 - Gráfico com o número de componentes ligados com um certo diâmetro

- Opção 6

Esta opção termina o programa. Se o programa for executado com valgrind, quando a sua execução for terminada é apresentado no ecrã a informação sobre a alocação e desalocação de memória durante a execução do programa.

```
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 See Hash Table
  4 See Colisions
  5 See graph info
  6 (terminate)
> 6
==6844==
==6844== HEAP SUMMARY:
==6844==     in use at exit: 0 bytes in 0 blocks
==6844==   total heap usage: 3,120,366 allocs, 3,120,366 frees, 138,263,992 bytes allocated
==6844==
==6844== All heap blocks were freed -- no leaks are possible
==6844==
==6844== For lists of detected and suppressed errors, rerun with: -s
==6844== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
diana@diana-Legion-5-Pro-16ACH6:~/LEI/AED/Pojeto2/A02$
```

O nosso código faz a desalocação de toda a memória alocado, não havendo memory leaks.

Conclusão

Com a realização deste trabalho conseguimos aprofundar os nossos conhecimentos acerca de vários assuntos lecionados ao longo da disciplina de Algoritmos e Estruturas de Dados, particularmente sobre Hash Tables, Linked Lists, Doubly Linked Lists, Breadh First Search e grafos não direcionados utilizando o union-find. Para além disto, este trabalho permitiu-nos também utilizar vários conceitos que estão por base na Linguagem de Programação em C, como a alocação e desalocação de memória.

À medida que fomos progredindo e compreendendo o que o enunciado do problema nos propunha, fomos desenvolvendo o código que nos era pedido, aos poucos e com cuidado, para ter a certeza de que a implementação da hash table e a construção do grafo eram efetuadas corretamente.

Desta forma, podemos concluir que os objetivos propostos foram alcançados com sucesso, uma vez que temos uma hash table e um grafo funcional.

Código C

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <time.h>

#define _max_word_size_ 32

typedef struct adjacency_node_s adjacency_node_t;
typedef struct hash_table_node_s hash_table_node_t;
typedef struct hash_table_s hash_table_t;

struct adjacency_node_s
{
  adjacency_node_t *next;    // link to the next adjacency list node
  hash_table_node_t *vertex; // the other vertex
};

struct hash_table_node_s
{
  // the hash table data
  char word[_max_word_size_]; // the word
  hash_table_node_t *next;    // next hash table linked list node
  // the vertex data
  adjacency_node_t *head;      // head of the linked list of 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)
  hash_table_node_t *last_word;      // last word of the conected 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
```

```
};
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);
}
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; // depois é preciso dividir pelo hash_table_size
}

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 = 2000;
  hash_table->number_of_entries = 0;
  hash_table->number_of_edges = 0;
  hash_table->heads = (hash_table_node_t **)malloc((size_t)hash_table->hash_table_size *
sizeof(hash_table_node_t *));
  if (hash_table->heads == NULL)
  {
    fprintf(stderr, "allocate hash_table->heads: out of memory\n");
    exit(1);
  }

  for (i = 0u; 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_heads = hash_table->heads;
  unsigned int old_hash_table_size = hash_table->hash_table_size;
  unsigned int i;

  hash_table->hash_table_size = hash_table->hash_table_size * 2;
  hash_table->heads = (hash_table_node_t **)malloc((size_t)hash_table->hash_table_size *
sizeof(hash_table_node_t *));

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

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

  for (i = 0u; i < old_hash_table_size; i++)
  {
    hash_table_node_t *node = old_heads[i];
    while (node != NULL)
    {
      hash_table_node_t *next = node->next;
      unsigned int j = crc32(node->word) % hash_table->hash_table_size;
      node->next = hash_table->heads[j];
      hash_table->heads[j] = node;
      node = next;
    }
  }

  printf("hash_table_grow: old hash table size = %u new hash table size = %u\n",
old_hash_table_size, hash_table->hash_table_size);

  free(old_heads);
}

static void hash_table_free(hash_table_t *hash_table)
{
  unsigned int i;
  hash_table_node_t *node;
  hash_table_node_t *next;
  adjacency_node_t *adjacency_node;
  adjacency_node_t *next_adjacency_node;

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

```
  {
    node = hash_table->heads[i];
    while (node != NULL)
    {
      next = node->next;
      adjacency_node = node->head;

      while (adjacency_node != NULL)
      {
        next_adjacency_node = adjacency_node->next;
        free_adjacency_node(adjacency_node);
        adjacency_node = next_adjacency_node;
      }

      free_hash_table_node(node);
      node = next;
    }
  }

  free(hash_table->heads);
  free(hash_table);
}

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

  i = crc32(word) % hash_table->hash_table_size;

  if (insert_if_not_found)
  {

    node = allocate_hash_table_node();
    strcpy(node->word, word);

    if (hash_table->number_of_entries > hash_table->hash_table_size)
      hash_table_grow(hash_table);

    node->next = hash_table->heads[i];
    hash_table->heads[i] = node;

    hash_table->number_of_entries++;
    node->representative = node;
```

```
    node->number_of_vertices = 1;
    node->visited = 0;
    node->head = NULL;
    node->last_word = NULL;
  }
  else
  {
    node = hash_table->heads[i];
    while (node != NULL)
    {
      if (strcmp(node->word, word) == 0)
      {
        return node;
      }
      node = node->next;
    }
  }

  return NULL;
}

void print_table(hash_table_t *hash_table)
{
  for (unsigned 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", i);
      hash_table_node_t *tmp = hash_table->heads[i];
      while (tmp != NULL)
      {
        printf("%s ->", tmp->word);
        tmp = tmp->next;
      }
      printf("\n");
    }
  }
}

void count_colisions(hash_table_t *hash_table)
{
```

```
  int colisions = 0;

  int array_numColisions[11] = {0};
  for (unsigned int i = 0; i < hash_table->hash_table_size; i++)
  {
    if (hash_table->heads[i] == NULL)
    {
      array_numColisions[0]++;
    }
    if (hash_table->heads[i] != NULL)
    {
      colisions = 1;
      hash_table_node_t *tmp = hash_table->heads[i];
      while (tmp->next != NULL)
      {
        colisions++;
        tmp = tmp->next;
      }
      array_numColisions[colisions]++;
    }
  }

  printf("Number of positions on hash table with x words: \n");
  int n_colisions = 0;
  for (int i = 0; i < 11; i++)
  {
    if (array_numColisions[i] != 0)
    {
      printf("  %i words: %i\n", i, array_numColisions[i]);
    }
    if (i > 1)
    {
      n_colisions = n_colisions + array_numColisions[i];
    }
  }
  printf("\nRate of colisions: %0.02f%%\n", (double)n_colisions / hash_table->hash_table_size *
100);
}

static hash_table_node_t *find_representative(hash_table_node_t *node)
{
  hash_table_node_t *representative, *next_node, *current_node;

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

```
    ;
  for (current_node = node; current_node != representative; current_node = next_node)
  {
    next_node = current_node->representative;
    current_node->representative = representative;
  }

  return representative;
}

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

  to = find_word(hash_table, word, 0);

  if (to == NULL)
  {
    return;
  }

  if (to != NULL)
  {

    link = allocate_adjacency_node();
    link->vertex = to;
    link->next = NULL;

    if (from->head == NULL)
    {
      from->head = link;
      from->number_of_edges++;
    }
    else
    {
      link->next = from->head;
      from->head = link;
      from->number_of_edges++;
    }

    link = allocate_adjacency_node();
    link->vertex = from;
    link->next = NULL;
```

```
    if (to->head == NULL)
    {
      to->head = link;
      to->number_of_edges++;
    }
    else
    {
      link->next = to->head;
      to->head = link;
      to->number_of_edges++;
    }

    hash_table->number_of_edges++;

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

    if (from_representative != to_representative)
    {
      if (from_representative->number_of_vertices > to_representative->number_of_vertices)
      {
        to_representative->representative = from_representative;
        from_representative->number_of_vertices += to_representative->number_of_vertices;
      }
      else
      {
        from_representative->representative = to_representative;
        to_representative->number_of_vertices += from_representative->number_of_vertices;
      }
    }
  }
}
static void break_utf8_string(const char *word, int *individual_characters)
{
  int byte0, byte1;

  while (*word != '\0')
  {
    byte0 = (int)(*(word++)) & 0xFF;
    if (byte0 < 0x80)
      *(individual_characters++) = byte0; // plain ASCII character
    else
    {
      byte1 = (int)(*(word++)) & 0xFF;
      if ((byte0 & 0b11100000) != 0b11000000 || (byte1 & 0b11000000) != 0b10000000)
```

```
      {
        fprintf(stderr, "break_utf8_string: unexpected UFT-8 character\n");
        exit(1);
      }
      *(individual_characters++) = ((byte0 & 0b00011111) << 6) | (byte1 & 0b00111111); // utf8 ->
unicode
    }
  }
  *individual_characters = 0; // mark the end!
}

static void make_utf8_string(const int *individual_characters, char word[_max_word_size_])
{
  int code;

  while (*individual_characters != 0)
  {
    code = *(individual_characters++);
    if (code < 0x80)
      *(word++) = (char)code;
    else if (code < (1 << 11))
    { // unicode -> utf8
      *(word++) = 0b11000000 | (code >> 6);
      *(word++) = 0b10000000 | (code & 0b00111111);
    }
    else
    {
      fprintf(stderr, "make_utf8_string: unexpected UFT-8 character\n");
      exit(1);
    }
  }
  *word = '\0'; // mark the end
}

static void similar_words(hash_table_t *hash_table, hash_table_node_t *from)
{
  static const int valid_characters[] =
      {
// unicode!
       0x2D,
// -
       0x41, 0x42, 0x43, 0x44, 0x45, 0x46, 0x47, 0x48, 0x49, 0x4A, 0x4B, 0x4C,
0x4D,             // A B C D E F G H I J K L M
       0x4E, 0x4F, 0x50, 0x51, 0x52, 0x53, 0x54, 0x55, 0x56, 0x57, 0x58, 0x59,
0x5A,             // N O P Q R S T U V W X Y Z
```

```
        0x61, 0x62, 0x63, 0x64, 0x65, 0x66, 0x67, 0x68, 0x69, 0x6A, 0x6B, 0x6C,
0x6D,             // a b c d e f g h i j k l m
        0x6E, 0x6F, 0x70, 0x71, 0x72, 0x73, 0x74, 0x75, 0x76, 0x77, 0x78, 0x79,
0x7A,             // n o p q r s t u v w x y z
        0xC1, 0xC2, 0xC9, 0xCD, 0xD3,
0xDA,                                                   // Á Â É Í Ó Ú
        0xE0, 0xE1, 0xE2, 0xE3, 0xE7, 0xE8, 0xE9, 0xEA, 0xED, 0xEE, 0xF3, 0xF4, 0xF5, 0xFA, 0xFC,
// à á â ã ç è é ê í î ó ô õ ú ü
        0};
  int i, j, k, individual_characters[_max_word_size_];
  char new_word[2 * _max_word_size_];

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

  list_of_vertices[0] = origin;
  origin->visited = 1;
  origin->previous = NULL;
  int j = 1;
  int i = 0;

  while (i < j && i < maximum_number_of_vertices)
  {

    hash_table_node_t *current = list_of_vertices[i];
    adjacency_node_t *adjacency_node = current->head;

    while (adjacency_node != NULL)
    {
```

```
      if (adjacency_node->vertex->visited == 0)
      {
        adjacency_node->vertex->visited = 1;
        adjacency_node->vertex->previous = current;
        list_of_vertices[j] = adjacency_node->vertex;
        j++;
      }

      adjacency_node = adjacency_node->next;
    }
    i++;
  }

  origin->last_word = list_of_vertices[j - 1];

  for (int i = 0; i < j; i++)
  {
    list_of_vertices[i]->visited = 0;
  }

  return list_of_vertices;
}
static void list_connected_component(hash_table_t *hash_table, const char *word)
{

  hash_table_node_t *node = find_word(hash_table, word, 0);

  if (node == NULL)
  {
    printf("Word not found.\n");
    return;
  }

  hash_table_node_t **list_of_vertices;
  list_of_vertices = (hash_table_node_t **)malloc((size_t)node->representative-
>number_of_vertices * sizeof(hash_table_node_t *));
  if (list_of_vertices == NULL)
  {
    fprintf(stderr, "malloc failed: out of memory\n");
    exit(1);
  }

  for (unsigned int i = 0u; i < node->representative->number_of_vertices; i++)
    list_of_vertices[i] = NULL;
```

```
  printf("Connected component %s belgons to: \n", node->word);

  list_of_vertices = breadh_first_search(node->representative->number_of_vertices,
list_of_vertices, node, NULL);

  for (int i = 0; i < node->representative->number_of_vertices; i++)
  {
    printf("[%d] %s \n", i, list_of_vertices[i]->word);
  }

  free(list_of_vertices);
}
static int largest_diameter;
static hash_table_node_t *largest_diameter_node;

static int connected_component_diameter(hash_table_node_t *node)
{
  int diameter = 0;

  hash_table_node_t **list_of_vertices;
  list_of_vertices = (hash_table_node_t **)malloc((size_t)node->representative-
>number_of_vertices * sizeof(hash_table_node_t *));
  if (list_of_vertices == NULL)
  {
    fprintf(stderr, "malloc failed: out of memory\n");
    exit(1);
  }

  list_of_vertices = breadh_first_search(node->representative->number_of_vertices,
list_of_vertices, node, NULL);

  for (unsigned int i = 0; i < node->representative->number_of_vertices; i++)
  {
    hash_table_node_t *current = list_of_vertices[i];
    int x = 0;
    while (current != NULL)
    {
      current = current->previous;
      x++;
    }
    if (x > diameter)
    {
      diameter = x;
    }
  }
```

```
  if (diameter > largest_diameter)
  {
    largest_diameter = diameter;
    largest_diameter_node = node;
  }

  free(list_of_vertices);

  return diameter;
}
static void path_finder(hash_table_t *hash_table, const char *from_word, const char *to_word)
{

  printf("path from %s to %s:\n", from_word, to_word);

  hash_table_node_t *from = find_word(hash_table, from_word, 0);
  hash_table_node_t *to = find_word(hash_table, to_word, 0);

  hash_table_node_t **list_of_vertices;
  list_of_vertices = (hash_table_node_t **)malloc((size_t)from->representative-
>number_of_vertices * sizeof(hash_table_node_t *));
  if (list_of_vertices == NULL)
  {
    fprintf(stderr, "malloc failed: out of memory\n");
    exit(1);
  }

  for (unsigned int i = 0u; i < from->representative->number_of_vertices; i++)
   list_of_vertices[i] = 0;

  list_of_vertices = breadh_first_search(from->representative->number_of_vertices,
list_of_vertices, to, from);

  hash_table_node_t *current = from;
  int x = 0;
  while (current != NULL)
  {
    printf("[%d] %s \n", x, current->word);
    current = current->previous;
    x++;
  }

  free(list_of_vertices);
}
```

```
static void graph_info(hash_table_t *hash_table)
{
  int number_of_connected_components = 0;

  hash_table_node_t *node;
  unsigned int i;

  for (i = 0u; i < hash_table->hash_table_size; i++)
  {
    for (node = hash_table->heads[i]; node != NULL; node = node->next)
    {
      if (node != NULL && node->representative == node)
      {
        number_of_connected_components++;
      }
    }
  }

  printf("Number of vertices in the graph: %d\n", hash_table->number_of_entries);
  printf("Number of edges in the graph: %d\n", hash_table->number_of_edges);
  printf("Number of connected components: %d\n", number_of_connected_components);

  unsigned int j = 0;
  int diameter = 0;
  int array[number_of_connected_components];

  for (i = 0u; i < hash_table->hash_table_size; i++)
  {
    for (node = hash_table->heads[i]; node != NULL; node = node->next)
    {
      if (node != NULL && node->representative == node)
      {
        diameter = connected_component_diameter(node);
        array[j] = diameter;
        j++;
      }
    }
  }

  int current = 0;
  for (i = 0; i < number_of_connected_components; i++)
  {
    for (j = i + 1; j < number_of_connected_components; j++)
    {
      if (array[i] > array[j])
```

```
      {
        current = array[i];
        array[i] = array[j];
        array[j] = current;
      }
    }
  }

  printf("\nNumber of connected components with a diameter of: \n");
  count = 1;
  for (i = 0; i < number_of_connected_components; i++)
  {
    if (i == number_of_connected_components - 1 && array[i] != array[i - 1])
    {
      printf("  %d : %d\n", array[i], count);
      break;
    }

    if (array[i] == array[i + 1])
    {
      count++;
    }
    else
    {
      printf("  %d : %d\n", array[i], count);
      count = 1;
    }
  }

  printf("\nLargest word ladder: \n");
  path_finder(hash_table, largest_diameter_node->word, largest_diameter_node->last_word->word);
}

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;
  clock_t tic = clock();

  // initialize hash table
  hash_table = hash_table_create();
```

```
  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);

  clock_t toc = clock();

  printf("\nHash table construction time: %0.03f seconds\n", (double)(toc - tic) /
CLOCKS_PER_SEC);

  // ask what to do
  for (;;)
  {
    fprintf(stderr, "\n");
    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 See Hash Table                \n");
    fprintf(stderr, "  4 See Colisions                 \n");
    fprintf(stderr, "  5 See graph info                \n");
    fprintf(stderr, "  6 (terminate)\n");
    fprintf(stderr, "> ");
    if (scanf("%99s", word) != 1)
      break;
    command = atoi(word);
    if (command == 1)
    {
      if (scanf("%99s", word) != 1)
        break;
      list_connected_component(hash_table, word);
```

```
    }
    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)
    {
      print_table(hash_table);
    }
    else if (command == 4)
    {
      count_colisions(hash_table);
    }
    else if (command == 5)
    {
      graph_info(hash_table);
    }
    else if (command == 6)
    {
      break;
    }
  }

  // clean up
  hash_table_free(hash_table);
  return 0;
}
```