

Licenciatura em Engenharia Informática

Algoritmos e Estruturas de Dados

Word Ladder



João Catarino
NMec: 93096

Rúben Garrido
NMec: 107927

Nuno Vieira
NMec: 107283

14 de janeiro de 2023

Índice

| | | |
|----------|--|-----------|
| 1 | Introdução | 2 |
| 2 | Representação do problema | 2 |
| 3 | Implementação | 2 |
| 3.1 | Funções de <i>hash table</i> | 2 |
| | hash_table_create | 2 |
| | hash_table_grow | 3 |
| | hash_table_free | 3 |
| | create_word_node | 3 |
| 3.2 | Funções de manipulação de grafo | 3 |
| | find_word | 3 |
| | similar_words | 3 |
| | find_representative | 4 |
| | add_edge | 4 |
| 3.3 | Funções de estudo de grafo | 4 |
| | breadth_first_search | 4 |
| | path_finder | 5 |
| | component_info | 5 |
| | graph_info | 5 |
| | connected_component_diameter | 5 |
| 3.4 | Funções de <i>queue</i> | 5 |
| | allocate_ptr_queue | 5 |
| | free_ptr_queue | 6 |
| | queue_put_hi | 6 |
| | queue_get_lo | 6 |
| 4 | Word ladders interessantes | 6 |
| 4.1 | Maior <i>word ladder</i> encontrada | 7 |
| 5 | Verificação de <i>memory leaks</i> | 8 |
| 6 | Análise do incremento da <i>hash table</i> | 9 |
| 6.1 | Explicação do código | 9 |
| 6.2 | Gráficos obtidos | 10 |
| 6.3 | Análise dos resultados | 11 |
| 7 | Código | 12 |
| 7.1 | solution_word_ladder.c | 12 |
| 7.2 | Função hash_table_grow que testa o melhor incremento | 27 |
| 7.3 | Script MATLAB que gera os gráficos para análise da hash_table_grow | 28 |



1 Introdução

Este trabalho centra-se no estudo de dicionários de palavras, interpretando cada um como um grafo não-orientado, onde cada nó representa uma palavra. Palavras que diferem num só carácter são unidas por um arco. Deste modo, é possível determinar *word ladders* entre palavras: os caminhos mais curtos entre elas, compostos por outras palavras. Por exemplo, *tudo*, *todo*, *nodo*, *nado*, *nada* é uma *word ladder* entre as palavras *tudo* e *nada*. Os objetivos deste trabalho incluem implementar uma *hash table*, a representação de um grafo não-orientado e a estrutura de dados *union find*, bem como determinar o componente complexo de uma palavra, o diâmetro de um componente conexo e a maior *word ladder* existente no dicionário fornecido.

2 Representação do problema

A implementação do grafo foi feita através de uma *linked hash table*, uma vez que esta possui uma complexidade computacional $O(1)$ para a inserção e procura de elementos. Esta complexidade computacional é importante, dado que a geração de arcos entre palavras implica verificar se todas as permutações de mudança de um carácter de um nó correspondem a palavras contidas no dicionário. Para além disso, caso a complexidade computacional fosse $O(n)$ e existisse um número avultado de palavras, a *word ladder* poderia demorar algum tempo a ser gerada.

A *linked hash table* é composta por um array de *buckets*: *linked lists* que contêm nós. Nós aos quais sejam atribuídos o mesmo índice de array pela função de dispersão são colocados no início da lista presente nesse índice. Cada nó contém uma *linked list* onde são guardadas referências aos nós adjacentes. A figura 1 mostra a estrutura da *hash table*.

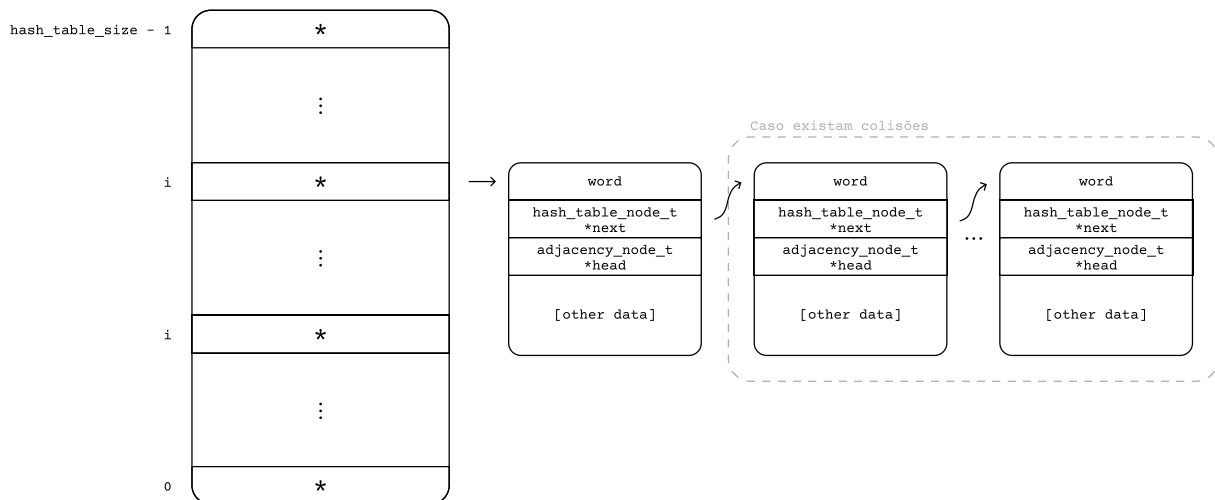


Figura 1: Estrutura da *hash table*

Os índices são calculados através do resto da divisão entre o valor retornado por uma *hash function* e o tamanho da *hash table*. A *hash function* utilizada faz uso do algoritmo CRC32 (acrónimo de *Cyclic Redundancy Check*, 32 bits).

3 Implementação

3.1 Funções de *hash table*

`hash_table_create`

Esta função aloca memória para uma estrutura de dados do tipo definido `hash_table_t`, que representa a *hash table*, e para o seu *array* interno `heads` que irá conter as entradas da tabela.

O tamanho inicial da tabela é definido no macro `_hash_table_init_size_` e as suas entradas são inicializadas a zero de forma implícita pela função `calloc`. No final, é guardado o tamanho inicial da tabela na estrutura de dados. A função termina o programa se uma das alocações não for bem sucedida.

hash_table_grow

`hash_table_grow` é a função responsável por aumentar o tamanho da *hash table*. A sua condição de incremento depende do rácio entre o tamanho desta e o número de colisões, onde, caso seja superior a 5, verificar-se-á o incremento. A função inicializa uma nova tabela (com um tamanho igual ao dobro da anterior) usando a função `calloc` e, para cada *node* desta, recalcula o seu índice, através da função `crc32`, e insere-o na nova tabela. Por fim, os endereços para o array `heads` e para o tamanho da tabela (`hash_table_size`) são atualizados.

hash_table_free

Esta função desaloca a memória previamente reservada para a *hash table* e outras estruturas de dados nela contidas. Esta começa por iterar sobre todas as posições do *array*, chamando a função `free_hash_llist`, libertando a *linked list* nessa posição. Por sua vez, esta irá iterar sobre todos os nós da lista através do ponteiro `next` e chamar a função `free_hash_table_node` sobre cada nó, que irá primeiro libertar a lista de adjacência com a função `free_adjacency_llist` e depois o próprio nó. A função `free_adjacency_llist` simplesmente itera sobre todos os membros da lista ligada e liberta cada um deles.

create_word_node

Cria um novo nó de *hash table* com a palavra fornecida como argumento. Este nó, cuja memória alocada é obtida através da função `allocate_hash_table_node`, é inicializado com a palavra e alguns parâmetros predefinidos. Por fim, a função retorna o endereço do *node* recém-criado.

3.2 Funções de manipulação de grafo

find_word

Função dedicada à procura e inserção de elementos na *hash table*. É calculado o índice, que resulta do resto da divisão entre o resultado da aplicação da *hash function* à palavra desejada e o tamanho máximo da tabela. Esta operação distribui o índice da palavra entre 0 e o tamanho da tabela. De seguida, a função itera sobre a lista ligada na posição calculada, procurando um nó igual através da função `strcmp`, e retorna-o se for o caso. Caso a palavra não exista e a opção `insert_if_not_found` seja verdadeira, o programa aloca um novo nó para a palavra através da função `create_word_node` e coloca-o na lista ligada, atualizando os valores estatísticos na *hash table*.

similar_words

É responsável por encontrar palavras semelhantes à palavra fornecida como argumento. A verificação é baseada em caracteres *Unicode*, onde, para cada caracter da palavra, este é substituído pelos vários caracteres aceites. Caso a nova palavra exista na *hash table*, é efetuada uma chamada à função `add_edge` (ver secção 3.2). A função termina quando não existirem mais caracteres para substituir.

find_representative

Função responsável pela determinação do nó representante do componente conexo de um nó requisitado. Antes de devolver o resultado, a função atualiza os representantes dos nós percorridos anteriormente.

O seu funcionamento consiste em iterar sobre o representante do nó dado como argumento recursivamente até que o representante de um nó seja ele próprio. Após determinar o representante, esta parte novamente do nó passado como argumento e altera o representante de forma recursiva para o anteriormente determinado até chegar a este novamente. Desta forma, o caminho para o representante é simplificado e as chamadas posteriores com o mesmo nó como argumento serão mais rápidas.

add_edge

Esta função tenta adicionar um arco entre um nó existente e uma palavra que pode não existir, para além de implementar os aspetos da estrutura de dados *Union-Find*. Esta começa por procurar a *string* dada no grafo, retornando caso não a encontre. Caso contrário, procura o segundo nó na lista de adjacência do primeiro e retorna se for o caso, para garantir que o arco não é duplicado. Ao adicionar o arco, é incrementado o contador de arcos da tabela e são adicionados nós de adjacência às tabelas de adjacência das palavras.

A função **insert_edge** trata de alocar o nó de adjacência para cada vértice e de os colocar nas listas respetivas. Neste sentido, a função trata de executar a operação *Union* entre os conjuntos (componentes conexos) de cada vértice. Primeiro, são determinados os representantes de cada um através da função **find_representative** vista anteriormente. Se estes forem diferentes, a operação ocorre. Para minimizar o tamanho do caminho de representantes, o nó com menos vértices na altura da operação é escolhido como o representante do novo componente conexo. Finalmente, são somados os números de vértices e arcos, é decrementado o número de componentes e é atualizado o tamanho do maior componente.

3.3 Funções de estudo de grafo**breath_first_search**

A função **breath_first_search**, cujos argumentos são o número máximo de vértices, uma lista de vértices, o array de origem e o ponto de destino, tem como objetivo retornar o número de vértices visitados. Se for especificado o destino, será possível obter o caminho mais curto até este seguindo recursivamente o seu ponteiro **previous**. Para este efeito, é utilizado o algoritmo *Breadth-first search*. Este percorre uma árvore nível a nível, garantindo o menor percurso para cada nó visitado. Foi implementada para este caso uma *queue* circular, que garante que o tamanho máximo desta é constante.

Em primeiro lugar, é efetuada uma chamada à função **allocate_ptr_queue** – que devolve uma *queue* –, e uma chamada à função **queue_put_hi**, que insere o nó de partida na *queue*. De seguida, dentro de um ciclo **while**, a função visita todos os nós na *queue*, retirando-os do início da *queue*. A sua variável **visited** é incrementada para marcar o nó como visitado e este é colocado na lista de nós visitados. Por cada nó visitado, são adicionados os seus nós adjacentes por visitar ao final da *queue*, guardando neles o valor do nó visitado em **previous**, bem como a profundidade do nível anterior em **visited**. Ao terminar a iteração, a função retorna -1 caso não tenha encontrado o destino (se for fornecido). Caso contrário, devolve o comprimento da lista de nós visitados.

Nota importante: o valor **visited** nesta implementação guarda a distância do nó até à raiz. Isto é útil na determinação do diâmetro na função **connected_component_diameter**, por tornar desnecessário percorrer o caminho entre o nó de partida e o nó destino para calcular a distância.



path_finder

Esta função tenta determinar o caminho mais curto entre duas palavras (uma *word ladder*) , imprimindo a sequência. Para tal, em primeiro lugar, é necessário obter os nós associados a cada palavra, utilizando a função `find_word`. Caso um não exista, é impressa uma mensagem de erro. Depois, todos os nós da tabela de *hash* são marcados como não visitados (variável `visited` de cada nó posta a -1). O grafo está agora pronto para ser estudado. É chamada a função `breath_first_search` com o argumento `list_of_vertices` nulo, visto que o caminho será obtido ao seguir os ponteiros `previous` a partir do destino. Como o caminho obtido desta forma será invertido, ou seja, irá da palavra destino até à de partida, a função anterior é chamada com os argumentos `from_word` e `to_word` alternados. Esta chamada devolve -1 caso as palavras não estiverem ligadas, e é impresso uma mensagem de erro neste caso. Caso contrário, o caminho existe e é impresso seguindo o ponteiro `previous` do nó `from_word` até que seja nulo.

component_info

Função que imprime dados de um componente conexo associado a uma palavra dada como argumento. Primeiro verifica que a palavra existe e depois obtém o nó representante do seu componente conexo. Este contém os dados atualizados do componente. São impressos a palavra representante, o número de vértices, o número de laços e o diâmetro do componente.

graph_info

Imprime a seguinte informação sobre o grafo: o número de nós, o número de laços, o número de nós de adjacência (apenas para verificar que são exatamente o dobro do número de laços), o número de componentes conexos, o tamanho do maior componente, o maior diâmetro do grafo e a maior *word ladder* correspondente.

connected_component_diameter

Determina o diâmetro do componente conexo ao qual o nó dado pertence. Atualiza o valor máximo global, bem como o caminho respetivo se for o caso. Para tal, a função itera sobre todos os nós do componente conexo, determina as suas excentricidades e guarda a maior encontrada. Em termos práticos, isto significa, em primeiro lugar, obter uma lista de todos os nós através da função `breath_first_search`. Por cada nó, é feita uma procura com a função `breath` partindo deste. Como esta função percorre o grafo nível a nível, o último nó inserido na lista de nós visitados encontra-se no nível mais profundo, e portanto será um dos nós mais distantes do anterior. Se a distância ao último nó da lista, obtido a partir da variável `visited`, for a maior determinada até agora, é guardada. O último valor a ser guardado após todas as pesquisas a partir de todos os vértices, é o diâmetro do componente conexo. Tendo determinado o diâmetro, a função verifica se este é o maior a nível global, guardando o seu valor e o seu caminho associado.

3.4 Funções de queue**allocate_ptr_queue**

Esta função aloca memória para uma *queue* de ponteiros. Esta operação começa por alocar memória para a estrutura de metadados da *queue*, que contém as variáveis necessárias para o manuseamento de uma *queue*. Posteriormente, é alocado o *array* circular de ponteiros. Finalmente, são inicializadas as variáveis de controlo da estrutura de dados e o seu ponteiro é devolvido.

free_ptr_queue

A função que liberta a memória alocada para a *queue*. Isto implica apenas libertar o seu *array* circular e finalmente a própria estrutura.

queue_put_hi

Implementação da operação *enqueue*, onde um elemento, neste caso um ponteiro, é adicionado ao final da fila. A função começa por executar um *assert* para assegurar que o tamanho da fila não é ultrapassado. Depois, é inserido o novo elemento na posição *hi*. Este valor não representa na verdade o final da fila, mas sim a posição imediatamente a seguir. O valor é agora incrementado e assume o valor do resto da sua divisão pelo tamanho máximo da *queue*. Isto implica que para valores menores que o maior índice possível, *hi* mantém-se, caso contrário, volta à posição 0, garantindo a circularidade. Finalmente, o tamanho da fila é incrementado.

queue_get_lo

Implementação da operação *dequeue*, onde o elemento inicial da fila é removido e devolvido. A operação começa por um *assert*, para garantir que o programa não tenta remover um elemento de uma fila vazia. De seguida, o valor na posição *lo* é guardado numa variável temporária. Isto é feito porque o início da fila pode “saltar” para o outro lado após a sua incrementação. Tendo o valor guardado, é atualizada a posição *lo* da mesma forma como na função **queue_put_hi**, e é decrementado o tamanho da fila. No final, é devolvido o valor guardado.

4 Word ladders interessantes

- De “Terra” a “Marte”:

1. Terra
2. perra
3. parra
4. parta
5. parte
6. Marte

- Do “tudo” ao “nada”:

1. tudo
2. todo
3. nodo
4. nado
5. nada

- De “novo” a “byte”:

1. novo
2. noto
3. boto
4. bote
5. byte

- De “mundo” a “clima”:

1. mundo
2. mondo
3. monto
4. conto
5. coito
6. coita
7. coima
8. clima



4.1 Maior *word ladder* encontrada

| | | |
|--------------|---------------|--------------|
| 1. visse-te | 32. bastaria | 63. exarares |
| 2. viste-te | 33. bostaria | 64. exararas |
| 3. veste-te | 34. tostaria | 65. exaradas |
| 4. deste-te | 35. tontaria | 66. exarados |
| 5. despe-te | 36. contaria | 67. exaramos |
| 6. desperte | 37. coutaria | 68. exaremos |
| 7. desperto | 38. chutaria | 69. extremos |
| 8. desporto | 39. chuparia | 70. entremos |
| 9. desponto | 40. chaparia | 71. enteemos |
| 10. desponte | 41. chavaria | 72. enfeemos |
| 11. desconte | 42. cravaria | 73. enfermos |
| 12. descente | 43. crivaria | 74. enfermas |
| 13. descende | 44. privaria | 75. enformas |
| 14. descenda | 45. provaria | 76. enfornas |
| 15. despenda | 46. proveria | 77. encornas |
| 16. despensa | 47. preveria | 78. encarnas |
| 17. dispensa | 48. preteria | 79. escarnas |
| 18. dispenso | 49. preteris | 80. escarpas |
| 19. distenso | 50. preteres | 81. escalpas |
| 20. distendo | 51. premeres | 82. escaldas |
| 21. distando | 52. tremeres | 83. escaldar |
| 22. discando | 53. tremares | 84. espaldar |
| 23. riscando | 54. tramares | 85. espalhar |
| 24. rascando | 55. aramares | 86. espalhas |
| 25. lascando | 56. afamares | 87. espelhas |
| 26. lassando | 57. afanares | 88. espelhos |
| 27. passando | 58. afinares | 89. espelhou |
| 28. pastando | 59. afileares | 90. espalhou |
| 29. bastando | 60. axilares | 91. empalhou |
| 30. bastardo | 61. exilares | 92. empaleou |
| 31. bastarda | 62. exalares | |



5 Verificação de *memory leaks*

Para verificar se existem *memory leaks* no programa criado, foi utilizado o *Valgrind*, um programa constituído por um conjunto de ferramentas para a deteção de erros de memória e de *threading*. Para o efeito, foi utilizado o *Memcheck*, que é uma ferramenta para deteção de erros de memória.

Obteve-se o seguinte resultado, para o programa `solution_word_ladder.c`, com o ficheiro `wordlist-four-letters.txt` como argumento:

```
==500745== Memcheck, a memory error detector
==500745== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==500745== Using Valgrind-3.19.0 and LibVEX; rerun with -h for copyright info
==500745== Command: ./solution_word_ladder wordlist-four-letters.txt
==500745==
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 WORD      (list component info)
  4           (list hash table info)
  5           (list graph info)
  0           (terminate)
> 0
==500745==
==500745== HEAP SUMMARY:
==500745==   in use at exit: 0 bytes in 0 blocks
==500745==   total heap usage: 27,700 allocs, 27,700 frees, 60,322,800 bytes
==500745==         allocated
==500745==
==500745== All heap blocks were freed -- no leaks are possible
==500745==
==500745== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Assim, concluímos que não existem *memory leaks* no programa.

6 Análise do incremento da *hash table*

Por padrão, o tamanho inicial da *hash table* é 1000. No entanto, quando o número de entradas começa a ser significativo, começam a surgir colisões, o que implica uma perda da complexidade computacional $O(1)$. Para evitar este problema, quando o rácio entre o tamanho da *hash table* e o número de colisões é superior a 5, o tamanho da *hash table* é incrementado, através da função `hash_table_grow`, que recebe como argumento a referida *hash table*.

Contudo, a escolha do fator de incremento deve ser ponderada, já que, se for muito pequeno, o número de colisões diminui pouco, e se for muito grande, existe demasiada memória alocada não utilizada, o que leva a um desperdício de recursos. É esta escolha que pretendemos analisar.

6.1 Explicação do código

Foi desenvolvida uma nova função `hash_table_grow`, num programa à parte, em que, após ser verificada a condição de incremento (rácio entre o tamanho da *hash table* e o número de colisões), é percorrido um ciclo `for`, onde são testados vários valores de `j` (fator de incremento).

```
if (hash_table->number_of_collisions > 0 && (hash_table->
    hash_table_size / hash_table->number_of_collisions) < 5)
{
    printf("\nFinding best j. Current hash_table_size is %u.\n",
        hash_table->hash_table_size);
    printf("    j    | new size | memory | free m | colnum\n");
    for (j = 1.1; j < 3; j += 0.005)
    {
```

Dentro deste ciclo, e após inicializar algumas variáveis (p.e., a nova *hash table* temporária), surgem dois novos ciclos `for`.

No primeiro `for`, é percorrida a *hash table* inicial, onde, para cada `node`, é calculado o novo índice, através do resto da divisão entre o valor retornado da função `crc32` e o tamanho da *hash table*. Após este cálculo, é verificada a existência de colisões no índice calculado anteriormente, e, caso existam, é incrementado o valor de `colnum`. Por fim, é associado o nó atual à nova *hash table*, na localização definida pelo índice.

```
for (i=0; i < hash_table->hash_table_size; i++)
{
    for (node = hash_table->heads[i]; node; node = next)
    {
        test_new_key = crc32(node->word) % test_new_size;
        next = node->next;
        if (test_new_table[test_new_key])
        {
            colnum++;
        }
        test_new_table[test_new_key] = node;
    }
}
```

No segundo `for`, é percorrida a nova *hash table*, onde é verificado o número de entradas livres desta. Caso `test_new_table[k]` seja nulo, significa que a posição `k` da *hash table* está livre, e, portanto, é incrementado o valor de `free_entries`.

```
for (k=0; k < test_new_size; k++) {
    if (!test_new_table[k]) {
        free_entries++;
    }
}
```

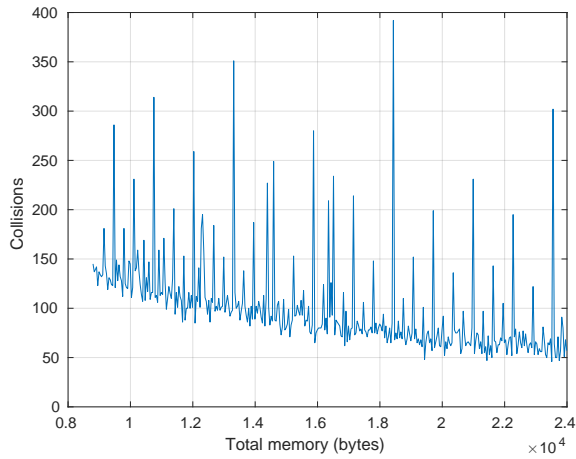
Por último, é impressa uma linha com os dados obtidos, nomeadamente o fator de incremento j , o novo tamanho da *hash table* `test_new_size`, a memória total ocupada `test_new_size * sizeof(hash_table_node_t *)`, a memória ocupada por entradas livres `free_entries * sizeof(hash_table_node_t *)` e o número de colisões `colnum`.

```
printf("%3.3f | %8u | %6lu | %6lu | %6u\n", j, test_new_size,
test_new_size * sizeof(hash_table_node_t *), free_entries * sizeof(
hash_table_node_t *), colnum);
```

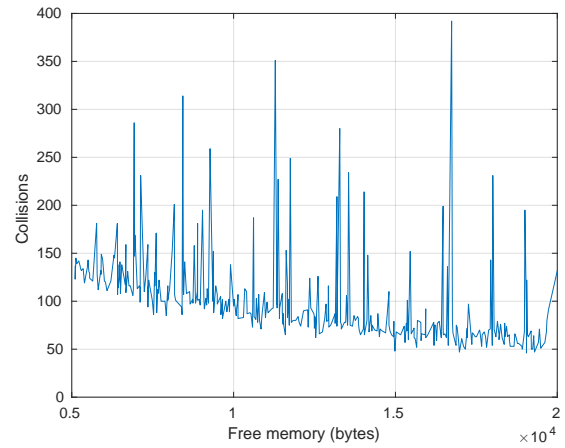
6.2 Gráficos obtidos

Através do MATLAB, foi possível obter um conjunto de gráficos, que relacionam colisões com memória livre e memória total. O script, disponível na secção 7.3, obtém os dados através de um ficheiro de texto, que contém a tabela imprimida pelo programa de teste.

Os gráficos em questão incidem sobre o primeiro incremento, onde o tamanho atual da *hash table* é 1000.

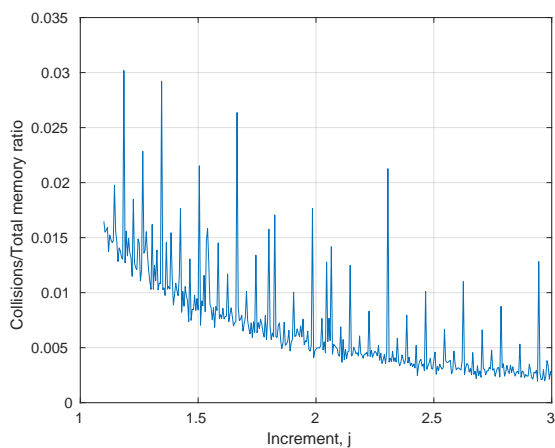


(a) Número de colisões em função da memória total.

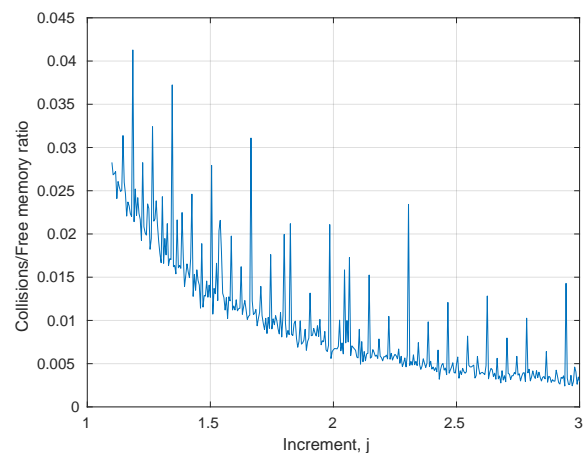


(b) Número de colisões em função da memória livre.

Figura 2: Número de colisões em função da memória.



(a) Rácio colisões/memória total.



(b) Rácio colisões/memória livre.

Figura 3: Rácio colisões/memória em função do incremento.

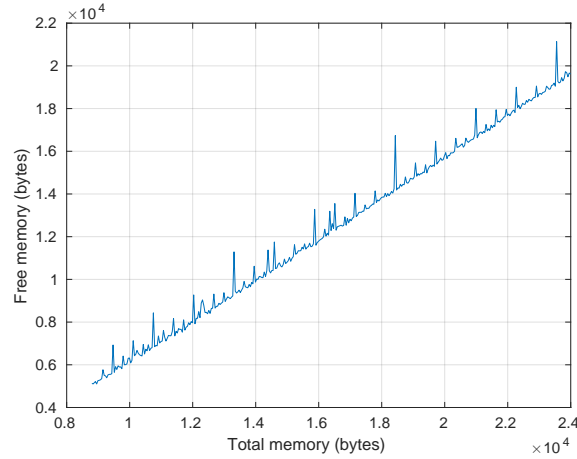


Figura 4: Memória livre em função da memória total.

6.3 Análise dos resultados

Em todos os gráficos, é possível observar irregularidades, associadas a uma maior ou menor quantidade de colisões. Isto deve-se ao facto de a *hash function* não ser perfeita e portanto, consoante o valor de j , o índice associado a cada **node** ser diferente.

No entanto, é possível observar que, em geral, o número de colisões diminui com o aumento de memória (ou seja, com incrementos maiores), tal como seria esperado. A relação entre o número de colisões e a memória livre segue a mesma tendência. Em ambos os gráficos, verifica-se uma tendência aproximadamente linear.

Quanto aos rácios colisões/memória em função do incremento, observa-se em ambos os gráficos uma curva descendente, do tipo $a \times x^b$, com $-2 < b < -1$. Por este motivo, verifica-se uma diferença mais acentuada no eixo das ordenadas para incrementos menores do que para incrementos maiores. Assim, considera-se que o melhor incremento é o que apresenta um valor de b mais próximo de 2, já que, a partir desse valor, o rácio tende a ser mais constante. Por outro lado, a similaridade entre rácios explica-se pelo facto de a relação entre a memória livre e a memória total ser aproximadamente linear, com um declive próximo de 1 (ver figura 4).

No que concerne à relação entre a memória livre e a memória total (ambas em *bytes*), apesar das irregularidades, é possível efetuar uma regressão linear, onde se obtém a equação $y = 0.9583x - 3357$.

Assim, uma vez que os gráficos não são completamente conclusivos quanto ao melhor fator de incremento, escolhemos utilizar o valor 2, já que este constitui um equilíbrio entre o número de colisões e a memória utilizada.

7 Código

7.1 solution_word_ladder.c

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <sys/param.h>

//
// static configuration
//

#define _max_word_size_ 32
#define _hash_table_init_size_ 1000

//
// 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;
typedef struct ptr_queue_s ptr_queue_t;

struct ptr_queue_s
{
    void **circular_array;
    unsigned int hi;
    unsigned int lo;
    unsigned int size;
    unsigned int max_size;
    int full;
};

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

struct hash_table_node_s
{
    // the hash table data
    char word[_max_word_size_]; // the word
    hash_table_node_t *next; // next hash table linked list node
    // the vertex data
    adjacency_node_t *head; // head of the linked list of
    adjacency edges
    int visited; // visited status (while not in use
    , keep it at 0)
    hash_table_node_t *previous; // breadth-first search parent
    // the union find data

```



```

hash_table_node_t *representative; // the representative of the
    connected component this vertex belongs to
int number_of_vertices;           // number of vertices of the
    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)
int component_diameter;           // only valid for the representative
    node
};

struct hash_table_s
{
    unsigned int hash_table_size;           // the size of the hash table
        array
    unsigned int largest_component_size;    //size of the biggest component
        (passed on to breadh_first as max list size)
    unsigned int number_of_entries;        // the number of entries in the
        hash table
    unsigned int number_of_collisions;     // the total of entries inserted on
        an occupied index
    unsigned int number_of_edges;          // number of edges (for information
        purposes only)
    unsigned int number_of_edge_nodes;     // number of edges (for information
        purposes only)
    unsigned int number_of_components;     // number of connected components
    hash_table_node_t **heads;             // the heads of the linked lists
};

//
// allocation and deallocation of queue
//

static ptr_queue_t *allocate_ptr_queue(unsigned int max_size)
{
    ptr_queue_t *queue = (ptr_queue_t *)malloc(sizeof(ptr_queue_t));
    if(queue == NULL)
    {
        fprintf(stderr, "allocate_ptr_queue: out of memory\n");
        exit(1);
    }
    queue->circular_array = (void **)malloc(sizeof(void *) * max_size);
    if(queue->circular_array == NULL)
    {
        fprintf(stderr, "allocate_ptr_queue->circular_array: out of memory\n"
        );
        free(queue);
        exit(1);
    }
    queue->max_size = max_size;
    queue->size = 0;
    queue->full = 0;
    queue->hi = 0;
    queue->lo = 0;
    return queue;
}

```



```

static void free_ptr_queue(ptr_queue_t *queue)
{
    free(queue->circular_array);
    free(queue);
}

//
// queue methods
//

static void queue_put_hi(ptr_queue_t *queue, void *ptr)
{
    assert(queue->size < queue->max_size);
    queue->circular_array[queue->hi] = ptr;
    queue->hi = (queue->hi + 1) % queue->max_size;
    queue->size++;
}

static void *queue_get_lo(ptr_queue_t *queue)
{
    assert(queue->size > 0);
    void *ret = queue->circular_array[queue->lo];
    queue->lo = (queue->lo + 1) % queue->max_size;
    queue->size--;
    return ret;
}

//
// 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_llist(adjacency_node_t *head)
{
    adjacency_node_t *next;
    for (; head; head = next)
    {
        next = head->next;
        free(head);
    }
}

static void free_hash_table_node(hash_table_node_t *node)
{
    free_adjacency_llist(node->head);
    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_llist(hash_table_node_t *head)
{
    hash_table_node_t *next;
    for (; head; head = next)
    {
        next = head->next;
        free_hash_table_node(head);
    }
}

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

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

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

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

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

    hash_table = (hash_table_t *)calloc(1, sizeof(hash_table_t));

```




```

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

static void hash_table_info(hash_table_t *hash_table)
{
    printf("Entries: %u\nCollisions: %u\nSize: %u\n",
        hash_table->number_of_entries,
        hash_table->number_of_collisions,
        hash_table->hash_table_size);
}

static void hash_table_grow(hash_table_t *hash_table)
{
    unsigned int    new_size;
    unsigned int    new_key;
    unsigned int    i;
    hash_table_node_t **new_table;
    hash_table_node_t *next;
    hash_table_node_t *node;
    // Determine size_inc based on collision count
    if (hash_table->number_of_collisions > 0 && (hash_table->
        hash_table_size / hash_table->number_of_collisions) < 5)
    {
        new_size = hash_table->hash_table_size * 2;
        new_table = (hash_table_node_t **)calloc(new_size, sizeof(
            hash_table_node_t *));
        if (!new_table)
        {
            fprintf(stderr, "hash_table_grow: out of memory\n");
            exit(1);
        }
        hash_table->number_of_collisions = 0;
        for (i=0; i < hash_table->hash_table_size; i++)
            for (node = hash_table->heads[i]; node; node = next)
            {
                new_key = crc32(node->word) % new_size;
                next = node->next;
                node->next = new_table[new_key];
                if (node->next)
                    hash_table->number_of_collisions++;
                new_table[new_key] = node;
            }
        free(hash_table->heads);
        hash_table->heads = new_table;
        hash_table->hash_table_size = new_size;
    }
}

```



```

    }
}

static void hash_table_free(hash_table_t *hash_table)
{
    for (unsigned int i = 0; i < hash_table->hash_table_size; i++)
        if (hash_table->heads[i])
            free_hash_llist(hash_table->heads[i]);
    free(hash_table->heads);
    free(hash_table);
}

static hash_table_node_t *create_word_node(const char *word)
{
    hash_table_node_t *node = allocate_hash_table_node();
    node->representative = node;
    node->visited = -1;
    node->number_of_vertices = 1;
    node->number_of_edges = 0;
    node->previous = NULL;
    node->next = NULL;
    node->head = NULL;
    strcpy(node->word, word);
    return node;
}

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;
    node = hash_table->heads[i];
    while (node)
    {
        if (strcmp(node->word, word) == 0)
            return node;
        node = node->next;
    }
    if (insert_if_not_found)
    {
        node = create_word_node(word);
        if (hash_table->heads[i])
            hash_table->number_of_collisions++;
        node->next = hash_table->heads[i];
        hash_table->heads[i] = node;
        hash_table->number_of_components++;
        hash_table->number_of_entries++;
        hash_table_grow(hash_table);
    }
    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;

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

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

static void insert_edge(hash_table_t *hash_table, hash_table_node_t *
    from, hash_table_node_t *to)
{
    adjacency_node_t *link;

    link = allocate_adjacency_node();
    link->vertex = to;
    link->next = from->head;
    from->head = link;
    hash_table->number_of_edge_nodes++;
}

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)
        return;
    for (link = from->head; link && link->vertex != to; link = link->next)
        ;
    if (link)
        return;
    hash_table->number_of_edges++;
    insert_edge(hash_table, from, to);
    insert_edge(hash_table, to, from);

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

    from_representative->number_of_edges++;
    if (from_representative != to_representative)
    {
        unsigned int vert_sum = from_representative->number_of_vertices +
            to_representative->number_of_vertices;
        unsigned int edge_sum = from_representative->number_of_edges +
            to_representative->number_of_edges;
        int cond = to_representative->number_of_vertices >
            from_representative->number_of_vertices;
    }
}

```

```

    hash_table_node_t *new_rep = cond ? from_representative :
    to_representative;
    new_rep->number_of_vertices = vert_sum;
    new_rep->number_of_edges = edge_sum;
    (cond ? to_representative : from_representative)->representative =
    new_rep;
    hash_table->number_of_components--;
    if (vert_sum > hash_table->largest_component_size)
        hash_table->largest_component_size = vert_sum;
}
}

//
// generates a list of similar words and calls the function add_edge for
// each one (done)
//
// man utf8 for details on the utf8 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) != 0
b10000000)
            {
                fprintf(stderr, "break_utf8_string: unexpected UTF-8 character\n"
);
                exit(1);
            }
            *(individual_characters++) = ((byte0 & 0b00011111) << 6) | (byte1
& 0b00111111); // utf8 -> unicode
        }
    }
    *individual_characters = 0; // mark the end!
}

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

    while(*individual_characters != 0)
    {
        code = *(individual_characters++);
        if(code < 0x80)
            *(word++) = (char)code;
        else if(code < (1 << 11))

```



```

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

static void similar_words(hash_table_t *hash_table, hash_table_node_t *
  from)
{
  static const int valid_characters[] =
  { // unicode!
    0x2D,
    // -
    0x41, 0x42, 0x43, 0x44, 0x45, 0x46, 0x47, 0x48, 0x49, 0x4A, 0x4B, 0x4C, 0x4D,
    // A B C D E F G H I J K L M
    0x4E, 0x4F, 0x50, 0x51, 0x52, 0x53, 0x54, 0x55, 0x56, 0x57, 0x58, 0x59, 0x5A,
    // N O P Q R S T U V W X Y Z
    0x61, 0x62, 0x63, 0x64, 0x65, 0x66, 0x67, 0x68, 0x69, 0x6A, 0x6B, 0x6C, 0x6D,
    // a b c d e f g h i j k l m
    0x6E, 0x6F, 0x70, 0x71, 0x72, 0x73, 0x74, 0x75, 0x76, 0x77, 0x78, 0x79, 0x7A,
    // n o p q r s t u v w x y z
    0xC1, 0xC2, 0xC9, 0xCD, 0xD3, 0xDA,
    // Á Â É Í Ñ Ú
    0xE0, 0xE1, 0xE2, 0xE3, 0xE7, 0xE8, 0xE9, 0xEA, 0xED, 0xEE, 0xF3, 0xF4, 0xF5, 0
    xFA, 0xFC, // à á â ã ç è é ê í î ó ô õ ú ü
    0
  };
};
int i, j, k, individual_characters[_max_word_size_];
char new_word[2 * _max_word_size_];

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

//
// breadth-first search (to be done)
//

```



```

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

static int breadth_first_search(unsigned int maximum_number_of_vertices,
    hash_table_node_t **list_of_vertices, hash_table_node_t *origin,
    hash_table_node_t *goal)
{
    unsigned int    list_len;
    hash_table_node_t *node;
    adjacency_node_t *link;
    ptr_queue_t     *queue;

    queue = allocate_ptr_queue(maximum_number_of_vertices);

    list_len = 0;
    queue_put_hi(queue, origin);
    while (queue->size > 0)
    {
        node = queue_get_lo(queue);
        node->visited++;
        if (list_of_vertices)
            list_of_vertices[list_len] = node;
        list_len++;
        if (node == goal)
            break;
        for(link = node->head; link && list_len < maximum_number_of_vertices
; link = link->next)
        {
            if (link->vertex->visited == -1)
            {
                link->vertex->visited = node->visited;
                link->vertex->previous = node;
                queue_put_hi(queue, link->vertex);
            }
        }
    }
    free_ptr_queue(queue);
    if (goal && goal != node)
        return -1;
    return list_len;
}

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

static void mark_all_vertices(hash_table_t *hash_table)
{
    hash_table_node_t *node;

    for(unsigned int i = 0; i < hash_table->hash_table_size; i++)
        for(node = hash_table->heads[i]; node != NULL; node = node->next)
            node->visited = -1;
}

```



```

static void list_connected_component(hash_table_t *hash_table, const char
    *word)
{
    hash_table_node_t *origin, *rep;
    hash_table_node_t **list;
    unsigned int      list_len;

    origin = find_word(hash_table, word, 0);
    if (!origin)
    {
        printf("\nWord not found: %s\n", word);
        return;
    }

    mark_all_vertices(hash_table);
    rep = find_representative(origin);
    list = malloc(rep->number_of_vertices * sizeof(hash_table_node_t *));
    list_len = breadth_first_search(rep->number_of_vertices, list, origin,
        NULL);
    for (unsigned int i=0; i < list_len; i++)
        printf("%s\n", list[i]->word);
    free(list);
}

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

static int largest_diameter;
static hash_table_node_t **largest_diameter_example;

static int connected_component_diameter(hash_table_node_t *node)
{
    int      diameter;
    unsigned int      j, i, comp_len, search_len;
    hash_table_node_t **comp_nodes, **node_list, *chain_start, *chain_end;

    diameter = 0;
    chain_start = chain_end = NULL;
    comp_nodes = calloc(node->representative->number_of_vertices, sizeof(
        hash_table_node_t *));
    comp_len = breadth_first_search(node->representative->
        number_of_vertices, comp_nodes, node, NULL);
    for (i = 0; i < comp_len; i++)
    {
        for (j = 0; j < comp_len; j++)
            comp_nodes[j]->visited = -1;
        node_list = calloc(node->representative->number_of_vertices, sizeof(
            hash_table_node_t *));
        search_len = breadth_first_search(node->representative->
            number_of_vertices, node_list, comp_nodes[i], NULL);
        if (node_list[search_len - 1]->visited >= diameter)
        {
            diameter = node_list[search_len - 1]->visited;
            chain_start = comp_nodes[i];
            chain_end = node_list[search_len - 1];
        }
    }
}

```



```

    free(node_list);
}
if (diameter > largest_diameter)
{
    largest_diameter = diameter;
    if (largest_diameter_example)
        free(largest_diameter_example);
    largest_diameter_example = calloc(diameter, sizeof(hash_table_node_t
    *));
    i = diameter;
    for (node = chain_end; node != chain_start; node = node->previous)
        largest_diameter_example[--i] = node;
}
free(comp_nodes);
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;
    int i, list_len;

    // switch from with to in order to print path in correct order
    from = find_word(hash_table, from_word, 0);
    to = find_word(hash_table, to_word, 0);

    if (!from)
    {
        fprintf(stderr, "\nWord not found: %s\n", from_word);
        return ;
    }
    if (!to)
    {
        fprintf(stderr, "\nWord not found: %s\n", to_word);
        return ;
    }

    mark_all_vertices(hash_table);
    list_len = breadth_first_search(find_representative(to)->
        number_of_vertices, NULL, to, from);
    if (list_len == -1)
        fprintf(stderr, "Words are not connected\n");
    else
    {
        i = 0;
        for(; from && from != to; from = from->previous)
            printf(" [%d] %s\n", i++, from->word);
        printf(" [%d] %s\n", i++, from->word);
    }
}

```




```

static void component_info(hash_table_t *hash_table, char *word)
{
    hash_table_node_t *origin, *rep;

    origin = find_word(hash_table, word, 0);
    if (!origin)
        return (void)fprintf(stderr, "\nWord not found.\n");
    rep = find_representative(origin);
    printf("\nRepresentative: %s\nVertices: %u\nEdges: %u\nDiameter: %u\n",
        ,
        rep->word,
        rep->number_of_vertices,
        rep->number_of_edges,
        rep->component_diameter);
}

//
// some graph information (optional)
//

static void graph_info(hash_table_t *hash_table)
{
    printf("\nGraph info:\n\nNodes\t\t\t: %u\nEdges\t\t\t: %u\nEdge nodes\
\t\t: %u\nComponents\t\t: %u\n\
Largest component size : %u\nLargest component\t: %s\nLargest diameter\
\t: %d\n",
        hash_table->number_of_entries,
        hash_table->number_of_edges,
        hash_table->number_of_edge_nodes,
        hash_table->number_of_components,
        hash_table->largest_component_size,
        find_representative(largest_diameter_example[0])->word,
        largest_diameter);
    printf("Largest word chain:\n");
    for(unsigned int i=0; i < (unsigned int)largest_diameter; i++)
        printf(" [%d] %s\n", i, largest_diameter_example[i]->word);
}

//
// 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, *rep;
    unsigned int i;
    int command;
    FILE *fp;

    largest_diameter_example = NULL;
    largest_diameter = 0;
    // 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 = 0; i < hash_table->hash_table_size; i++)
    for(node = hash_table->heads[i]; node != NULL; node = node->next)
        similar_words(hash_table, node);
for(i = 0; i < hash_table->hash_table_size; i++)
    for(node = hash_table->heads[i]; node != NULL; node = node->next)
    {
        if (node->visited == -1)
        {
            rep = find_representative(node);
            rep->component_diameter = connected_component_diameter(node);
        }
    }
// ask what to do
for(;;)
{
    fprintf(stderr, "\nYour wish is my command:\n");
    fprintf(stderr, "  1 WORD          (list the connected component WORD\n\nbelongs to)\n");
    fprintf(stderr, "  2 FROM TO      (list the shortest path from FROM to\n\nTO)\n");
    fprintf(stderr, "  3 WORD          (list component info)\n");
    fprintf(stderr, "  4              (list hash table info)\n");
    fprintf(stderr, "  5              (list graph info)\n");
    fprintf(stderr, "  0              (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)
    {
        if(scanf("%99s", word) != 1)
            break;
        component_info(hash_table, word);
    }
}
```



```
    else if(command == 4)
        hash_table_info(hash_table);
    else if(command == 5)
        graph_info(hash_table);
    else if(command == 0)
        break;
}
// clean up
hash_table_free(hash_table);
if (largest_diameter_example)
    free(largest_diameter_example);
return 0;
}
```



7.2 Função hash_table_grow que testa o melhor incremento

```
static void hash_table_grow(hash_table_t *hash_table)
{
    unsigned int    i;
    double          j;
    unsigned int    k;
    unsigned int    test_new_size;
    unsigned int    test_new_key;
    hash_table_node_t *next;
    hash_table_node_t *node;
    hash_table_node_t **test_new_table;
    unsigned int    colnum;
    unsigned int    free_entries;

    if (hash_table->number_of_collisions > 0 && (hash_table->
        hash_table_size / hash_table->number_of_collisions) < 5)
    {
        printf("\nFinding best j. Current hash_table_size is %u.\n",
            hash_table->hash_table_size);
        printf("    j    | new size | memory | free m | colnum\n");
        for (j = 1.1; j < 3; j += 0.005)
        {
            colnum = 0u;
            free_entries = 0u;
            test_new_size = (double)hash_table->hash_table_size * j;
            test_new_table = (hash_table_node_t **)calloc(test_new_size,
                sizeof(hash_table_node_t *));

            for (i=0; i < hash_table->hash_table_size; i++)
            {
                for (node = hash_table->heads[i]; node; node = next)
                {
                    test_new_key = crc32(node->word) % test_new_size;
                    next = node->next;
                    if (test_new_table[test_new_key])
                    {
                        colnum++;
                    }
                    test_new_table[test_new_key] = node;
                }
            }
            for (k=0; k < test_new_size; k++) {
                if (!test_new_table[k]) {
                    free_entries++;
                }
            }
            printf("%3.3f | %8u | %6lu | %6lu | %6u\n", j, test_new_size,
                test_new_size * sizeof(hash_table_node_t *), free_entries * sizeof(
                    hash_table_node_t *), colnum);
        }
    }
}
```



7.3 Script MATLAB que gera os gráficos para análise da hash_table_grow

```
% Get data from file
table = load("first.txt");
j = table(:,1);
new_size = table(:,2);
memory = table(:,3);
free_memory = table(:,4);
collisions = table(:,5);

% Sort free_memory & collisions arrays, based on free_memory
[free_memory_sorted,sortIdx] = sort(free_memory,'ascend');
collisions_sorted = collisions(sortIdx);

% Get ratios
ratio_col_mem = collisions./memory;
ratio_col_free = collisions./free_memory;

% Plots
figure(1)
plot(memory,collisions)
xlabel('Total memory (bytes)')
ylabel('Collisions')
grid on

figure(2)
plot(free_memory_sorted,collisions_sorted)
xlabel('Free memory (bytes)')
ylabel('Collisions')
grid on
xlim([5000 20000])

figure(3)
plot(j,ratio_col_mem)
xlabel('Increment, j')
ylabel('Collisions/Total memory ratio')
grid on

figure(4)
plot(j,ratio_col_free)
xlabel('Increment, j')
ylabel('Collisions/Free memory ratio')
grid on

figure(5)
plot(memory,free_memory)
xlabel('Total memory (bytes)')
ylabel('Free memory (bytes)')
grid on
```

