Licenciatura em Engenharia Informática

Algoritmos e Estruturas de Dados

Word Ladder

João Catarino
NMec: 93096

Rúben Garrido
NMec: 107927

Nuno Vieira
NMec: 107283

8 de janeiro de 2023

Índice

1 Introdução

Este trabalho consiste na criação de uma *word ladder* a partir de um ficheiro de texto, que contém uma lista de palavras. A *word ladder* é uma sequência de palavras, em que cada palavra difere da anterior por apenas uma letra. Por exemplo, *tudo, todo, nodo, nado, nada* é uma *word ladder* entre as palavras *tudo* e *nada*.

2 Estrutura de dados

Para armazenar as palavras obtidas a partir do ficheiro de texto, bem como as suas interligações, foi utilizada uma *hash table*, pelo facto de esta possuir uma complexidade computacional $O(1)$ para a inserção e remoção de elementos. Esta complexidade computacional é importante, uma vez que, caso esta fosse $O(n)$ e existisse um número avultado de palavras, a *word ladder* poderia demorar algum tempo a ser gerada.

A *hash table* é composta por um array de *buckets*, que contêm as palavras e as suas interligações. Cada *bucket* é constituido por uma *linked list*. A *linked list* contém *nodes*, onde, no caso de haver colisões, existirá mais do que um. Cada *node* contém uma palavra e um apontador para o próximo *node* da *linked list*. A figura 1 mostra a estrutura da *hash table*.

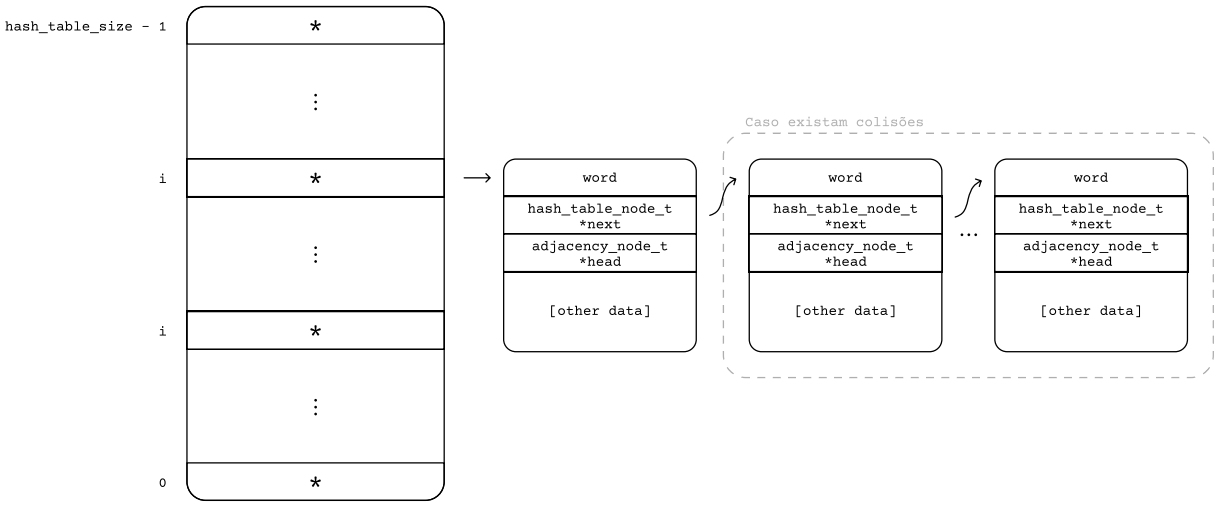


Figura 1: Estrutura da *hash table*

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 Redudancy Check*, 32 bits).

3 Verificação de *memory leaks*

Para verificar se existem *memory leaks* no programa criado, foi utilizado o *Valgrind*. Este programa é um conjunto de ferramentas para a deteção de erros de *memory* 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
             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.

4 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 perca 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.

4.1 Explicação do código

Foi desenvolvida uma nova função `hash_table_grow`, num programa à parte, onde, 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 definir 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++;
      }
    }
```

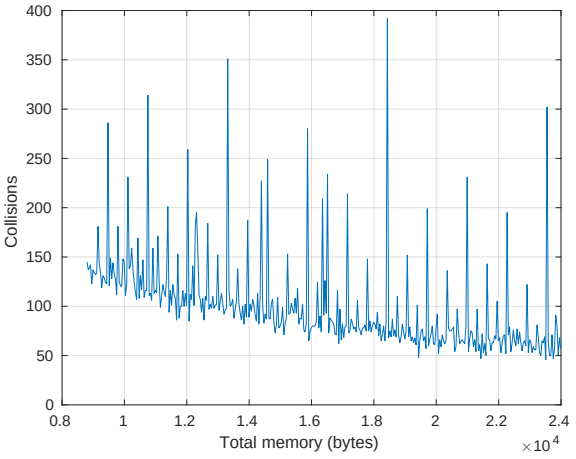
Por fim, é impressa uma linha com os dados obtidos, nomeadamente o fator de incremento `j`, o novo tamanho da *hash table* `test_new_size`, a memória total ocupada `test_new_size * sizeof(hash_table_node_t *)`, a memória ocupada por entradas livres `free_entries * sizeof(hash_table_node_t *)` e o número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

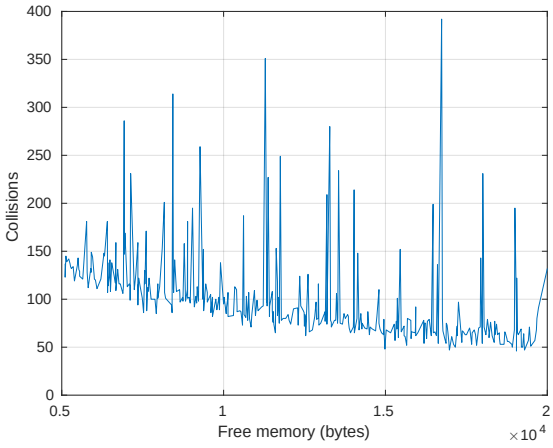
4.2 Gráficos obtidos

Através do MATLAB, foi possível obter um conjunto de gráficos, que relacionam colisões com memória livre e memória total. O script, disponível na secção 5.2, 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.

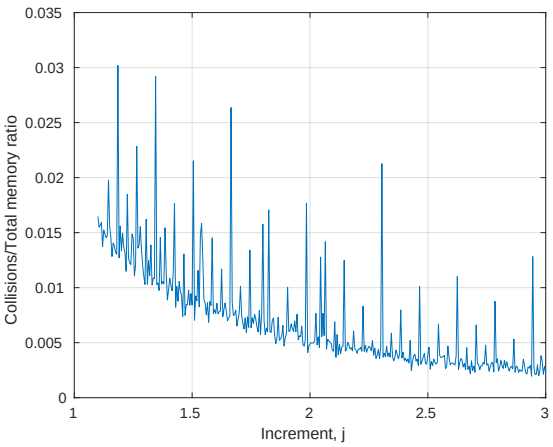


(a) Número de colisões em função da memória total.

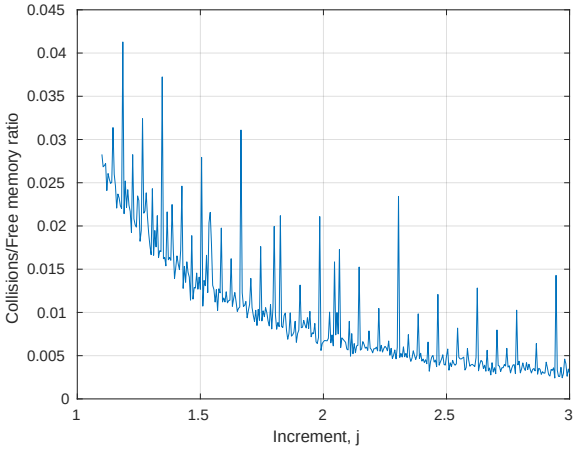


(b) Número de colisões em função da memória livre.

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.

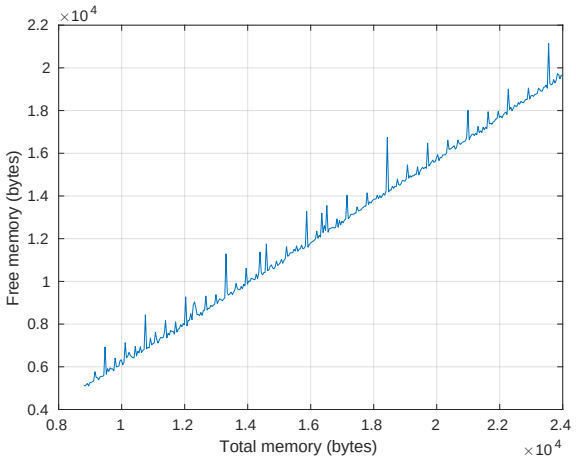


Figura 4: Memória livre em função da memória total.

4.3 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, o que leva a colisões.

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 similariedade entre rácios explica-se através do 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 diz respeito 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.

5 Código

5.1 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);
    }
  }
}
```

5.2 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
```