Algoritmo de Armazenamento de Palavras

Luis Alexandre Ferreira Bueno e Vitor Bruno de Oliveira Barth

February 1, 2017

Abstract

Este trabalho tem por objetivo apresentar um algoritmo, que, por meio dos conceitos de dicionário endereçado por meio de hash, armazene palavras da língua inglesa e portuguesa, e graficamente apresente as colisões geradas pela função de espalhamento.

1 Introdução

Foi-nos apresentado a seguinte atividade: "Elaborar um algoritmo que armazene palavras da língua inglesa e portuguesa brasileira na estrutura de dicionário de dados e gere uma imagem para ilustrar as colisões de cada conjunto de palavras contidas na estrutura".

O conceito de dicionário de dados consiste em uma lista organizada na qual os elementos da mesma são tradados como grupos ou conjuntos de dados. Quando se deseja adicionar um dado é determinado o conjunto na qual ele será armazenado. Para desenvolvermos esse conceito criamos uma biblioteca cujas funções alocam e armazenam os dados de forma dinâmica na memória.

Algoritmo principal ficou responsável por ler o arquivo que contém uma lista de palavras, executar as funções da biblioteca e gerar a imagem que ilustra as colisões. Para trabalharmos com a imagem, escolhemos utilizar o formato Netpbm (extensões. pbm ou .ppm), o qual usa o padrão ASCII para salvar as informações de cor, e é fácil de ser incorporado a um código C.

```
P1
# This is an example bitmap of the letter "J"
6 10
0 0 0 0 1 0
0 0 0 0 1 0
0 0 0 0 1 0
0 0 0 0 1 0
0 0 0 0 1 0
0 0 0 0 1 0
1 0 0 0 1 0
0 1 1 1 0 0
0 0 0 0 0 0
0 0 0 0 0 0
```

2 Tabela Espalhamento

2.1 Estrutura

Para que pudéssemos trabalhar com os conceitos de dicionário de dados em nossa biblioteca, organizamos os dados em três estruturas: Node (nó), List (lista de nós), ListMatrix (matriz de listas de nós).

A estrutura Node é responsável por armazenar cada palavra e indicar qual é a posição em memória da palavra seguinte, posicionadas de acordo com a ordem de entrada, ou seja, sempre adicionando ao final.

A List é onde se encontrarão as palavras que possuem o mesmo hash. Ela armazena a posição dos primeiro e último nós, e a quantidade de palavras a ela ligadas.

Por fim, ListMatrix é a estrutura que organiza todas as listas e sobre o formato dela é que geraremos a imagem. Eis a declaração das estruturas:

```
typedef struct Node {
  char key[KEY_SIZE]; // Armazena a string
  struct Node *next;  // Endereça o próximo nó
} Node;

typedef struct List {
  Node *firstNode;
  Node *lastNode;
  int  size;
} List;

typedef struct ListMatrix {
  List *matrix[MATRIX_HW][MATRIX_HW]; // Matriz de listas
  int size;                           // Quantidade de nós ligados a matriz
} ListMatrix;
```

2.2 Função Hash (função de espalhamento)

Quando desejamos adicionar um dado em nossa em estrutura, primeiramente temos que determinar em qual conjunto de dados ele deverá ser armazenado. Para isso ele passa por uma função, chamada Função Hash ou Função de Espalhamento, que devolve a posição na matriz onde a palavra deverá ser salva.

Existem diversos tipos de algoritmos geradores de hash, e o objetivo deste trabalho é mostrar a qualidade destes algoritmos de acordo com a maneira que as colisões estão espalhadas pela matriz. Quanto mais homogênea for a quantidade de colisões, melhor o algoritmo gerador de hash é.

3 Geração de Imagem

Se tratando em gerar imagens em .ppm usamos a função genPPM, que possui cinco variáveis principais: - resolutionWidth e resolutionHeight que correspondem, respectivamente, largura e altura (em pixels) da imagem a ser gerada. - blockSizeWidth e blockSizeHeight que armazenam, respectivamente, a largura e altura (em pixels) de cada bloco da imagem a ser gerada. - MATRIX-HW que corresponde à quantidade de blocos da matrixList em largura e altura.

A parte mais importante desta função é composta de quatro loops for encadeados. Os dois mais externos servem para iterar as linhas da imagem, e os dois mais internos servem iterar as colunas da imagem, e deste modo gerar cada bloco da linha.

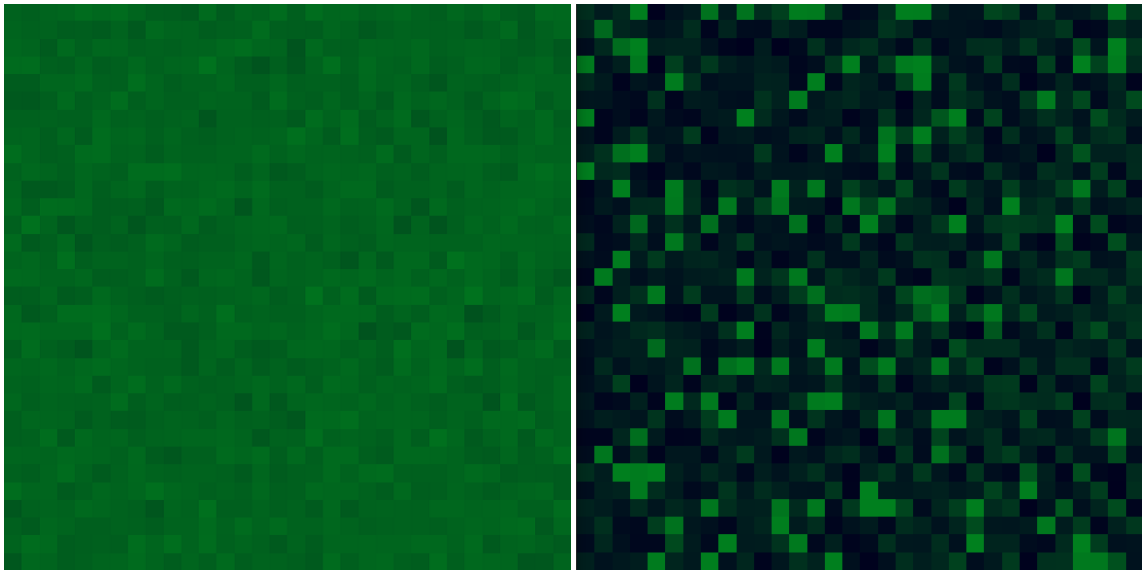
Para representar as colisões foi escolhida a cor verde como cor base para os blocos. Nosso sistema de cores RGB foi predefinido em 0 G 30, sendo G variável, podendo assumir valores entre 0 a 128 gerando assim tonalidades de verdes, sendo tons mais claros representam os conjuntos de palavras que tiveram menos colisões e tons mais escuros representam os conjuntos de palavras que tiveram mais colisões. A formula que define a tonalidade é esta.

```
int block = MAXVALUE - (listMatrix->matrix[k][i]->size%MAXVALUE);
fprintf(image, "%i %i %i ", 0, block, 30);
```

A função hash que utilizamos inicialmente foi:

```
int genHash(char key[KEY_SIZE]) {
        int acumulator = 0;
        for (int i = 0; i < KEY_SIZE; i++)
                acumulator += key[i] * (i + 1) * 3;
        return abs(acumulator % (MATRIX_SIZE));
```

Abaixo estão as imagens geradas pelo nosso programa utilizando a função hash acima. A esquerda estão representadas as colisões quando adicionamos uma lista de 29.858 palavras em português, e a direita quando adicionamos 10x vezes mais: 235.886 palavras em inglês.

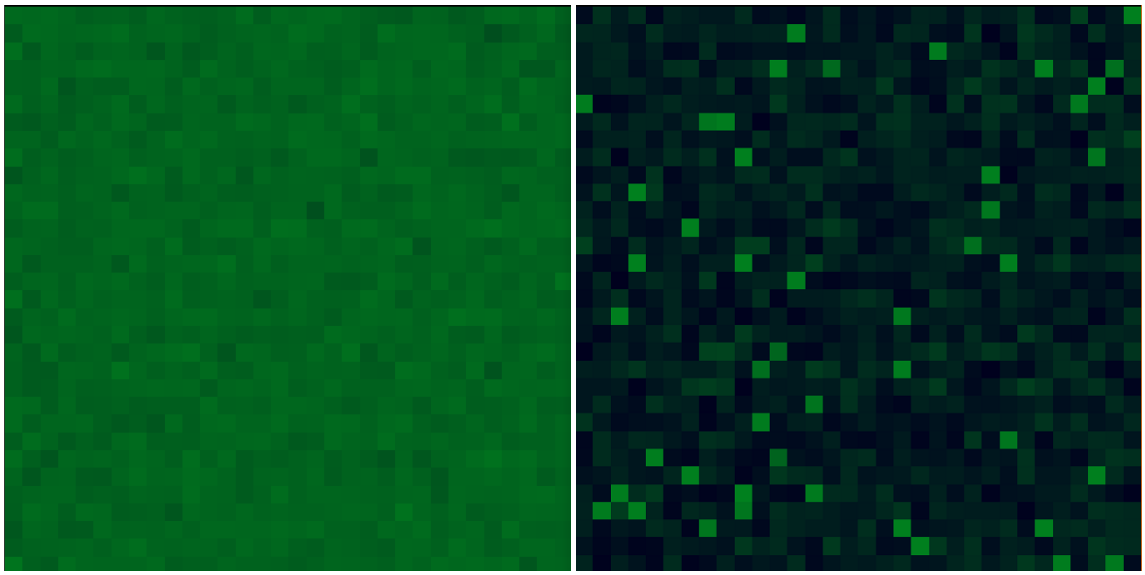


Como se pode perceber, quanto mais palavras são adicionaas, maior o número de colisões, e logo mais escura a imagem fica. Contudo, existem pontos bastante claros na imagem da direita, indicando que a função de espalhamento não é muito boa.

Mudando a função hash para

```
int genHash(char key[KEY_SIZE]) {
        int acumulator = 0;
        for (int i = 0; i < KEY_SIZE; i++)
                acumulator += (int)        (sin(key[i])*10000);
        return abs(acumulator % (MATRIX_SIZE));
}
```

Obtemos as seguintes imagens:



Como se pode perceber ao comparar com as imagens anteriores, a lista de palavras em inglês produziu uma imagem um pouco mais homogêna, com menos pontos claros. Contudo, usar a lista de palavras em português não gerou diferença perceptivel no número de colisões.