Artigo: An optimal algorithm for generating minimal perfect hash functions

Autores: Zbigniew J. & George Havas and Bohdan S. Majewski

Os autores iniciam considerando que *W* seja um conjunto de *m* palavras formadas por um alfabeto ∑, existe então uma função hash *h: W→I* que mapeia o conjunto de palavras de *W* em um intervalo de inteiros *I*, onde I é de tamanho *k* maior ou igual a *m*. Eles explicam que palavras que são computadas para o mesmo endereço *I* são sinônimas, e que geram uma situação chamada colisão.

A colisão implica que duas palavras tais como w_1 e w_2 são adereçadas para o mesmo local. Uma função *h* que computa todas as palavras de *W* para endereços únicos é conhecida como função hash perfeita mínima. Os autores falam que existem vários algoritmos de diferentes complexidades que sugerem funções hash perfeitas mínimas, onde um cientista alegou ter um algoritmo cuja complexidade era O(m^4).

Nesse sentido, os autores apresentam um novo algoritmo baseado em random graphs para encontrar uma função hash mínima perfeita da seguinte forma:

$$h(w) = (g(f_1(w)) + g(f_2(w))) \bmod m$$

Onde f_1 e f_2 são funções que mapeiam strings em inteiros, e *g* é uma função que mapeia inteiros em [0, *m*-1].

O algoritmo proposto pelos autores é composto por duas etapas: mapeamento e atribuição. Na etapa de mapeamento é construído um grafo G = (V,E) onde V = {0,...,n-1}, onde n será determinado depois e E = {(f_1(*w*),f_2(*w*)): w pertence a W}. As funções auxiliares f_1 *e* f_2 são modeladas para ser duas funções aleatórios de mapeamento W em [0,n-1] independentes. O pseudo-código do algoritmo é:

```
repeat
    initialize E := ∅;
    randomly generate tables T1 and T2;
    for w ∈ W loop
        f1(w) := (Σ_{j=1}^{|w|} T1(j, w[j])) mod n;
        f2(w) := (Σ_{j=1}^{|w|} T2(j, w[j])) mod n;
        add the edge (f1(w), f2(w)) to graph G;
    end loop;
until G is acyclic;
```

Fig. 1. The mapping step.

```
procedure traverse(u : vertex);
begin
    visited[u] := TRUE;
    for w ∈ neighbours(u) loop
        if not visited[w] then
            g(w) := (h(e = (u, w)) − g(u)) mod m;
            traverse(w);
        end if;
    end loop;
end traverse;

begin
    visited[v ∈ V] := FALSE;
    for v ∈ V loop
        if not visited[v] then
            g(v) := 0;
            traverse(v);
        end if;
    end loop;
end;
```

Fig. 2. The assignment step.

```
function h(w : string) : integer;
begin
    u := (∑_{j=1}^{|w|} T1(j, w[j])) mod n;
    v := (∑_{j=1}^{|w|} T2(j, w[j])) mod n;
    return (g(u) + g(v)) mod m;
end;
```

Fig. 3. Evaluating the hash function.

Nas análises e testes realizados pelos autores, eles mostram que o tempo de complexidade do algoritmo é linear a quantidade (m) de palavras em W, ou seja O(m). A complexidade de espaço é cm log m + O(1) log n bits, ou cm + O(1) palavras, contanto que n caiba em uma palavra. Abaixo tabela elaborada pelos autores com os resultados obtidos a partir de m entradas.

Table 1
Experimental results

$m = n/3$	iterations	mapping	assignment	total
512	1.704	0.037	0.010	0.047
1024	1.684	0.052	0.019	0.072
2048	1.776	0.095	0.037	0.132
4096	1.676	0.169	0.067	0.236
8192	1.668	0.320	0.142	0.463
16384	1.680	0.628	0.293	0.921
24692	1.688	0.950	0.444	1.394
32768	1.636	1.353	0.597	1.949
65536	1.696	2.718	1.198	3.916
131072	1.676	5.448	2.416	7.864
262144	1.768	11.273	4.813	16.087
524288	1.736	22.493	10.414	32.907

Acesso:
https://reader.elsevier.com/reader/sd/pii/002001909290220P?token=99C70042B8A630FC14654DD48A1F952D42BD27027ED7F4DB227024BDF2DFBB166F7CDFD875EEFB6C5E94B4519F06876B