Statistical Behavior of Fast Hashing of Variable-Length Text Strings

Jacques Savoy

Université de Montréal

Département d'informatique et de recherche opérationnelle

P. O. Box 6128, station A

Montréal (Qc) H3C 3J7 (Canada)

Introduction

In information retrieval, we often have to store and search for a particular record into a large amount of information. For example, during a document indexing process or when a program is trying to spell a text, a dictionary has to be used in an efficient way. A solution to that problem resides in using a hash table. However, if we known many algorithms for manipulating or accessing hash tables [Knuth 73], [Standish 80], [Wiederhold 87], the main problem is to define a "good" hash function for a variable-length string. In order to answer that question our main goals are to present some concrete algorithms and to study their statistical behavior.

Keywords

key-to-address transformation, hashing, hash function, hash coding, direct access method, scatter storage, dictionary lookup, information retrieval.

The problem

Without prior knowledge of the keys distribution, we need to find a "good" hashing function to map variable-length string into integer. We may wish to delete some previously-stored records or to add new ones from time to time. In that case, one cannot assert that the keys set is static and, therefore, one cannot use a perfect hashing functions [Sprugnoli 77], [Cichelli 80]. Moreover, the hash function must be able to return distinct values for similar but non-identical keys, such as "money" and "honey" or as the anagrams ("poem" and "mope"). Finally, the key transformation must only require a few operations.

In the previous studies, [Lum 72] and [Lum 73] propose various methods to solve that problem but, the authors conclusions can be sum up by the following remark of D. Knuth :

> "Many more methods for hashing have been suggested, but none of these has proved to be superior to the simple division and multiplication methods ..." [Knuth 73, p. 512]

If we agree to use a hash function corresponding to the division or the multiplication schemes, we must still define the implementations details and study its statistical behavior.

The Pearson's solution

In a recent article [Pearson 90], the author suggests a new algorithm to map variable-length string onto integer. A C string consists of a sequence of characters c_0, c_1, ..., c_n. Each character is represented by one byte (in the range of 0 to 255). The algorithm needs an auxiliary table named T whose index goes from 0 to 255. This table memorizes a permutation of the values (0, 1, ..., 255) (for example, by a shuffling algorithm [Knuth 81, Section 3.4.2]). The proposed solution is the following :

```
integer hash(array c);

h = 0;
n = ArrayLength(c) - 1;
for i = 0 to n do
    h = T[bitXor(h, c[i])];  /* XOR bit by bit */
return h;
```

This algorithm returns a small integer between 0 and 255, a table T element. How does it work ? As an example, one can limit the alphabet to the four symbols {'a', 'b', 'c', 'd'}. The corresponding codes are {'a ' = 0, 'b' = 1, 'c' = 2, 'd' = 3}. Two bits are needed to code each character of the alphabet. The table T is set to the array [2 3 1 0], a random permutation of the value 0 to 3.

hash ('bc') = T[T[0 bitXor 1] bitXor 2] = T[3 bitXor 2] = 3
hash ('cb') = T[T[0 bitXor 2] bitXor 1] = T[1 bitXor 1] = 2
hash ('ab') = T[T[0 bitXor 0] bitXor 1] = T[2 bitXor 1] = 0
hash ('ba') = T[T[0 bitXor 1] bitXor 0] = T[3 bitXor 0] = 0

With that example, we can easily see that the proposed solution separates the anagrams (the string 'bc' and 'cb'). However, this fact may not be applicable to all anagrams (e. g. the words 'ab' and 'ba' generates the same integer). What that solution garantees is the fact that if two strings differ in only one bit or by one character, they will have different hash value.

Varying the table size

If one want an extended range for the hash value, the author ([Pearson 90]) proposes the following algorithm :

```
integer hash16(array c);

h1 = hash(c);
c[1] = (c[1] + 1) \\ 256; /* c[1] + 1 modulo 256 */
h2 = hash(c);
h1 = bitShift(h1, 8);    /* shift 8 bits to the left */
return (h1 + h2);        /* concatenate h1 and h2 */
```

This version leads to outcomes from 0 to 65,535. In this scheme, the size variation is very limited !

Statistical performance of the solution

We had to study the hash function statistical behavior and, specially the question "Are the 256 outcomes equally probable ?". The author does one traditional chi-square goodness-of-fit test with an English dictionary of 26,662 words. His answer is yes ($\chi^2 = 255.64$, d.f. 255, $p = 0.477$). I have used the same T table with a French dictionary of 52,748 entries (word in lowercase but with some accents which means a dozen codes between 128 and 255). The words mean length is 8.81334, and the variance is 7.85056 (standard deviation = 2.801885). The chi-square test result I obtained is alarming ($\chi^2 = 309.383$, d.f. 255, $p = 0.0103$).

If the dictionary size is equal to n and the hash table length is m, each word has a probability of 1/m of being assigned to any one cell in the hash table. The probability that one cell receives k words is given by a binomial distribution :

$$\text{Prob [one cell receives k words]} = \frac{n!}{k! \cdot (n-k)!} \cdot \left(\frac{1}{m}\right)^k \cdot \left(1 - \frac{1}{m}\right)^{n-k} \qquad (1)$$

In our example, since $n >> 1$, $m >> 1$, and $k << n$, we can approximate the binomial distribution by a Poisson distribution in which the density factor equals n/m.

$$\text{Prob [one cell receives k words]} = e^{-n/m} \cdot \frac{(n/m)^k}{k!} \qquad (2)$$

$$\text{Prob [one cell receives no word]} = e^{-n/m}$$

$$\text{Prob [one cell receives one or more words]} = 1 - e^{-n/m}$$

If we use a table size of $2^{16} = 65536$ and a the dictionary size n = 52,748, the expected number of collision is :

mean = n - m · Prob [one cell receives one or more words] =

$$n - m \cdot (1 - e^{-n/m}) = 16{,}516. \qquad (3)$$

with a standard deviation = $\frac{n}{m} \cdot \sqrt{m} = 206.046$

In my sample, I count 23,920 collisions (expected mean is 36 times more than the standard deviation). The hash function does not produce an equally probable outcome for all values.

Collision-resolution policy

A common problem amongst all hashing methods is the definition of the collision-resolution policy. In other words, what we do when :

h(C) = h(C'), for a string C ≠ C' (i.e. hash('ab') = hash('ba')).

Of course, we can adopt the well known chaining procedure (or linked list), but this solution needs extra memory whose amount is expressed in equation 3. With the open addressing policy, the computer scientist must find a method to generate a probe sequence, a permutation of the table addresses, based on p(C).

We do not use the linear probing :

$$p_j = h(C) - j \cdot p(C) \text{ modulo } m, \qquad (4)$$

where m is table size and p(C) = 1 for all C string.

This solution leads to a phenomenon called primary clustering [Knuth 73, Section 6.4]. We can program a pseudo random probing :

$$p_j = \begin{cases} 0 & \text{if } j = 0 \\ h(C) + r_j \text{ modulo } m & \text{if } j > 0 \end{cases} \qquad (5)$$

where m is the table size and r is a random number sequence in which the seed is defined by the hash(C) value.

In that case, we encounter a similar problem named secondary clustering [Amble 74] that occurs when p(C) is defined in the form p(C) = f(h(C)). To solve the collision problem, [Pearson 90] suggests :

> "The function h [hash in this paper] is well suited to this sort of application. By repeatedly incrementing the first character of the input string, modulo 256, one causes the hash index returned by h to pass through all 256 possible index values in a vary irregular manner." [Pearson 90, p. 679]

Since the key itself changes, we do not encounter the primary or secondary clustering problem [Knuth 73, Section 6.4].

What if the same collision-resolution policy, for a hash function, would be used with a table of size 65,536 ? In that case, we increment the first and the second letter of the word to generate all the combinations of the two first letters of the string as described in equation 6.

$$p_j = h(c_0 + j \setminus\setminus 256, c_1 + j // 256, c_2, ..., c_n)). \qquad (6)$$

With all those combinations, can we produce a permutation of the space index ? The answer is no as shown in the following example.

We want to map a sequence of words composed by a combination of our four letters into a 4 bits integer. We apply a simpler version of the algorithm hash16. The generation of all two letters combinations of our alphabet gives the following results ($\oplus$ means the concatenation operator) :

hash16 ('aa')	= hash('aa') $\oplus$ hash('ba')	= 1 $\oplus$ 0	= 4
hash16 ('ab')	= hash('ab') $\oplus$ hash('bb')	= 0 $\oplus$ 1	= 1
hash16 ('ac')	= hash('ac') $\oplus$ hash('bc')	= 2 $\oplus$ 3	= B
hash16 ('ad')	= hash('ad') $\oplus$ hash('bd')	= 3 $\oplus$ 2	= E
hash16 ('ba')	= hash('ba') $\oplus$ hash('ca')	= 0 $\oplus$ 3	= 3
hash16 ('bb')	= hash('bb') $\oplus$ hash('cb')	= 1 $\oplus$ 2	= 6
hash16 ('bc')	= hash('bc') $\oplus$ hash('cc')	= 3 $\oplus$ 0	= C
hash16 ('bd')	= hash('bd') $\oplus$ hash('cd')	= 2 $\oplus$ 1	= 9
hash16 ('ca')	= hash('ca') $\oplus$ hash('da')	= 3 $\oplus$ 2	= E
hash16 ('cb')	= hash('cb') $\oplus$ hash('db')	= 2 $\oplus$ 3	= B
hash16 ('cc')	= hash('cc') $\oplus$ hash('dc')	= 0 $\oplus$ 1	= 1
hash16 ('cd')	= hash('cd') $\oplus$ hash('dd')	= 1 $\oplus$ 0	= 4
hash16 ('da')	= hash('da') $\oplus$ hash('aa')	= 2 $\oplus$ 1	= 9
hash16 ('db')	= hash('db') $\oplus$ hash('ab')	= 3 $\oplus$ 0	= C
hash16 ('dc')	= hash('dc') $\oplus$ hash('ac')	= 1 $\oplus$ 2	= 6
hash16 ('dd')	= hash('dd') $\oplus$ hash('ad')	= 0 $\oplus$ 3	= 3

If we consider the probe sequence implied by the string 'aa', we generate the strings { 'ba', 'ca', 'da', 'ab', 'bb', 'cb', 'db', 'ac', 'bc', 'cc', 'dc', 'ad', 'bd', 'cd', 'dd' } and the following addresses { 3, E, 9, 1, 6, B, C, B, C, 1, 6, E, 9, 4, 3 }.

Thus, we can not obtain a permutation of the space index. The proposed collision-resolution policy does not work. Since the table size is a multiple of two, we can calculate $h_2(C)$ as a odd value in such a way that $h_2(C)$ is relatively prime to the table size [Bell 70]. In that case, the probe sequence defined by :

$$p_j = h(C) - j \cdot h_2(C) \qquad (7)$$

does not suffer from secondary clustering. The function h_2 is the following :

```
integer hash2(array c);

c[1] = (c[1] + 1) \\ 256;     /* c[1] + 1 modulo 256 */
h = hash(c);
return (bitOr(h, 1));    /* to be sure that h is odd */
```

Alternatives and their performances

If the Pearson's solution is not the best choice, what else can we do ? Various key-to-address functions are use everyday. Some of them could have acceptable performances. We select some hash function with the following criteria :

- no restriction on the length of the text string,
- no requirement for the string length to be known in advance,
- needs very little arithmetic,
- similar strings are separated (specially the anagrams)
- with a good statistical behavior.

Some of the alternatives are very simple; they use the first character, the second and the string size (hashSolution1) or the first character, the last and the string size (hashSolution2). This proposition is a simpler version of the algorithm proposed by [Cichelli 80]. The Smalltalk system [Goldberg 89] uses a variation of this schema (hashSolution3).

Some authors propose the use of all the information content in the string in a hash function [Wiederhold 87]. Their solutions are variations on the simple additive schema (hashSolution4). We may use the xor operator instead of adding (hashSolution5). In order to separate the anagrams, we may shift before the addition (hashSolution6 and hashSolution7).

```
integer hashSolution1(array c);

n = ArrayLength(c);
if n == 0 then return 0;
if n == 1 then h = F[c[0]] + 1;
   else h = F[c[0]] + F[c[1]] + n;
return h \\ 256;          /* result modulo 256 */
```

```
integer hashSolution2(array c);

n = ArrayLength(c);
if n == 0 then return 0;
h = F[c[0]] + F[c[n-1]] + n;
return h \\ 256;


integer hashSolution3(array c);

n = ArrayLength(c);
if n == 0 then return 85;
if n == 1 then return ((F[c[0]] * 106) \\ 256);
if n == 2 then n = 3;
h = F[c[0]] * 48 + F[c[n-2]] + 1;
return h \\ 256;


integer hashSolution4(array c);

h = 0;
n = ArrayLength(c) - 1;
for i = 0 to n do
    h = h + F[c[i]];
return h \\ 256;


integer hashSolution5(array c);

h = 0;
n = ArrayLength(c) - 1;
for i = 0 to n do
    h = bitXor(h, F[c[i]]);
return h;


integer hashSolution6(array c);

h = 0;
n = ArrayLength(c) - 1;
for i = 0 to n do
    h = bitShift(h, 1) + F[c[i]];
return h \\ 256;
```

```
integer hashSolution7(array c);

h = 0;
n = ArrayLength(c) - 1;
for i = 0 to n do
    h = bitXor(bitShift(h, 1), F[c[i]]);
return h \\ 256;


integer hashSolution8(array c);

h = 0;
n = ArrayLength(c) - 1;
for i = 0 to n do
    h = (h * 137 + F[c[i]]) \\ 256;
return h;
```

I have programmed these hash functions. In the previous functions, the array F[] returns the character's ascii code (code between 0 and 255) or a random permutation of these numbers. The outcome of these functions are a value between 0 and 255. The following table resumes the statistical behavior of all these hash functions. We did this test with a 52,748 words french dictionary.

Hash function	the array F[] returns the ascii value	F[] is a random permutation of (0, ..., 255)
hashSolution1	$\chi^2 = 246110$	$\chi^2 = 31179.8$
	d.f. 255, $p \simeq 0.00$.	d.f. 255, $p \simeq 0.00$.
hashSolution2	$\chi^2 = 276623$	$\chi^2 = 24576.1$
	d.f. 255, $p \simeq 0.00$.	d.f. 255, $p \simeq 0.00$.
hashSolution3	$\chi^2 = 19449.6$	$\chi^2 = 19228.9$
	d.f. 255, $p \simeq 0.00$.	d.f. 255, $p \simeq 0.00$.
hashSolution4	$\chi^2 = 323.157$	$\chi^2 = 273.391$
	d.f. 255, $p = 0.002$.	d.f. 255, $p = 0.2055$.
hashSolution5	$\chi^2 = 61659.9$	$\chi^2 = 357.782$
	d.f. 255, $p \simeq 0.00$.	d.f. 255, $p \simeq 0.00$.
hashSolution6	$\chi^2 = 19249.8$	$\chi^2 = 14653.7$
	d.f. 255, $p \simeq 0.00$.	d.f. 255, $p \simeq 0.00$.
hashSolution7	$\chi^2 = 23051.7$	$\chi^2 = 17367.1$
	d.f. 255, $p \simeq 0.00$.	d.f. 255, $p \simeq 0.00$.
hashSolution8	$\chi^2 = 283.641$	$\chi^2 = 259.181$
	d.f. 255, $p = 0.1044$.	d.f. 255, $p = 0.4182$

Table 1 : Chi-square test

Regarding the table 1, we can reject a hash function based only on a subset of the text string (hashSolution1 to hashSolution3) . The solutions (hashSolution4 to hashSolution7) derived from the simple additive schema do not present a good separation of the text strings into the value range. The solution hashSolution8 is the only one who provides an acceptable statistical behavior. Finally, another point to consider is the use of a table lookup instead of the ascii code. For all solutions, the use of a random permutation of the ascii code gives a better chi-square result.

Hash function	Can you separate anagrams ?
hashSolution1	No (yes in limited case)
hashSolution2	No (yes in limited case)
hashSolution3	No (yes in limited case)
hashSolution4	No
hashSolution5	No
hashSolution6	Yes
hashSolution7	Yes
hashSolution8	Yes

Table 2 : Anagrams separation

Beside the statistical considerations, the table 2 shows us that the hash function had to separate similar strings. This is not a garanteed fact for all proposed solutions. For example, ours solution, hashSolution8, separates well similar but non identical strings.

Conclusion

When we define a hash function for text strings, we often adopt a simple solution, for example by adding characters ascii codes with or without a shift between each character. Recording to the references, we only find a recommendation for the division or multiplication methods. In this article, we have proved that the recent Pearson's solution [Pearson 90] can not be as random as the author suggests. On the other hand, we have studied some alternatives and found another hash function which has a good statistical behavior and which separates well similar text strings.

References

[Amble 74] O. Amble, D. E. Knuth : Ordered Hash Tables. Computer Journal, 17:2, 1974, pp. 135-142.

[Bell 70] J. R. Bell, C. H. Kaman : The Linear Quotient Hash Code. Communications of the ACM, 13:11, 1970, pp. 675-677.

[Cichelli 80] R. J. Cichelli : Minimal Perfect Hash Functions Made Simple. Communications of the ACM, 23:1, 1980, pp. 17-19.

[Goldberg 89] A. Goldberg, D. Robson : Smalltalk-80 : The Language. Addison-Wesley, Reading (Massachusetts), 1989.

[Knott 75] G. D. Knott : Hash Functions. Computer Journal, 18:3, 1975, pp. 265-278.

[Knuth 73] D. E. Knuth : The Art of Computer Programming, Volume 3 : Sorting and Searching. Addison-Wesley, Reading (Massachusetts), 1973.

[Knuth 81] D. E. Knuth : The Art of Computer Programming, Volume 2 : Seminumerical Algorithms. Addison-Wesley, Reading (Massachusetts), 2nd edition, 1981.

[Lum 72] V. Y. Lum, P. S. T. Yuen, M. Dodd : Key-to-Address Transform Techniques : A Fundamental Performance Study on Large Existing Formatted Files. Communications of the ACM, 14:4, 1972, pp. 228-239.

[Lum 73] V. Y. Lum, P. S. T. Yuen : Additional Results on Key-to-Address Transform Techniques : A Fundamental Performance Study on Large Existing Formatted Files. Communications of the ACM, 15:11, 1973, pp. 996-997.

[Pearson 90] P. K. Pearson : Fast Hashing of Variable-Length Text Strings. Communications of the ACM, 33:6, 1990, pp. 677-680.

[Sprugnoli 77] R. Sprugnoli : Perfect Hashing Functions : A Single Probe Retrieving Method for Static Sets. Communications of the ACM, 20:11, 1977, pp. 841-850.

[Standish 80] T. A. Standish : Data Structure Techniques. Addison-Wesley, Reading (Massachusetts), 1980.

[Wiederhold 87] G. Wiederhold : File Organization for Database Design. McGraw-Hill, New-York, 1987.