Algoritmy 2009 proceedings, Slovak University of Technology, Bratislava, pp.382-391, ISBN 978-80-227-3032-7, 2009

EFFICIENT HASH FUNCTION FOR DUPLICATE ELIMINATION IN DICTIONARIES

VACLAV SKALA[†] , JAN HRADEK

Abstract. Fast elimination of duplicate data is needed in many areas, especially in the textual data context. A solution to this problem was recently found for geometrical data using a hash function to speed up the process. The usage of the hash function is extremely efficient when incremental elimination is required especially for processing large data sets. In this paper a new construction of the hash function is presented, giving short clusters with few collisions only. The proposed hash function is not a perfect hash function, nevertheless it gives similar properties to it. The hash function used takes advantage of the relatively large amount of available memory on modern computers, and works well with large data sets.

Experiments have proved that different approaches should be used for different types of languages, because the structures of Slavonic and Anglo-Saxon languages are different. Therefore, tests were made with a Czech dictionary having 2.5 million words and an English dictionary having 130 thousands words. Algorithm was also tested for a few other languages. Experimental results are presented in this paper as well.

Key words. Hash function, hash table, duplicate elimination, data structure, dictionary

AMS subject classification. 65Y20, 68P05, 68P10, 68P20

1. Notation

M - the number of items in the original data set with duplicates,
N - the number of items in the final data set after the duplicate elimination,
I - the number of items in a cluster (also stated as the length of a cluster),
I_a, I_m - the average cluster length, resp the maximal cluster length,
C_I - the number of clusters of length I,
Q - the value of the absolute criterion,
Q' - the value of the relative criterion,
$h(x)$, HS - the hash function value, resp. the hash table length,
c - the multiplication coefficient of the proposed hash function,
q - the hash function coefficient of the proposed hash function,
x - the processed string from dictionary,
x_i - an alphabetical value (A=0, B=1, …) of the character in the string x at the position i,
L_x L_{max} - the length of the string x, resp. maximal length,
n - the number of characters in the given alphabet,
f - the load factor of the hash function,
α,β,γ - the coefficient of the recent hash function.

2. Introduction. Many problems require fast elimination of duplicate data. Resolving this problem can be very difficult, especially as the size of the data set increases. The solution also depends on how the data set is to be processed, e.g. if it is to be processed incrementally or in a batch. Several approaches can be taken to solve this problem. Some strategies that can be used are presented in TABLE 1.

[†] Department of Computer Science and Engineering, Faculty of Applied Sciences, University of West Bohemia, Plzen, Czech Republic (skala@kiv.zcu.cz; http://herakles.zcu.cz)

The key advantages of hash function use are:

- very low expected complexity (*O(1)* expected) for a query if the item is already stored in the hash data structure,
- it is easy to implement.

TABLE 1
Complexity of various approaches to duplicate elimination

	Batch processing complexity (for all items)	Insert one item with duplicate elimination
Sort and duplicate elimination	*O(M lgM)*	*O(N)*
Using a tree including balancing	*O(M lgM)*	*O(lgN)* without balancing
Hash function use (expected)	$O(M*I_a)$	$O(I_a)$
Hash function use (the worst case)	$O(N^2)$	*O(N)*

There are two approaches to the hash function design:

1. *The perfect hash function* design is applicable to the final data sets that are not expected to change, and its computation is of *O(M)* expected complexity for the given data set [9]. The perfect hash function gives a unique index for each item from the data set. *The minimal perfect hash function* is the perfect hash function for which the hash table has no holes, i.e. the size of the hash table is equal to the number of items. This hash function can be made for a static list only and it is usually referred to as *the dictionary problem* [1].
2. The hash function design is based on experience with recently designed hash functions. Such an approach must be used in the case where the hash table is build incrementally. However some problems will occur:
   - To design a hash function properly, the fundamental requirement is that the number of collisions must be as small as possible. Collision occurs when different items are transformed to the same index to the hash table.
   - There can be a problem with memory requirements as the size of the hash table rises, particularly as the functionality of the hash function depends on the hash table length.

The hash function has been used effectively in several geometrical applications [Gla94a], for the duplicate elimination among geometric entities. The experiments with geometric applications made recently [10], [11] raised a question whether a similar approach can be taken for string-based problems as well, especially for large data sets and for batch and incremental processing as well.

3. Previous work. The perfect hash function cannot be used for cases when the data sets are built incrementally. In these cases a hash function design is based on experience. Nevertheless this approach is not reliable as there is no always-optimal hash function and final results might be quite strange.

The recommendation for the hash function *h(x)* is usually of the form [6]:

$$index = h(\mathbf{x}) = (\alpha * x + \beta * y + \gamma * z + ...) \mathbf{mod}\ HS \quad (1)$$

where:

α, β, γ - are the coefficients - mostly prime numbers are taken,
x, y, z - are the values that form a string,
index - is the evaluated index to the hash table, see FIG. 1.

When this hash function is used it is important to anticipate collisions, and to prepare effective solutions for these events. If parameters α, β, γ are not selected properly long clusters might be produced, causing unwieldy sequential searches. Re-hashing, overflow areas and chaining can be used to solve collisions effectively, see [3], [4]. In our approach the chaining technique in a separate memory is used, see FIG. 1. If the hash function is well designed, and the maximal cluster length is close to one, then a query if the item is already stored has expected *O(1)* complexity and duplicate elimination is therefore of *O(M)* expected complexity for the whole data set.

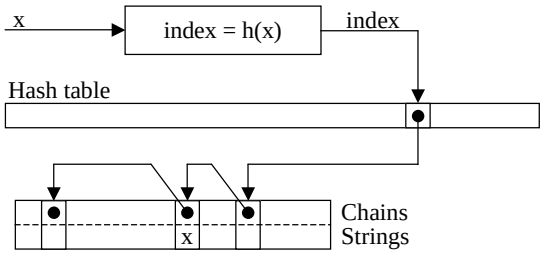

FIG. 1. *The data structures used*

Properties of hash function. The most important hash function properties are:

- to keep the cluster lengths as low as possible,
- to cope well with large data sets,
- to keep the hash function simple to speed up evaluation,
- to use all cells of the hash table as much as possible , i.e. to minimize the number of empty cells.

To be able to compare different hash functions it is necessary to introduce some general criteria. Assume that there are already N items stored in the data structure and I is the cluster length. Three basic situations can occur when a new item i.e. a string is inserted to the structure:

1. The item is not stored in the data structure and the appropriate cluster is empty. The item is inserted to this cluster. The cost of this operation for all such items can be expressed as

   (2) $$Q_1 = 0$$

2. The item is not stored in the data structure so the whole cluster is to be searched and the item is to be inserted to an appropriate cluster. The cost of this operation for all such items can be expressed as (because the cluster of the length I must be searched for all items in this cluster I-times, value I is powered by two)

   (3) $$Q_2 = \sum_{I=1}^{I_m} I^2 \, C_I$$

3. The item is stored already in the data structure so the corresponding cluster is to be searched and the item is not inserted to the appropriate cluster. Because only half of the cluster is to be searched on average, the cost of this operation for all such items can be expressed as

   (4) $$Q_3 = \sum_{I=1}^{I_m} \frac{1}{2} I^2 C_I$$

It is necessary to point out that the cost of the hash function evaluation has not been considered, as it is the same for all cases. The cost of item insertion to a cluster was omitted[1]. The final criterion can be expressed as

$$Q = Q_1 + Q_2 + Q_3 = \sum_{I=1}^{I_m} \frac{3}{2} I^2 C_I \quad (5)$$

Empty clusters are not considered by this criterion because the hash table length *HS* depends on the number of items stored. It can be seen that the criterion *Q* depends on the number of items. We used a relative criterion to evaluate properties of hash functions for different data sets with different sizes defined as

$$Q' = \frac{Q}{N} \quad (6)$$

Several experiments with coefficients α, β, γ for the hash function defined by Equation 3.1 were made recently. These coefficients were taken as decimal numbers and hash function behavior was tested for large data sets with very good results being obtained for geometrical applications, see [8], [10], [11]. These results encouraged us to apply the functions to large dictionaries. For these purposes two dictionaries, Czech and English, were used [5].

4. Proposed solution. There were several significant results from the previous experiments with large geometrical data sets. The most important assumptions of our approach have been:

- large available memory for applications is considered,
- the load factor *f* (will be defined later in this paper) should be smaller than or equal to *0.5*,
- the hash function value should be in the interval of $< 0; 2^{32}-1 >$ before the *modulo* operation is used to get a better spread for all considered items,
- the expected number of items to be stored is in the range $< 10^5; 10^7 >$ or higher,
- hash functions used for strings must be different from the hash function used in geometrical applications, because the strings can have various length, i.e. non-constant dimensionality, and characters are taken from a discrete value set[2],
- non-uniform distribution of characters in dictionaries will not be considered in order to obtain reasonable generality and simplicity.

Hash table size. The size of the hash table depends on the number of items to be stored in the data structure and also on the load factor *f*. The load factor *f* is a ratio of the number of items (strings) stored and the total length of the hash table. The recommended value for the load factor is 0.5 [7]. The hash table size is determined as

$$HS = 2^{\left\lceil \log_2 (\frac{1}{f} N) \right\rceil} \quad (7)$$

If the number of unique elements *N* in the data set is unknown, it is possible to use *M*, i.e. the number of elements with duplicates, or to take some estimation. The relationship between hash table size and the number of elements has been widely studied [2], [10,11].

Hash function. A hash function for strings must be designed with a different strategy as strings consist of different number of characters represented by discrete values and the number

[1] The item is always placed at the beginning of the cluster

[2] In geometrical applications the dimensionality is usually constant and values are from an "unlimited" value set (a set of possible decimal values)

of characters in an alphabet is limited, i.e. the dimension varies and the range of values is fixed. On the other hand a simple function is needed in spite of the fact that the string length is not constant. Considering this fact a simple polynomial function has been selected, generally defined as:

$$h(x) = \left\lfloor c \sum_{i=0}^{L_x - 1} x_i q^i \right\rfloor \textbf{mod } HS \tag{8}$$

The form of this hash function is based on an idea that all possible strings should be transformed in to different indices in the hash table. It is well known that using *the modulo operation* (**mod**) improves the spread of items over the hash table. If the length of the hash table is the power of two modulo operations can be implemented using binary **and** operation. Because of the various length of strings we need to make this sum convergent and therefore q values must be taken from the interval of $(0; 1)$. Because of that it is possible to compute the maximal value of the Equation 8:

$$h_{\max}(x) = \max_{\forall x} \{h(x)\} = \frac{L_{\max}}{1-q} \tag{9}$$

Also the constant c must also be determined. The purpose of this constant is to obtain the maximal range of logical indices that are transformed to the actual size of the hash table HS using the modulo operation. It means that the value of the hash function $h(x)$ is mapped to the interval $\langle 0; 2^{32}-1 \rangle$ if a 32 bits representation for integer is used. The constant c is computed as:

$$c = \frac{\left(2^{32}-1\right)}{h_{\max}(x)} = \frac{\left(2^{32}-1\right)(1-q)}{L_{\max}} \tag{10}$$

If the hash table size HS is the power of 2 then the **mod** function can be replaced with binary **and** operation that is faster.

$$h(x) = \left\lfloor c \sum_{i=0}^{L_x - 1} x_i q^i \right\rfloor \textbf{and } (HS - 1) \tag{11}$$

This function works well for the English dictionary but for the Czech dictionary the results were not so good, because of the different structure of the language. The Czech language uses prefix and suffix connected with the kernel of the word as all Slavonic languages do. Because the suffix varies more (the characters and also its length) in Slavonic languages than the prefix and the root, it is better to process strings from the end.

Therefore the hash function for the Czech language is actually computed similarly but starting from both sides towards to the end and to the beginning of the string. The computing complexity of this hash function is $O(L_x)$.

5. Experiments. Both hash functions were tested using Czech and English ISPELL dictionaries [5]. The data sets for both these languages were taken from the `ispell` package for spell checking. The Czech dictionary contains approx. $2.5 * 10^6$ words and the English dictionary contains approx. $1.3 * 10^5$ words. Several experiments were made including slight modifications for the Czech language.

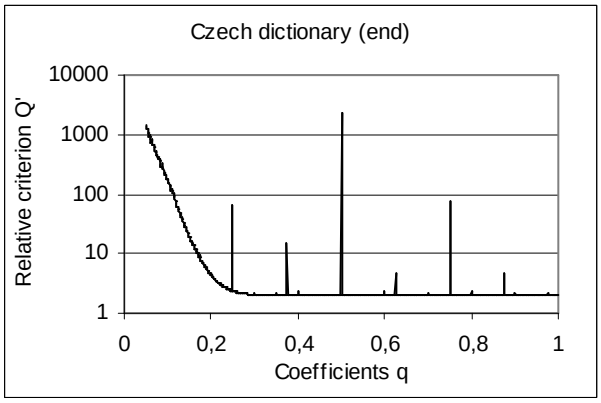

FIG. 2. *Relative criterion for Czech dictionary (I_a : min. 1.15, I_m : min. 6)*

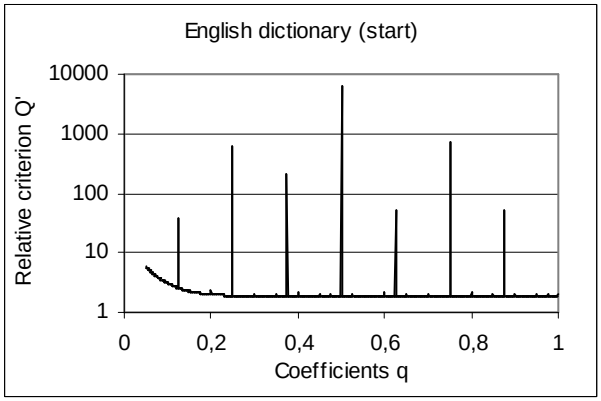

FIG. 3. *Relative criterion for English dictionary (I_a : min 1.13, I_m : min 4)*

The properties of the hash function are very poor at the interval *(0; 0.3)*. This behavior is caused by the non-uniform frequency of characters in words.
Perfect overlapping of some words causes the peaks for the multipliers of 0.125. For example: the strings “ab”, ”aac”, “aaae”, … will have exactly the same value.

It can be seen from the graphs that the properties of the hash function were improved in the mean of lower relative criterion.

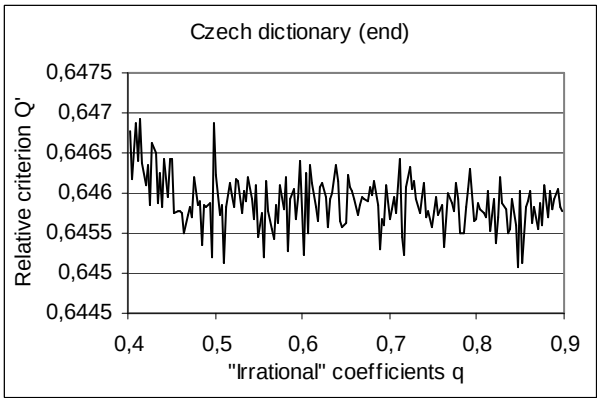

FIG. 4. *Relative criterion for Czech dictionary*

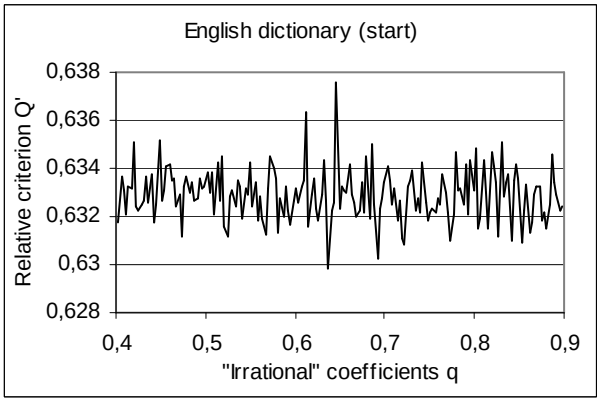

FIG. 5. *Relative criterion for English*

Varying the size of the hash table. Some small improvement can be expected if the size of the hash table is increased and dramatic changes in behavior can be expected if the hash table is half of the designed length. Such behavior has been proved in another experiments, see FIG. 6-7.

Note that the sizes of the hash table varied in interval *<1,048,576; 33,554,432>* for the Czech dictionary and *<65,536; 2,097,152>* for the English dictionary, according do Equation 4.1.

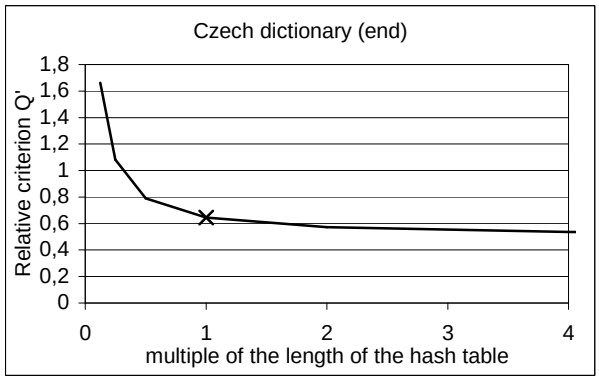

FIG. 6. *Relative criterion for Czech dictionary when varying size of the hash table (I_a : 1.02-2.58, I_m : 4-12)*

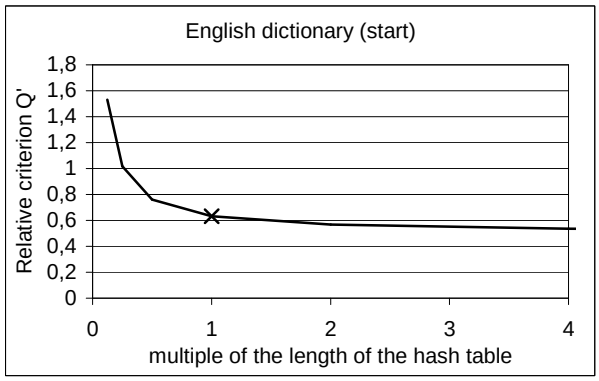

FIG. 7. *Relative criterion for English dictionary when varying size of the hash table (I_a : 1.02-2.36, I_m : 3-10)*

Additional experiments. The partial conclusions drawn from recent experiments were also supported by additional experiments with other languages. In TABLE 2 are the general properties of the dictionaries of the selected languages.

TABLE 2

The general properties of additional languages

Language	*M*	*N*	Designed HS
French	285992	220291	524288
German	309838	294899	1048576
Hebrew	10669	10669	32768
Russian	963212	956715	2097152

The selected dictionaries were tested using the same hash functions as the Czech and English dictionaries. The Russian dictionary was processed like the Czech dictionary (both of them are Slavonic languages), i.e. the words are processed from the end. The hash function that processed words from the start on the other languages was used for the rest of the dictionaries. The results obtained are presented in TABLE 3.

TABLE 3
The properties of proposed hash function for additional languages

Language	I_a	I_m	Q'
French	1,22	5-7	~0,707
German	1,15	5-8	~0,638
Hebrew	1,16	4-6	~0,652
Russian	1,24	6-8	~0,726

It is obvious from TABLE 3 that the behavior of the proposed hash function with other languages is similar to that with Czech and English dictionaries.

6. Conclusion. This paper presents a new hash function with stable properties convenient for textual data processing. The behavior of the hash function has been tested on Czech and English dictionaries as these two languages belong to different language groups. Additional experiments made with four other languages proved properties of the proposed approach.

For the proposed data structure the optimal hash table length was derived and also the recommendations for q values were verified. It was proved that the Slavonic languages (in proposed experiments Czech and Russian) should be processed from the end because of the different language structure (more suffixes and fewer prefixes combined with each word). Also the influence of hash table length was experimentally verified.

It has been proved that the proposed hash function can be used for incremental processing, i.e. the hash table size designed from the initial size of the data set is also satisfying if the size of the data set is doubled. The proposed hash function can be used in situations where the final size of the data set is known only approximately and it also must be guaranteed that the hash function have good properties even for larger data sets.

The proposed hash function has low complexity. With the average cluster length ~1.17 the complexity $O(M*I_a)$ is very near to the $O(M)$ complexity.

7. Acknowledgement. The authors would like to thank to all who contributed to this work, especially to colleagues, MSc. and PhD. students at the University of West Bohemia in Plzen who have stimulated this work. The project was supported by the project VIRTUAL 2C006 of the Ministry of Education of the Czech Republic.

REFERENCES

[1] Gettys, T. (2001) : Generating perfect hash function, *Dr. Dobb's Journal*, Vol. 26. No.2, pp.151-155.

[2] Glassner,A. (1994): "Building Vertex Normals from an Unstructured Polygon List", *Graphics Gems*, IV, pp.60 - 73. Academic Press, Inc., Cambridge.

[3] Horowitz,E., Sahni,S.: *Fundamentals of Data Structures*, Pitman Publ.Inc., 1976

[4] Morris,J., Hash Tables: *http://swww.ee.uwa.edu.au/~plsd210/ds/hash_tables.html*

[5] SPELL Dictionaries, *http://ficus-www.cs.ucla.edu/geoff/ispell-dictionaries.html*

[6] Knuth,D.,E. (1969-90) *The Art of Computer Programming, vol. 3, Searching and sorting*, Addison-Wesley.

[7] Korfhage,R.,R., Gibbs,N.E. (1987) : *Principles of Data Structures and Algorithms with Pascal*, Wm.C.Brown Publishers

[8] Kuchar,M., (supervisor V. Skala) (2000) : *Construction of the triangular meshes from STL Data Format and Stereoscopic visualization*, MSc. thesis. University of West Bohemia, Plzen, Czech Republic.

[9] Pagh,R. (*1*663) : Hash and Displace: Efficient Evaluation of Minimal Perfect Hash Functions *WADS '99, LNCS*, pp.49-54. Springer-Verlag.

[10] Skala,V., Kuchar,M. (2000) : Hash Function for geometry Reconstruction in Rapid Prototyping, *Algoritmy 2000 Int.Conf. proceedings*, pp.279-384, Slovakia.
[11] Skala,V., Kuchar,M. (2001) :The Hash Function and Principle of Duality, *IEEE CGI proceedings*, pp. 167-174, 2001, Hong Kong.