Eighth International Workshop on Optimal Codes and Related Topics
July 10-14, 2017, Sofia, Bulgaria *pp. 81-86*

An implementation of hash table for classification of combinatorial objects[1]

Mariya Dzhumalieva-Stoeva mdzhumalieva@gmail.com
Venelin Monev venelinmonev@gmail.com
Faculty of Mathematics and Informatics, Veliko Tarnovo University,
5000 Veliko Tarnovo, Bulgaria

Abstract. An approach for efficient implementation of storing a set of combinatorial objects using hash table data structure is presented. It is suitable for classification algorithms.

1 Introduction

Given is a finite set of combinatorial objects of a certain type Ω, concerned with an equivalence relation $\cong$. The classification problem consists of finding a subset $T \subset \Omega$, such that:

1. $\forall a \in \Omega, \exists b \in T : a \cong b$,
2. $\forall a, b \in T \Rightarrow a \ncong b$.

In other words, the set T contains exactly one representative of each equivalence class of the relation $(\Omega, \cong)$.

An important part of every classification algorithm is the test whether two given objects $x, y \in \Omega$ are isomorphic $x \cong y$ (isomorphism problem). In general such a test must be performed for each two objects $x, y \in \Omega$, so the classification becomes computationally heavy task.

There exist approaches, which reduce the number of performed isomorph tests - isomorph free generation. The main types are via recorded objects, orderly generation (based on canonical form of the objects of interest) and canonical augmentation [4]. Even though, in case of large number of combinatorial objects, it is hard to implement algorithms, working for appropriate time and using suitable amount of computer memory.

We consider a hash table implementation for recording nonisomorphic combinatorial objects, which can be easily combined with canonical representative map and canonical augmentation isomorph free generation algorithms.

[1]This research was partially supported by Grant DN 02/2/13.12.2016 of the Bulgarian National Science Fund.

2 Preliminaries

The most of the combinatorial objects have computer representation as binary or $\{0,1\}$-matrices, so that the isomorphism problem of various types of objects (linear codes, graphs, designs, etc.) can be reduced to binary matrices isomorphism problem [2]. So, we present the approach in terms of binary matrices.

Let A be a new generated binary matrix and $T' \in T$ is the current set of nonisomorphic matrices. Two matrices of the same size are *isomorphic* if the rows of the first one can be obtained from the rows of the second one through a permutation of the columns. The recorded object technique consists of isomorph check between A and all matrices in T'. Shortly a matrix $A \in \Omega$ is included in the set T' if $A \ncong B, \forall B \in T'$, otherwise A is rejected. Binary matrices isomorph test itself is a hard task [1], therefore we consider a subset $U \subset \Omega$ of objects in canonical form, obtained according to a certain canonical representative map, defined as follows:

Definition 1. *A canonical representative map is a function $\rho : \Omega \mapsto \Omega$ which satisfies the following properties:*

1. *for all $X \in \Omega$ it holds that $\rho(X) \cong X$,*
2. *for all $X, Y \in \Omega$ it holds that $X \cong Y$ implies $\rho(X) = \rho(Y)$.*

The object X is in canonical form if $\rho(X) = X$.

In this case, to test whether $A \cong B$ is to test their canonical forms for equality. Obviously, each equivalence class contains exactly one element in canonical form. If T' consists of canonical forms, the equivalence class of an object A is already represented if and only if $\rho(A) \in T'$. Otherwise, if $\rho(A) \notin T'$ than $T' = T' \cup \{\rho(a)\}$. The idea of recording objects in canonical forms is presented in the following algorithm:

Procedure RejectionByRecordCan
Input: U -set of binary matrices
Output: T -set of binary matrices in canonical form
1] $T' \leftarrow \emptyset$
2] **for(** $\forall a \in U$ **) do**
3] **obtain** $\rho(a)$
4] **if(** $\rho(a) \notin T'$**)**
5] $T' \leftarrow T' \cup \{\rho(a)\}$
6] **end if**
7] **end for**
8] $T \leftarrow T'$
end procedure

An important part of the algorithm is how to look up fast and store an element $\rho(A)$ to the set T'. In this paper we will focus on the effective implementation of this problem. One of the easiest implementations of storing the objects is the use of single-linked list data structure. In this case, the time complexity of the operations "searching", "adding" and "deleting" is a linear function, depending on the number of objects in the set T', which is ineffective approach.

An appropriate method is using a *hash table* data structure, which allows the requested operations to be performed in a constant time [4]. Moreover, all the entries of the hash table will be unique as in the set T.

Each element of the given set is represented in the hash table via two fields: key and value. The key is calculated by specific hash function of the stored object, while the value is the object itself.

Definition 2. *A hash function* $\boldsymbol{h}$ *is any mathematical function of the form*

$$\boldsymbol{h} : \Omega \mapsto \{0, 1, \ldots, s-1\},$$

In other words the hash function $\mathbf{h}$ maps an integer number to each object of Ω. Practically, a hash table of size s can be implemented as one dimensional array $HT[0 \ldots s-1]$, which cells are usually called *slots*. In this case the key $\mathbf{h}(a)$ is the index of an element a, so that the slot $HT[h(a)]$ contains an essential information about the object a.

Usually the result of a hash function is a large integer which exceeds the number of slots in $HT[0 \ldots s-1]$, so we can take the remainder of $\mathbf{h}(a)$ modulus s. The size s of the hash table is often chosen to be a large prime not too close to $2^i - 1$, $i \in \mathbb{N}$, hence the hash values can be fairly well distributed [5].

A good distribution is desirable, but it is possible that two distinct elements $a_1 \neq a_2$ have the same hash value $h(a_1) = h(a_2)$, which is called *collision*.

3 Implementation of the hash table

Hash tables are widely used in fast data searching (digital dictionaries, programming language compilers, etc). In case of small data amounts, the RAM computer memory is sufficient for the implementation. Otherwise, the objects need to be stored in files on the hard drive. One of the first hash table implementations using dynamic files is connected to databases, but it has heavy file organization and ineffective memory usage.

There are two main problems about the implementation of a hash table: the first one is using an appropriate hash function and the second one is carefully selecting a method for collision resolution. The most common methods of collision resolution are chaining and open addressing [3].

We present an implementation of a hash table for storing huge amount of binary matrices of different sizes. The hash table is separated into two

structures: an one-dimensional integer array of fixed size and a random access file of records. Each record is available by its beginning position and size. Let $a \in \Omega$ be a matrix and $A = \rho(a)$ is the canonical form to be stored in the hash table. The value A is appended as a record to the file, while its beginning position is written in the array slot with index $h(A)$. If the slot is not empty, either the equivalence class of A is already represented in T or a collision has occur. In our case the collision resolution is implemented by chaining via linked list structure. Hence, the first field of each record contains a pointer to the position of the next record with the same key. The hash table implementation is given by the following procedure:

Procedure HashTableLookUp
Input: A - matrix in canonical form $\rho(a)$
Output: boolean value **stored**: true or false
1] key ← **h(**A**)**;
2] **if(** slot HT[key] is empty **)**
3] append A to file;
4] HT[key] ← position of A in the file;
5] return stored ← true;
6] **else**
7] currentRec ← record in position HT[key];
8] **while(** currentRec.next ≠ end of list **);**
9] **if(** currentRec.matrix **is equal** to A **)**
10] return stored ← false;
11] currentRec ← currentRec.next;
12] **end while**
13] **if(** currentRec.next is end of list **)**
14] append A to file;
15] currentRec.next ← position of A in the file;
16] return stored ← true;
17] **end if**
18] **end else**
end procedure

In general, the key $h(A)$ is unique, so if the corresponding slot is not taken the matrix is simply added to the file (rows 2-4). In case of collision a linked list of objects having one and the same key is constructed. The new matrix is compared to each matrix of the list. It is rejected in case of equality (the corresponding equivalence class is already represented). Otherwise it is appended to the file as last element of the current list (rows 6-15). The procedure itself goes to rows 4-6 of the procedure RejectionByRecordCan.

4 Experimental results and conclusion

The algorithm is implemented in C programming language on a computer with CPU Intel Core i7-6700, 3,40GHz, 16 GB RAM memory, having an ordinary HDD. To test the procedure we use a set of binary random binary matrices, stored in an input file. We assume that they are already in canonical form. If two matrices coincide then they are representatives of one and the same equivalence class. We use the FNV-1a hash function [6], which hashes strings. So, each matrix is written row by row and converted to string of printable ASCII characters.

The main results are given in the table below. The number of the matrices and their sizes are listed, as well as the hash table size, the number of collisions and the running time of each test. The last row of the table shows the running time in case of using single linked list structure for storing the matrices of the set T.

♯ matrices	size $n \times m$	memory cells	collisions	Running time
1 000 000	100×100	524309	2	696.14 sec
100 000	$50 \leq n, m \leq 100$	65537	9	33.737 sec
100 000	$50 \leq n, m \leq 100$	131101	6	31.659 sec
100 000	$50 \leq n, m \leq 100$	262147	6	31.860 sec
100 000	100×100	65537	2	51.541 sec
100 000	100×100	131101	2	51.706 sec
100 000	100×100	262147	2	50.323 sec
100 000	100×100	1	-	1115.695 sec $\approx$ 19 min

Table 1: Main results

As a conclusion we give the following remarks:

- The procedure HashTableLookUp is called for each input matrix. Its running time is insignificant (less than 0.01 sec) but the total execution time also includes the reading from the input file.

- The number of collisions depends on the hash table sizes and the type of the objects.

- As we don't need to exclude objects of the set T' during the algorithm, the operation "deleting" is not implemented. There are no empty records in the output file. As the computer memory is well managed, the searching operation goes faster.

- If it is necessary, more than one file could be used for the implementation of the hash table.
- The algorithm can be easily implemented for the non binary case.

References

[1] I. Bouyukliev, About the code equivalence, in *Advances in Coding Theory and Cryptology*, T. Shaska, W.C. Huffman, D. Joyner, V. Ustimenko, Series on Coding Theory and Cryptology, World Scientific Publishing, Hackensack, NJ, 2007.

[2] I. Bouyukliev and M. Dzhumalieva-Stoeva, Representing Equivalence Problems For Combinatorial Objects, *Serdica Journal of Computing 8*,No 4, 327-354, 2014

[3] Th. H. Cormen, Ch. E. Leiserson, R. L. Rivest, Cl. Stein, Introduction to algorithms, Third edition, The MIT Press Cambridge, Massachusetts London, England, 2009.

[4] P. Kaski and P. R. J. Östergård, Classification Algorithms for Codes and Designs, *Springer-Verlag, Berlin Heidelberg*, 2006.

[5] St. Skiena, The Algorithm Design Manual, Second Edition, Springer-Verlag London, 2008.

[6] Fowler-Noll-Vo hash funciton - FNV-1a alternate algorithm (http://www.isthe.com/chongo/tech/comp/fnv/index.html).