



PII: S0031-3203(96)00187-2

IMAGE DATABASES AND NEAR-PERFECT HASH TABLE

CHAMAN L. SABHARWAL^{†*} and SANJIV K. BHATIA[‡]

[†]Graduate Engineering Center, University of Missouri–Rolla, St. Louis, MO 63121-4499, U.S.A.

[‡]Department of Mathematics and Computer Science, University of Missouri–St. Louis,
 St. Louis, MO 63121-4499, U.S.A.

(Received 14 May 1996; in revised form 19 November 1996)

Abstract—The 2D-string is one of the important data structures to represent spatial information in images and has been used to perform queries in image database systems. A 2D string can be readily converted to a set of triples which can be used to generate a perfect hash table for indexing the image database. The perfect hash table allows for $O(1)$ retrieval in image database but the computation of the hash table itself, by the fastest heuristic algorithm, is of exponential time complexity. Every time the database is modified, the hash table needs to be recomputed. This limits the use of the hash table to applications where the image database is relatively fixed, such as the ones found on a CD-ROM. In this paper, we present an enhancement of the perfect hash table to allow for insertion and deletion in the hash table. The new hash table allows for a relatively small number of collisions and is called *near-perfect hash table*. It provides the flexibility of a live database while closely approximating the efficiency of retrieval with the perfect hash table as shown by the asymptotic analysis of the search procedure. © 1997 Pattern Recognition Society. Published by Elsevier Science Ltd.

Image database systems Hash addressing Storage Retrieval Indexing

1. INTRODUCTION

An image database system is concerned with the problem of storage, retrieval, and manipulation of pictorial data in an efficient manner. The nature of data being managed (pictures) does not readily lend itself to the conventional database techniques for data organization and retrieval. The problem is harder as the pictorial data cannot be characterized in non-ambiguous terms due to the large number of parameters involved. These parameters can be shapes (round or near-round), size (relative to other objects *v.* absolute), or spatial location relative to other objects in the image.⁽¹⁾ In this paper, we will be primarily concerned with the last parameter, *i.e.* the storage and retrieval of images based on the relative location of objects within them.

The representation of pictorial objects and the spatial relationship between them is one of the most important problems in the image database design.^(2,3) The problem is generally solved by assuming the existence of low-level image processing primitives to reduce an object to determine its exact spatial location or centroid.⁽⁴⁾ This reasoning allows for better organization of data and leads to robust techniques in spatial reasoning.⁽⁵⁾ Ideally, we will like to build the representation by automatically extracting the objects in a picture with image processing primitives. Such an automatic process to build the representation eliminates the need to tag each picture with its syntactic description to be used for storage and retrieval while allowing for content-based retrieval.

In content-based retrieval, the database does not contain any keywords assigned by the humans to describe the

contents of the pictures. If the query is specified in terms of a picture showing the spatial relationship between different objects, the problem of matching the objects in the query against the objects in the images in the database using graphical techniques becomes a daunting task. Therefore, the pictures in the database are processed to identify the different objects contained within them and the objects are then mapped onto symbols (or *iconic objects*) that can facilitate retrieval. Such techniques are exemplified by data structures like 2D strings^(6–10) and quad trees⁽¹¹⁾.

The idea of a 2D string and its use in the syntactic and semantic abstraction of pictures for developing picture indices was advanced by Chang.⁽⁶⁾ It was used to build a prototype image database system that supports spatial reasoning, efficient retrieval of images, and other traditional image database operations.⁽⁴⁾ Later, the 2D strings led to the development of another data structure—a set of triples—to represent the spatial relationship between each pair of objects in a picture.⁽¹²⁾ The triples were used to index an entire image database based on a perfect hash table leading to an $O(1)$ retrieval of desired images.

A perfect hash table does not permit any collisions on any of its addresses. Therefore, the retrieval process is simplified to simply computing the hash address from the specified spatial constraints in the picture depicting the query. The hash address contains a pointer to a linked list of pictures that contain the specified pattern. Thus, each retrieval based on a constraint that can be specified in a single triple can be performed in $O(1)$ time whereas the retrieval of pictures based on m triples is performed in $O(m)$ time.

Ideally, the hash table contains just as many addresses as the number of triples. Since there are no collisions,

* Author to whom correspondence should be addressed.

each triple is mapped to a separate address. Such a hash table is known as the *minimal perfect hash table*. In the context of image databases, the number of addresses in the minimal perfect hash table is the same as the total number of triples across all the pictures in the database. In reference (13), we showed that Chang and Lee's algorithm⁽¹²⁾ could not guarantee a minimal perfect hash table, because of conflicts in address computations that could not be resolved within the minimal address space. Thus, the constraint of minimality on perfect hash table makes the problem of hash table construction computationally expensive. We proposed heuristics to speed up the computation of the hash table while minimizing the address space requirements as compared to Chang and Lee's hash table.

Despite the speed of retrieval, the hash table has seen limited functionality in image databases due to the amount of time needed to compute the table. The problem becomes more acute if the database is frequently updated by insertion and deletion of images because such an update requires the computationally expensive reconstruction of the hash table index. These constraints limit the application of hash table indexing to relatively static databases such as the ones found on the CD-ROMS.

There has been some research performed in the area of perfect hash tables. For example, Czech and Majewski⁽¹⁴⁾ have proposed a linear time hashing algorithm for hashing words based on their probabilistic distribution. The hash value of a word is expressed in terms of three pseudo-hashing functions. Other dynamic perfect hashing functions based on probability distributions are given in reference (15). However, in our treatment, the hash function deals with image objects and their interrelationships. Also, we make no assumptions about the distribution of objects and hence, cannot use the conventional hashing techniques.

In this paper, we present an enhancement of the minimal perfect hash table structure that can allow the dynamic insertion and deletion of pictures in an image database. We call the new data structure a *near-perfect hash table*. The near-perfect hash table allows for a maximum of 10% of its addresses to resolve collisions and may contain some unassigned addresses. We present the techniques to update the hash table whenever pictures are added to or deleted from the image database. We also perform an asymptotic analysis to show the effectiveness of retrieval of pictures with the modified index.

In Section 2, we briefly outline the 2D string data structure, development of triples from the 2D string, and the algorithm to compute the hash table index. In Section 3, we describe the near-perfect hash table data structure and the associated deletion and insertion operations. In Section 4, we present the asymptotic analysis of retrieval with different possibilities in the near-perfect hash table.

2. PERFECT HASH TABLE

A perfect hash table is used to provide an index into an image database. The index facilitates an $O(1)$ retrieval of

images based on simple queries. The hash table is computed from a set of triples derived from the 2D string data structure. In this section, we present a brief introduction to the computation of 2D strings from a given image. This is followed by derivation of triples from the 2D strings. Finally, we present an algorithm to compute the hash table index from the set of triples.

2.1. 2D String data structure

The 2D string data structure was advanced by Chang^(6,7) and is particularly suitable to satisfy the queries that require retrieval based on the spatial relationship between two or more objects in a picture. In the 2D string approach, each picture is processed using low level image processing primitives to identify different objects contained therein. In particular, each object is identified by taking a projection of its boundaries along the x - and y -axis. These projections enclose each object in a rectangle, known as the *minimum bounding rectangle* (MBR), and help define the bounds and centroid of the symbolic object. The position of each symbolic object in the image is taken to be the centroid of the MBR. The spatial relationship between objects is represented using the centroid of each object. The set of symbolic objects for the entire image database forms the *vocabulary* of the database.

The general form of a 2D string is an ordered pair (u, v) where each of u and v denote a 1D string containing the spatial relations between symbolic objects along the x - and y -axis, respectively. The 2D string uses a set of relational operators to show the spatial relationship between pairs of objects along each axis. The set of relational operators, denoted by A , is given by

$$A = \{=, <, :\}.$$

Consider two symbolic picture objects S_1 and S_2 . $S_1 = S_2$ along an axis implies that the projections of the two objects coincide along that axis. $S_1 : S_2$ implies that the centroid of both the objects lies in the same MBR. $S_1 < S_2$ in u implies that S_1 is "to the left of" S_2 , whereas $S_1 < S_2$ in v implies that S_1 "is below" S_2 .

A general representation for a 2D string is given by

$$(S_1x_1S_2x_2 \cdots x_{n-1}S_n, S_{p(1)}y_1S_{p(2)}y_2 \cdots y_{n-1}S_{p(n)}),$$

where S_i are the symbolic objects, $p : \{1, \dots, n\} \rightarrow \{1, \dots, n\}$ is a permutation over $\{1, \dots, n\}$, and each of $x_1, \dots, x_{n-1}, y_1, \dots, y_{n-1} \in A$.

The 2D string can be illustrated by using a simple line drawing image presented in Fig. 1.

In Fig. 1, there is a canoe outside a lakefront house. The objects of interest in this picture are enumerated as canoe, house, and lake. The minimum bounding rectangles corresponding to the objects and their centroids are presented in Fig. 2.

The 2D string corresponding to Fig. 1 can be given by

$$(\text{canoe} < \text{lake} = \text{house}, \text{lake} < \text{canoe} < \text{house}).$$

In simple terms, it implies that the centroid of the lake and the house coincide along the x -axis and the

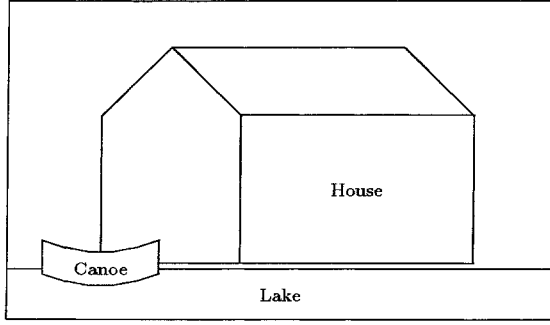


Fig. 1. A simple line drawing of a canoe outside a house.

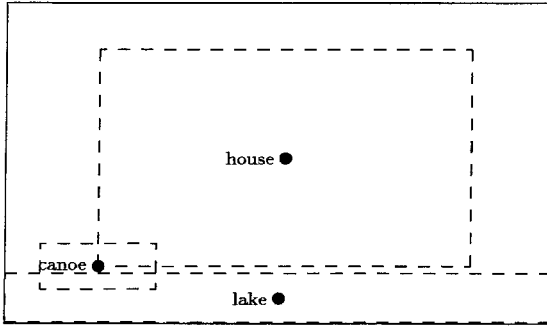


Fig. 2. Minimum bounding rectangles and centroids of objects.

centroid of the canoe is to the left of both of them. Along y-axis, the centroid of the lake is below the centroid of the canoe which itself is below the centroid level of the house.

2.2. Development of triples

A 2D string can be easily converted to a set of triples which can be used to develop the hash table for storage and retrieval.⁽¹²⁾ Each triple is of the form (S_i, S_j, r_{ij}) where $1 \leq r_{ij} \leq 9$ represents the spatial relation between the symbolic objects S_i and S_j . Each triple allows us to map the relationship between two objects in one of the nine relative directions—north, north-west, west, south-west, south, south-east, east, north-east, and coincidental. The nine relative directions are mapped onto integers from 1 to 9 as shown in the Fig. 3. In each mapping, the object at the center is considered as the reference object. In the terminology of triples, the second symbol is treated as the reference symbol.

Given a 2D string, it is straightforward to obtain a set of triples corresponding to it. For example, if the 2D string for two symbolic objects S_i and S_j is $(S_i < S_j, S_j < S_i)$, it is easy to infer the spatial relationship between the two symbols into a triple as $(S_i, S_j, 2)$, or equivalently, $(S_j, S_i, 6)$. A simple algorithm will pick one symbolic object S_i and determine its relationship with respect to every other object along the x- and y-axis to form triples. The object S_i is then eliminated from further consideration and the previous step is repeated with the rest of the objects.

NW	N	NE
W	C	E
SW	S	SE

 $\Leftrightarrow$

2	1	8
3	9	7
4	5	6

Fig. 3. Integer mapping of spatial relations.

Using the mapping in Fig. 3, the 2D string corresponding to Fig. 1 yields the set of triples as

$$T = \{(\text{canoe}, \text{house}, 4), (\text{canoe}, \text{lake}, 2), (\text{house}, \text{lake}, 1)\}$$

The triples are used to perform indexing of the database by using a hash table generation algorithm.^(12,13,16) The algorithm associates a numerical integer value to each symbolic object which, together with the relationship integer in the triple, is used to compute the hash address. The heuristic-based algorithm presented by Sabharwal and Bhatia⁽¹⁶⁾ is summarized in Section 2.3.

2.3. Computing the perfect hash table

The perfect hash table construction is computationally expensive. Once the hash table is developed, the retrieval process is trivial. In hash table calculation, we have to balance a trade off between the complexity of hash table computation and the savings in the storage space and the speed of retrieval process. In this subsection, we describe the computation of the hash table from the set of triples, and illustrate it with an example from reference (12). Let I denote the example database containing four pictures, identified by P_1, P_2, P_3 , and P_4 . In terms of triples, these pictures are described as

$$P_1 = \{(A, B, 6), (A, C, 6), (B, C, 8)\},$$

$$P_2 = \{(F, G, 3), (F, G, 4), (G, G, 4)\},$$

$$P_3 = \{(B, F, 7)\},$$

$$P_4 = \{(A, B, 7), (A, C, 1), (A, D, 8), (A, E, 7), (B, C, 2), (B, D, 1), (B, E, 9), (C, D, 7), (C, E, 6), (D, E, 5)\}.$$

Let $S = \{S_1, S_2, \dots, S_n\}$ denote the set of all iconic picture objects in the image database. Let T denote the set of all the triples identified from all the images in a database. Each triple in T is of the form (S_i, S_j, r_{ij}) , where r_{ij} is the spatial relation of S_i to S_j . For the example database I , S and T can be described as

$$S = \{A, B, C, D, E, F, G\}, T = P_1 \cup P_2 \cup P_3 \cup P_4.$$

A *perfect* hash function is defined by $h : A \rightarrow \{1, 2, \dots, m\}$ such that h is one-to-one $m \geq |A|$. The function h is a *minimal* perfect hash function if h is onto such that $m = |A|$. The perfect hash function used in this paper is defined by

$$h : T \rightarrow \{1, 2, \dots, m\} h(S_i, S_j, r_{ij}) = a(S_i) + a(S_j) + r_{ij},$$

where m is greater than or equal to the size of the range of h (the number of addresses in the hash table), and $a(S_i)$ is the value associated with the symbolic object S_i .

The original algorithm to compute a hash table for image databases was brute force and was developed by Chang and Lee.⁽¹²⁾ This algorithm was motivated by the work of Sprugnoli,⁽¹⁷⁾ Cichelli,⁽¹⁸⁾ and Cook and Oldhoeft⁽¹⁹⁾ in the domain of programming languages. We designed a set of heuristics to speed up this algorithm.⁽¹³⁾ Later, we developed a new algorithm by relaxing conditions on the values associated with the symbolic objects and performed asymptotic analysis of the algorithm.⁽¹⁶⁾

The algorithm begins by computing the frequency of each symbolic object S_i in the set of triples T . If a symbol occurs twice in a triple, for example triples of the form (S_i, S_i, r_{ii}) , it is assigned a higher frequency value compared to the other symbols. Such triples are known as *isoconic triples* (such as the triple $(G, G, 4)$ in picture P_4 of the example database). The set of objects S is then sorted on the decreasing order of the frequency of occurrence of the objects in T . The ordered S for our I is

$$S = \langle G, B, A, C, E, D, F \rangle.$$

Next, all the triples in T are rearranged such that the frequency of the symbolic object in the first place in the triple has a higher frequency compared to that of the object in the second place. That is, for each triple (S_i, S_j, r_{ij}) , the frequency of S_i is always greater than that of S_j . In case the frequency of S_j is higher than that of S_i , we rearrange the triple to (S_j, S_i, r_{ji}) where $|r_{ij} - r_{ji}| = 4$. This rearrangement of triples is called *intratriple ordering*.

The ordered triples are now sorted on the decreasing frequency of the second object in the triple. The sorting is called *intertriples ordering*. The resulting intratriple- and intertriples-ordered set of triples is used to *search* for the associated values for each picture object, that will map the triple to a unique address in the hash table. The search for associated values consumes most of the time as it backtracks to satisfy the constraint of no collisions in the hash table. The intratriple- and intertriples-ordered T corresponding to I is described by

$$T = \{(G, G, 4), (B, A, 3), (B, A, 2), (B, C, 2), (A, C, 1), (B, C, 8), (A, C, 6), (C, E, 6), (B, E, 9), (A, E, 7), (E, D, 1), (C, D, 7), (B, D, 1), (A, D, 8), (B, F, 7), (G, F, 8), (G, F, 7)\}.$$

The search algorithm presented by Chang and Lee⁽¹²⁾ is summarized as follows:

1. Initialize all associated values to zero.
2. Select the triple at the head of the ordered list of triples.
3. Calculate the hash address for this triple. Generally, the first object in the triple will already have a value assigned due to the ordering in triples.
4. A collision occurs if the corresponding hash address is already in use. Whenever a collision occurs, reassign the current symbol a new value by incrementing the old value by 1. Repeat step 3 using the new value until the collision is resolved. If the collision is resolved, move the current item pointer in the list to point to

the next triple and go to step 3. Otherwise, proceed to step 5.

5. Step 4 fails if the hash address for the triple is beyond the available address space. In this case, backtrack to most recently assigned symbol and start from step 2 by incrementing the current value of this symbol by 1. Go to step 3.

The search algorithm is the computational bottleneck in hash table development. We improved this algorithm by designing several heuristics.⁽¹³⁾ The first heuristic guesses a starting value that is not necessarily zero by using the available addresses. In case of a collision, the second heuristic lays down a plan to adaptively increment the associated value by a number that is greater than or equal to 1. The third heuristic uses a look-ahead mechanism to determine the possibility of a collision at later stages with the current associated value. Another triple normalizes the isoconic triples of the form (S_i, S_i, r_{ii}) to (S_i, S_i, r'_{ii}) if r_{ii} is greater than 4. The normalization assures that r'_{ii} is less than or equal to 4 in all cases.

We showed that even with the above heuristics, the algorithm is not sufficient to guarantee a minimal perfect hash function.⁽¹³⁾ We later developed a new heuristic that is much faster than the previous algorithms and promises the construction of a minimal perfect hash function.⁽¹⁶⁾ The idea is to utilize the spaces left unused by the other algorithms by using negative associated values. It is found that the lower bound on the associated values is $-(m+4)$ where m is the number of triples. The complexity of the algorithm is computed to be $O(m!(m-4)^{n/2})$. The worst case occurs when the minimal perfect hash table may not be achieved. In such a case, the hash table may be constructed with m' additional addresses leading to the computational complexity of $O((m+m')!(m+m'-4)^{n/2}/m'!)$, m' is small as compared to m . It may be noted that we were able to get a minimal perfect hash table for all the examples considered (i.e. $m' = 0$). The minimal perfect hash table corresponding to I using the above algorithm results in the following associated values:

$$\{(A, 1), (B, 9), (C, -1), (D, 5), (E, -1), (F, -7), (G, 2)\}.$$

Lemma 1. The total space requirement for the hash table is less than or equal to twice the number of triples in the hash table, i.e. $m' \leq m$.

Proof. The worst case for the hash table occurs when all the triples in the database are isoconic of the form (S_i, S_i, r_{ii}) . Furthermore, in this case, the value of r_{ii} is of the same parity (even or odd) in all the triples. All the triples in such a case will map to even or odd addresses depending on the value of r_{ii} . Successive triples map to successive even or odd slots leaving half the table unused. Hence, for m triples, the number of addresses in the hash table is at most $2m$, or $m' = m$. $\square$

2.4. Query and retrieval

The hash table is the central data structure for retrieval. In addition to the list of pictures containing different

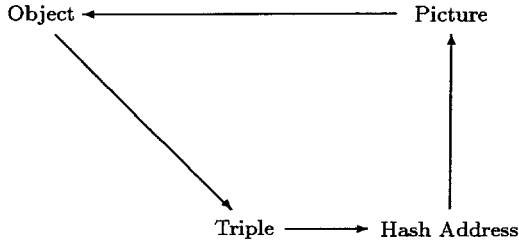


Fig. 4. Data flow/connectivity in the perfect hash table.

spatial relations between symbolic objects, it must contain certain additional information that can help during the query process.

The computation of the perfect hash table and the associated query process is explained with the help of Fig. 4. The pictures in the image database are processed to yield the set of symbolic objects and their spatial relationship to each other. The spatial relation between objects is represented by a 2D string which is then converted to a set of triples. The algorithm to generate the hash table associates the integer values with each object and returns the hash address corresponding to each triple.

The perfect hash table data structure is presented in Fig. 5. The ordered list of symbols in the data structure is required to perform intratriple ordering within each query triple. The associated value for each symbol helps in computation of the hash address. Finally, each address A_i in the hash table contains a pointer to a linked list that contains the addresses of pictures P_{A_j} containing the spatial pattern mapped onto the corresponding hash table address. The picture address can be in the form of a complete path name of the image in the database and can be directly used to retrieve the image. The perfect hash table structure corresponding to the database I is given in Fig. 6. It may be noted that in the example, we did not have a triple occurring in more than one picture. For multiple pictures with the same triple, there will be more than one picture in the linked list at the corresponding address.

When a user poses a query to the system, the symbolic objects in the query are separated and a set of triples is generated corresponding to the query. The query is also converted to a set of intratriple ordering using the frequency distribution information for symbolic objects in the original database (kept with the hash table). Then, the auxiliary information in the hash table is used to deter-

Ordered list of symbols (based on frequency)				
Associated value for each symbol				
Hash table				
A_1	$\triangleright \rightarrow$	$P_{A_1 1}$	$\rightarrow$	$P_{A_1 2} \rightarrow \dots \square$
A_2	$\triangleright \rightarrow$	$P_{A_2 1}$	$\rightarrow$	$P_{A_2 2} \rightarrow \dots \square$
$\vdots$				
A_n	$\triangleright \rightarrow$	$P_{A_n 1}$	$\rightarrow$	$P_{A_n 2} \rightarrow \dots \square$

Fig. 5. The perfect hash table data structure and associated information.

$\{G, B, A, C, E, D, F\}$				
$\{(A, 1), (B, 9), (C, -1), (D, 5), (E, -1), (F, -7), (G, 2)\}$				
1	$\triangleright \rightarrow$	P_4	$\rightarrow$	$\square$
2	$\triangleright \rightarrow$	P_2	$\rightarrow$	$\square$
3	$\triangleright \rightarrow$	P_2	$\rightarrow$	$\square$
4	$\triangleright \rightarrow$	P_4	$\rightarrow$	$\square$
5	$\triangleright \rightarrow$	P_4	$\rightarrow$	$\square$
6	$\triangleright \rightarrow$	P_1	$\rightarrow$	$\square$
7	$\triangleright \rightarrow$	P_4	$\rightarrow$	$\square$
8	$\triangleright \rightarrow$	P_2	$\rightarrow$	$\square$
9	$\triangleright \rightarrow$	P_3	$\rightarrow$	$\square$
10	$\triangleright \rightarrow$	P_4	$\rightarrow$	$\square$
11	$\triangleright \rightarrow$	P_4	$\rightarrow$	$\square$
12	$\triangleright \rightarrow$	P_1	$\rightarrow$	$\square$
13	$\triangleright \rightarrow$	P_4	$\rightarrow$	$\square$
14	$\triangleright \rightarrow$	P_4	$\rightarrow$	$\square$
15	$\triangleright \rightarrow$	P_4	$\rightarrow$	$\square$
16	$\triangleright \rightarrow$	P_1	$\rightarrow$	$\square$
17	$\triangleright \rightarrow$	P_4	$\rightarrow$	$\square$

Fig. 6. The perfect hash table data structure corresponding to database I .

mine the associated value for each symbolic object in the query. The hash function is applied to the associated values and spatial relationship mapping to form the hash address. The hash table is then consulted to determine the address of pictures, from the hash address, that contain the desired spatial relationship between the specified objects.

Since the hash table does not have any collisions, the intersection of the resulting sets of pictures can be retrieved and presented to the user in response to the query. For each triple, this retrieval can be performed in $O(1)$ steps. If the query contains m triples, the retrieval is performed in $O(m)$ steps.

3. NEAR-PERFECT HASH TABLE—UPDATE OPERATIONS

The perfect hash table is static in nature. For any update of the image database, the hash table must be recomputed from scratch. In order to bypass the computationally intensive reconstruction process for dynamic image databases, it is proposed that we allow for some potential collisions to occur in the new hash table. It is obvious that the amount of update—insertion and deletion—is relatively small compared to the size of the database (hash table). To keep the advantage of the perfect hash table, we propose that the collisions be limited to no more than 10% of the address space, or number of triples. The figure of 10% is inspired by the 10%-rule heuristic proposed by Cook and Oldehoeft.⁽¹⁹⁾ Their idea was to speed up the calculation of a static hash table at the cost of 10% additional space. We have extrapolated it further to the dynamic hash table that allows for insertion and deletion. Such a table is called the *near-perfect hash table*. In this section, we describe the additional information that is required to construct and maintain the near-perfect hash table data structure. Then, we present the query and update operations in the database.

The near-perfect hash table is defined to be one that allows no more than 10% of the triples to resolve collision in the entire hash table. In addition, it does

not allow for more than 10% of its addresses to be unused. If the number of collisions exceeds 10% of the size of the hash table, or if the number of free spaces exceed 10% of the hash table size, the perfect hash table is recomputed using the algorithm presented in reference (16), or by using some incremental technique that is not yet known.

The near-perfect hash table is not simply a hash table but also a dynamic dictionary storing additional information and data structures. Since it allows for collisions, we must keep the data required to identify the occurrence of a collision at every address, that is the triple itself, in addition to the data kept in the perfect hash table. In addition, we must keep a pointer to the next address for collision resolution. The modified data structure is presented in Fig. 7. In this figure, A_i indicates an address in the hash table, T_{A_i} indicates the triple at the address A_i , A'_i contains a pointer to the next address in case of a collision, and P_{A_i} form the linked list of pictures containing the triples T_{A_i} . The corresponding transformation of Fig. 6 is presented in Fig. 8.

The deletion operation is simple if we allow some unused spaces in the perfect hash table by allowing for the deleted addresses to be considered as placeholders and therefore, we will discuss that first. This will be followed by the insertion operation. We will also ensure that at no time, the size of the hash table with n distinct triples should exceed cn where $1 \leq c \leq 1.1$ as dictated by the 10%-rule.⁽¹⁹⁾

3.1. Deletion of images in the database

The operation of deleting images in an image database involves removing some pictures that may contain a given pattern. The pictures containing the pattern are pointed to from within the hash table by using a linked list. If the linked list attached to a hash address is completely deleted, we say that the hash address is not in use any more. This effectively means that the triple that pointed to the hash address does not exist any more because all the pictures containing that pattern have been removed from the database. This hash address is released to free the occupied slot, and the slot is marked as unused.

Deletion of a picture from the database will also affect the frequency of symbolic objects that has been retained with the hash table. This frequency is not modified until the hash table violates the 10%-rule and needs to be recomputed. This is necessary because at the time of retrieval, the intratriple ordering is performed on the

Ordered list of symbols (based on frequency)									
Associated value for each symbol									
Hash table									
A_1	T_{A_1}	A'_1	$\triangleright \rightarrow$	$P_{A_1,1}$	$\rightarrow$	$P_{A_1,2}$	$\rightarrow$	$\dots$	$\square$
A_2	T_{A_2}	A'_2	$\triangleright \rightarrow$	$P_{A_2,1}$	$\rightarrow$	$P_{A_2,2}$	$\rightarrow$	$\dots$	$\square$
$\vdots$									
A_n	T_{A_n}	A'_n	$\triangleright \rightarrow$	$P_{A_n,1}$	$\rightarrow$	$P_{A_n,2}$	$\rightarrow$	$\dots$	$\square$

Fig. 7. The near-perfect hash table data structure and associated information.

$\langle G, B, A, C, E, D, F \rangle$									
$\{(A, 1), (B, 9), (C, -1), (D, 5), (E, -1), (F, -7), (G, 2)\}$									
1	(A, C, 1)	-	$\triangleright \rightarrow$	P_4	$\rightarrow$	$\square$			
2	(G, F, 7)	-	$\triangleright \rightarrow$	P_2	$\rightarrow$	$\square$			
3	(G, F, 8)	-	$\triangleright \rightarrow$	P_2	$\rightarrow$	$\square$			
4	(C, E, 6)	-	$\triangleright \rightarrow$	P_4	$\rightarrow$	$\square$			
5	(E, D, 1)	-	$\triangleright \rightarrow$	P_4	$\rightarrow$	$\square$			
6	(A, C, 6)	-	$\triangleright \rightarrow$	P_1	$\rightarrow$	$\square$			
7	(A, E, 7)	-	$\triangleright \rightarrow$	P_4	$\rightarrow$	$\square$			
8	(G, G, 4)	-	$\triangleright \rightarrow$	P_2	$\rightarrow$	$\square$			
9	(B, F, 7)	-	$\triangleright \rightarrow$	P_3	$\rightarrow$	$\square$			
10	(B, C, 2)	-	$\triangleright \rightarrow$	P_4	$\rightarrow$	$\square$			
11	(C, D, 7)	-	$\triangleright \rightarrow$	P_4	$\rightarrow$	$\square$			
12	(B, A, 2)	-	$\triangleright \rightarrow$	P_1	$\rightarrow$	$\square$			
13	(B, A, 3)	-	$\triangleright \rightarrow$	P_4	$\rightarrow$	$\square$			
14	(A, D, 8)	-	$\triangleright \rightarrow$	P_4	$\rightarrow$	$\square$			
15	(B, D, 1)	-	$\triangleright \rightarrow$	P_4	$\rightarrow$	$\square$			
16	(B, C, 8)	-	$\triangleright \rightarrow$	P_1	$\rightarrow$	$\square$			
17	(B, E, 9)	-	$\triangleright \rightarrow$	P_4	$\rightarrow$	$\square$			

Fig. 8. The near-perfect hash table data structure corresponding to database I .

triples in the query based on the frequency that existed at the time of hash table computation. If an object is completely removed, that is, all the pictures containing that object have been deleted from the database, its frequency information and the associated value are also eliminated. However, the hash table itself remains unchanged.

In addition to the information on frequency-based ordering of symbols and the associated value of each symbolic object, the following information is necessary to facilitate the deletion operation:

1. The triple corresponding to an address in the hash table. In the existing hash tables (see Fig. 6), the triple is not kept with the hash table. Instead, only the pictures that contain the triple are kept in a linked list starting at the corresponding hash address. This is required to test the triple in satisfying a query.
2. A linked list of all triples corresponding to each picture object. This can establish if a symbolic object is still in use after a picture has been deleted. Moreover, it can facilitate the computation of frequency to be used with an incremental algorithm to compute the hash table that is being investigated.

This tracking information requires the use of additional data structures to be stored with the hash table. Most of these data structures are best implemented by linked list abstract data type, as their associated operations of deletion, insertion, and search are already well known.

In our example, we remove the picture P_3 which results in the removal of the triple and the picture at address 9 in Fig. 8, making the address available for future use. The table need not be recomputed.

3.2. Inserting new images in the database

The issue of construction of a near-perfect hash table arises when we have to insert a new picture in the image database. Whenever we receive a new picture, we need to identify all the picture objects contained in it and create

all the triples to show the spatial relationships between those objects. In order to construct a near-perfect hash table, we have to consider three possibilities for each triple:

1. None of the picture objects exists in the database, i.e. both the picture objects in the triple are new. Both of these objects will need to be assigned associated values.
2. Exactly one of the picture objects already exists in the database. This picture object already has an associated value. We need to assign an associated value to the other picture object.
3. Both the picture objects already exist in the database. This will lead to further subcases:

- (a) The spatial relationship between the picture objects is such that the triple already exists. It is easy to handle this by simply adding the picture to the existing linked list starting at the slot in the hash table corresponding to the triple.
- (b) The spatial relationship is new. This would lead to collisions as the three elements used to compute the hash address ($a(S_i)$, $a(S_j)$, and r_{ij}) have already been defined. This is the most complex case in database updation and makes it necessary to recompute the hash table, or allow for a few collisions, leading to near-perfect hash table.

If a new associated value needs to be assigned to any object in the triple, the problem may be more simply resolved by manipulating the associated value. It is easy to see from the above that the collisions pose the most serious problem when we have a triple with two objects that already have an associated value and a new relationship between them which is different from the one that exists in the database. In this case, the only recourse left

is to allow for collisions in the hash table, or to recalculate the hash table.

We will illustrate case 1 by inserting the picture presented in Fig. 1 into the database I as it exists now after removal of picture P_3 . The new hash table, with P_3 removed and Fig. 1 (referred to by P_n) added, is shown in Fig. 9.

3.3. Near-perfect hash table

For both insertion and deletion, the hash table is allowed to be *near-perfect*, where *near-perfect* is defined to be a hash table that satisfies the following conditions:

1. For insertion, the near-perfect hash table will allow a number of collisions that does not exceed 10% of its address space at any time.
2. For deletion, the near-perfect hash table will not allow for more than 10% of its address space to be unused at any time.

Whenever either of these conditions is violated, the hash table will not be updated, but reconstructed. We are also working on the computation of the hash table using an incremental technique that may take into account a part of the existing hash table to reduce the calculations.

The construction of near-perfect hash table is accomplished by extending the size of the perfect hash table by at most 10%. That is, first a hash table is already calculated by the approach presented in reference (16). In order to update the database by insertion limited to 10% of the original space, this table is extended to near-perfect hash table. The analysis of query/search is presented in Section 4 with the assumption that the near-perfect hash table is already constructed.

$\langle G, B, A, C, E, D, F, canoe, house, lake \rangle$									
$\{(A, 1), (B, 9), (C, -1), (D, 5), (E, -1), (F, -7), (G, 2), (canoe, 4), (house, 10), (lake, 3)\}$									
1	(A, C, 1)	-	$\triangleright \rightarrow$	P_4	$\rightarrow$	$\square$			
2	(G, F, 7)	-	$\triangleright \rightarrow$	P_2	$\rightarrow$	$\square$			
3	(G, F, 8)	-	$\triangleright \rightarrow$	P_2	$\rightarrow$	$\square$			
4	(C, E, 6)	-	$\triangleright \rightarrow$	P_4	$\rightarrow$	$\square$			
5	(E, D, 1)	-	$\triangleright \rightarrow$	P_4	$\rightarrow$	$\square$			
6	(A, C, 6)	-	$\triangleright \rightarrow$	P_1	$\rightarrow$	$\square$			
7	(A, E, 7)	-	$\triangleright \rightarrow$	P_4	$\rightarrow$	$\square$			
8	(G, G, 4)	-	$\triangleright \rightarrow$	P_2	$\rightarrow$	$\square$			
9	(canoe, lake, 2)	-	$\triangleright \rightarrow$	P_n	$\rightarrow$	$\square$			
10	(B, C, 2)	-	$\triangleright \rightarrow$	P_4	$\rightarrow$	$\square$			
11	(C, D, 7)	-	$\triangleright \rightarrow$	P_4	$\rightarrow$	$\square$			
12	(B, A, 2)	-	$\triangleright \rightarrow$	P_1	$\rightarrow$	$\square$			
13	(B, A, 3)	-	$\triangleright \rightarrow$	P_4	$\rightarrow$	$\square$			
14	(A, D, 8)	-	$\triangleright \rightarrow$	P_4	$\rightarrow$	$\square$			
15	(B, D, 1)	-	$\triangleright \rightarrow$	P_4	$\rightarrow$	$\square$			
16	(B, C, 8)	-	$\triangleright \rightarrow$	P_1	$\rightarrow$	$\square$			
17	(B, E, 9)	-	$\triangleright \rightarrow$	P_4	$\rightarrow$	$\square$			
18	(lake, house, 5)	19	$\triangleright \rightarrow$	P_n	$\rightarrow$	$\square$			
19	(canoe, house, 4)	-	$\triangleright \rightarrow$	P_n	$\rightarrow$	$\square$			

Fig. 9. The near-perfect hash table data structure corresponding to database I after the deletion and insertion operations. At address 18, there is a collision which is resolved at address 19.

4. ANALYSIS OF THE SEARCH ALGORITHM

In this section, we present the analysis of the search algorithm for the near-perfect hash table and demonstrate that the search algorithm performs almost as well as the earlier algorithm based on the perfect hash table.

Let T_h , T_t , and T_l be the computation times used for the

colliding triples at any address. The search terminates in one step at each of the $n - c$ non-colliding triples, in one step at half of the c colliding triples when the searched-for triple is in the first place, and in two steps at the remaining half of the c colliding triples. The computational complexity can be given by

$T_h + T_t + T_l$	For each of $n - c$ non-colliding triples,
$T_h + T_t$	For each of the first occurrence of colliding triples,
$T_h + T_t + T_l + T_t$	For each of the remaining colliding triples.

The average time complexity in this case is

$$\begin{aligned}
 T_n &= \frac{1}{n} ((T_h + T_t + T_l)(n - c) + (T_h + T_t) \frac{c}{2} + (T_h + T_t + T_l + T_t) \frac{c}{2}) \\
 &= \frac{1}{n} ((T_h + T_t + T_l)(n - c) + (T_h + T_t + T_l) \frac{c}{2} - T_l \frac{c}{2} + (T_h + T_t + T_l) \frac{c}{2} + T_t \frac{c}{2}) \\
 &= \frac{1}{n} ((T_h + T_t + T_l)(n - c) + (T_h + T_t + T_l)c + (T_t - T_l) \frac{c}{2}) \\
 &= \frac{1}{n} ((T_h + T_t + T_l)n + (T_t - T_l) \frac{c}{2}) \\
 &= (T_h + T_t + T_l) + (T_t - T_l) \frac{c}{2n} \\
 &\leq (T_h + T_t + T_l) + (T_t - T_l)(0.05) \quad c \leq 0.1n \\
 &= O(1).
 \end{aligned}$$

calculation of hash address, testing for the presence of triple by matching it with the triple in the table, and testing for the non-empty linked list.

Let the hash table consist of n addresses. Let the collisions occur on c of the n addresses, ($0 \leq c \leq 0.1n$). Then, there are $n - c$ addresses on which there is no collision. The collision resolution space can be enumerated by the following criterion:

$T_h + T_t + T_l$	For each of $n - c$ non-colliding triples,
$T_h + T_t$	For the first triple on the colliding triples list,
$T_h + T_t + T_l + T_l$	For the second triple on the colliding triples list,
$T_h + T_t + 2(T_l + T_t)$	For the third triple on the colliding triples list,
$\vdots$	$\vdots$
$T_h + T_t + (c - 1)(T_l + T_t)$	For the c th triple on the colliding triples list.

4.2. Worst case analysis

The worst case occurs when all the colliding triples map to the same address in the hash table. Thus, we have $n - c$ triples for which the search is terminated in one step, and one triple each for which the search is terminated in $1, 2, \dots, c$ steps. The computational complexity is given by

1. Each collision occurs at a different address. The search is performed by resolving the collision in at most two steps.
2. All collisions occur at the same address. This occurs when all the c triples involved in collisions are mapped to the same address by the hash function. In such a case, the search is performed in at most c steps.
3. The collisions are randomly distributed but some addresses may have more than two triples mapped onto them. The search in this case may be performed in more than two steps (but less than c steps).

The above three possibilities lead to three cases for search with collisions—best case, worst case, and the average case. The analysis for each of these cases is performed in the following subsections.

4.1. Best case analysis

In the best case, the collisions are evenly distributed in the hash table. That is, there are no more than two

The average time complexity in this case is

$$\begin{aligned}
 T_n &= \frac{1}{n} ((T_h + T_t + T_l)(n - c) + (T_h + T_t)c \\
 &\quad + T_l + (T_l + T_t)(1 + 2 + \dots + (c - 1))) \\
 &= \frac{1}{n} \left((T_h + T_t + T_l)(n - c) + (T_h + T_t)c + T_l \right. \\
 &\quad \left. + (T_l + T_t) \frac{c(c - 1)}{2} \right) \\
 &= \frac{1}{n} \left((T_h + T_t + T_l)n - T_l c + T_l + (T_l + T_t) \frac{c(c - 1)}{2} \right) \\
 &= \frac{1}{n} \left((T_h + T_t + T_l)n - (c - 1)T_l + (T_l + T_t) \frac{c(c - 1)}{2} \right) \\
 &= \frac{1}{n} \left((T_h + T_t + T_l)n - T_l(c - 1) + T_l \frac{c(c - 1)}{2} + T_t \frac{c(c - 1)}{2} \right) \\
 &= \frac{1}{n} \left((T_h + T_t + T_l)n + T_l \frac{(c - 2)(c - 1)}{2} + T_t \frac{c(c - 1)}{2} \right) \\
 &\leq \frac{1}{n} \left((T_h + T_t + T_l)n + T_l \frac{c^2}{2} + T_t \frac{c^2}{2} \right)
 \end{aligned}$$

$$\begin{aligned}
&= (T_h + T_t + T_l) + \frac{0.01n^2}{2n}T_l + \frac{0.01n^2}{2n}T_t \\
&= (T_h + T_t + T_l) + 0.005nT_l + 0.005nT_t \\
&= (T_h + T_t + T_l) + 0.005(T_l + T_t)n \\
&= C_1 + 0.005C_2n \text{ where } C_1 \text{ and } C_2 \text{ represent constants} \\
&= O(n).
\end{aligned}$$

4.3. Average case analysis

In the average case, the collisions are randomly distributed across the hash table. Thus, the average case falls in between the best case and the worst case. With m slots in the hash table (the number of triples), there are $k_1, k_2, \dots, k_p$ collisions at different hash addresses such that $\sum_{i=1}^p k_i = c$ and $1 \leq p \leq c$. That is, at hash address a_i , the collisions are resolved with at most k_i steps, $2 \leq k_i < c$. Therefore, one of the n triples can fall in the following two categories:

1. $(n - c)$ non-colliding triples;
2. k_i colliding triples in the same space where $i = 1, 2, \dots, p$.

The corresponding times for computational complexity are given by

For each of $n - c$ triples	$(T_h + T_t + T_l),$
For each k_i , the k_i collisions	
can be resolved in time	$(T_h + T_t + T_l)k_i + (k_i - 1)T_t + [(k_i - 1)(k_i - 2)/2](T_t + T_l).$

The average time complexity in this case is the sum of all these terms divided by the total number of triples n and is computed as follows:

$$\begin{aligned}
T_n &= \frac{1}{n} \left((T_h + T_t + T_l)(n - c) \right. \\
&\quad \left. + \sum_{i=1}^m \left((T_h + T_t)k_i + T_l + (T_t + T_l) \frac{k_i(k_i - 1)}{2} \right) \right) \\
&= \frac{1}{n} \left((T_h + T_t + T_l)(n - c) + (T_h + T_t) \sum_{i=1}^m k_i \right. \\
&\quad \left. + mT_l + \frac{1}{2}((T_t + T_l) \sum_{i=1}^m (k_i(k_i - 1))) \right).
\end{aligned}$$

Since $m < c$ and $\sum_{i=1}^m k_i = c$,

$$\begin{aligned}
T_n &\leq \frac{1}{n} \left((T_h + T_t + T_l)(n - c) + (T_h + T_t)c \right. \\
&\quad \left. + cT_l + \frac{1}{2}(T_t + T_l) \left(\sum_{i=1}^m k_i^2 - \sum_{i=1}^m k_i \right) \right) \\
&= \frac{1}{n} \left((T_h + T_t + T_l)(n - c) + (T_h + T_t + T_l)c \right. \\
&\quad \left. + \frac{1}{2}(T_t + T_l) \sum_{i=1}^m k_i^2 - \frac{1}{2}(T_t + T_l) \sum_{i=1}^m k_i \right) \\
&= \frac{1}{n} \left((T_h + T_t + T_l)n + \frac{1}{2}(T_t + T_l) \sum_{i=1}^m k_i^2 \right. \\
&\quad \left. - \frac{1}{2}(T_t + T_l) \sum_{i=1}^m k_i \right).
\end{aligned}$$

By substituting $\sum_{i=1}^m k_i = c$ and

$$\sum_{i=1}^m k_i^2 \leq \left(\sum_{i=1}^m k_i \right)^2 = c^2,$$

$$\begin{aligned}
T_n &\leq \frac{1}{n} \left((T_h + T_t + T_l)n + \frac{1}{2}(T_t + T_l)c^2 \right. \\
&\quad \left. - \frac{1}{2}(T_t + T_l)c \right).
\end{aligned}$$

Since $c \leq 0.1n$,

$$\begin{aligned}
T_n &\leq \frac{1}{n} \left((T_h + T_t + T_l)n + \frac{1}{2}(T_t + T_l)(0.1n)^2 \right. \\
&\quad \left. - \frac{1}{2}(T_t + T_l)(0.1n) \right) \\
&\leq (T_h + T_t + T_l) + 0.005(T_t + T_l)n - 0.05(T_t + T_l) \\
&\leq (T_h + T_t + T_l) - 0.05(T_t + T_l) + 0.005(T_t + T_l)n \\
&\leq C'_1 + C'_2n \text{ where } C'_1 \text{ and } C'_2 \text{ are constants} \\
&= O(n).
\end{aligned}$$

Since the triples have already been sorted in the algorithm to determine associated values, and are stored as such as a part of the hash table, their collating sequence can be used to organize the colliding triples in order. Binary search can be used to look for the presence of triples at a slot giving the time complexity T_n to be $O(\log n)$. However, the collisions are limited in number (not more than 10% of the size of the hash table) and hence, the implementation does not gain much by using binary tree instead of a linked list.

5. CONCLUSION

The perfect hash table has already proved its utility in efficient storage and fast retrieval of pictures containing specified symbolic objects from image database systems. The best known algorithm for the construction of the perfect hash table itself is of exponential time complexity as proved in reference (16). Updating the static image databases requires the reconstruction of the entire hash table from the beginning which limits the application of such table. We have attempted to relax the perfect hash table constraint to near-perfect hash table that allows us to dynamically update the database without losing the computations spent in the construction of the existing hash table. We have analyzed the complexity of retrieval with such a hash table showing promising results. The near-perfect hash table update is computationally feasible as the complexity of retrieval is of linear order in terms of the size of the table such that the constant of linearity is very small.

REFERENCES

1. V. Gudivada, V. Raghavan and K. Vanapipat, A unified approach to data modeling and retrieval for a class of image database applications, in *Multimedia Database Systems: Issues and Research Directions*, S. Jajodia and V. Subrahmanian, eds. Springer, Berlin (1995).
2. T. Arndt, A survey of recent research in image database management, *Proc. IEEE Workshop on Visual Languages*, pp. 92–97 (October 1990).
3. T. Tamura and N. Yokoya, Image database systems: A survey, *Pattern Recognition* **17**, 29–43 (1984).
4. S. K. Chang, C. W. Yan, D. C. Dimitroff and T. Arndt, An intelligent image database system, *IEEE Trans. Software Eng.* **14**(5), 681–688 (May 1988).
5. S.-Y. Lee and F.-J. Hsu, 2D C-string: A new spatial knowledge representation for image database systems, *Pattern Recognition* **23**(10), 1077–1087 (1990).
6. S. K. Chang and S. H. Liu, Picture indexing and abstraction techniques for pictorial databases, *IEEE Trans. Pattern Analysis Mach. Intell.* **PAMI-6**, 475–484 (July 1984).
7. S. K. Chang, Q. Y. Shi and C. W. Yan, Iconic indexing by 2D strings, *IEEE Trans. Pattern Analysis Mach. Intell.* **PAMI-9**(3), 413–428 (1987).
8. S.-K. Chang, *Principles of Pictorial Information Systems Design*. Prentice-Hall, Englewood Cliffs, New Jersey (1989).
9. S.-K. Chang and E. Jungert, A spatial knowledge structure for image information systems using symbolic projections, in *Proc. 1986 Fall Joint Computer Conf.*, H.S. Stone and S. Winkler, eds, pp. 79–86. Dallas, Texas (November 1986).
10. S.-K. Chang, E. Jungert and Y. Li, The design of pictorial databases based upon the theory of symbolic projections, in *Design and Implementation of Large Spatial Databases: Proc. First Symp. SSD '89*, A. Buchman, O. Günther, T. R. Smith and Y. F. Wang, eds, pp. 303–323, Santa Barbara, California (July 1989). (*Lecture Notes in Computer Science*, Vol. 409) (1990).
11. H. Samet, The quadtree and related hierarchical data structures, *ACM Comput. Surveys* **16**(2), 187–260 (June 1984).
12. C.-C. Chang and S.-Y. Lee, Retrieval of similar pictures on pictorial databases, *Pattern Recognition* **24**(7), 675–680 (1991).
13. S. K. Bhatia and C. L. Sabharwal, A fast implementation of a perfect hash function for picture objects, *Pattern Recognition* **27**(3), 365–376 (1994).
14. Z. J. Czech and B. S. Majewski, A linear time algorithm for finding minimal perfect hash functions, *Comput. J.* **36**(6), 579–587 (1993).
15. M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Heide, H. Rohnert and R. E. Tarjan, Dynamic perfect hashing: Upper and lower bounds, *SIAM J. Comput.* **23**(4), 738–761 (August 1994).
16. C. L. Sabharwal and S. K. Bhatia, Perfect hash table algorithm for image databases using negative associated values, *Pattern Recognition* **28**(7), 1091–1101 (July 1995).
17. R. Sprugnoli, Perfect hashing functions: A single probe retrieving method for static sets, *Comm. ACM* **20**(11), 841–850 (November 1977).
18. R. Cichelli, Minimal perfect hashing function made simple, *Comm. ACM* **23**(1), 17–19 (January 1980).
19. C. R. Cook and R. Oldehoeft, A letter-oriented minimal perfect hashing function, *ACM SIGPLAN Notices* **17**, 18–27 (1982).

About the Author—CHAMAN SABHARWAL received his B.A. (Honors) and M.S. degrees in Mathematics from the Punjab University, Chandigarh, India. He earned the M.S. and Ph.D. in Applied Mathematics from the University of Illinois, Urbana-Champaign. He is Professor of Computer Science at the University of Missouri—Rolla. He has been Professor of Mathematics and Director of Undergraduate Computer Science at St. Louis University. He has also been Lead Engineer and Senior Specialist at McDonnell Douglas Corporation. He has published in the areas of Software Engineering, Feature Recognition, Robotics, and Geometric Modeling. His research interests include the areas of Computer Graphics, Computer Vision, Geometric Modeling, Machine Learning, Parallel Processing, Pattern Recognition, and Robotics. He has published in various journals including SIAM Journal on Applied Mathematics, Computer Aided Geometric Design, and Pattern Recognition. He is a member of ACM and IEEE.

About the Author—SANJIV BHATIA received his B.E. degree in Computer Science Engineering from Motilal Nehru Regional Engineering College, Allahabad, India, in 1983. He then worked with Engineers India Limited, New Delhi, India, for three years. He received his M.S. (Computer Science) from the University of Arkansas, Fayetteville, in 1987, and Ph.D. (Computer Science) from the University of Nebraska, Lincoln, in 1991. Since 1991, he has been with the University of Missouri—St. Louis. He has published in the areas of image databases, face recognition, knowledge acquisition, machine learning, and information retrieval. He is a member of ACM and AAAI.