A Fast Implementation of A Perfect Hash Function for Picture Objects

Abstract

In Image Database Systems, symbolic pictures are represented by 2D strings that are converted into triples. Each triple is mapped to a unique hash address for timely retrieval of pictures, reducing the pattern-matching problem corresponding to a query to that of computation of hash function. The values associated with the picture objects are used to compute hash addresses for triples developed from the query. In this paper, we propose heuristics to speed up the computation of the associated values for the picture objects. Experimental results show that the new algorithm achieves almost a 90% gain, in search space, over existing algorithm to compute the associated values.

Keywords: image database systems, 2D string, perfect hashing function, A* search algorithm, associated value function
1 Introduction

A database management system is employed to organize, store, and retrieve large amount of data. Traditionally, databases have been used for management of alphanumeric data. Thus, most of the existing database techniques for search and retrieval have concentrated on alphanumeric pattern matching. In contrast, an image or pictorial database has to concern itself with the management of a large number of images. The traditional techniques for query and search cannot be applied to pictorial database. Hence, a different set of techniques is needed to completely or partially match the queries with the pictures in a database for retrieval.

Image databases are required in many applications such as office automation, CAD systems, and the fifth generation computing systems. Some of these applications, such as CAD systems, require that the database techniques include operations for manipulation of databases to account for parts of an image [1]. Sometimes, a user may want to retrieve images by specifying relations between parts of an image. A scene in an image is described in terms of atomic objects, known as picture elements. The requirement to allow for operations on picture elements, specially the retrieval operations that specify adjacency and relative positioning, puts new strains on computation during query and retrieval [2]. For timely access to images, the retrieval process needs a well-developed index. The index provides an efficient storage and access mechanism, and is essential for any viable image database. Therefore, the enhancement in storage and access methods is warranted by the large volume of complex image data as the processing time to retrieve images in real-time can be enormous. In this paper, we will primarily focus on the operations for indexing and
picture retrieval in an image database.

The earliest image database systems employed *lookup tables* to organize the database and to access images for retrieval. The retrieval of images from earlier image databases involved the process of pictorial pattern matching. This is achieved by matching pictorial objects and their spatial relationship with the objects in the image database. The problem of matching individual picture elements in two given pictures to determine their similarity is identical to graph matching problem [3], and can be extremely time consuming for any moderately complex picture. The application of this technique has limited use in the image databases. Therefore, the earlier image database systems are not suitable for spatial reasoning on images because of inefficient organization and limited choice of data structures [4].

The capability to represent pictorial objects and the spatial relationship between these objects is a major database design issue [5, 6]. The database systems based on the *Packed R-tree* structure [7] and the *Intelligent Image Database System* (IIDS) [4] took the first steps in providing this additional capability. In particular, spatial reasoning and similar image information retrieval are two of the strengths of IIDS. Since spatial relationship depends on human interpretation and is a fuzzy concept, the IIDS is concerned with the high level object oriented approach to picture retrieval rather than low level image primitive of objects at exact spatial locations [8].

Chang, et. al., proposed a data structure, known as a *2D string*, for spatial reasoning reasoning on images and quick retrieval of images based on partial match [9, 10, 11, 12, 13]. The 2D strings are well suited to images where orthogonal symbolic projections are disjoint, and can be described by using only global operators.
However, the 2D strings are not sufficient enough to represent complex images. A new set of local operators is introduced to compensate for images whose projections on orthogonal axes are overlapping. The concept of 2D string is generalized to 2D G-string [14]. However, the concept of 2D G-string is not well suited for storage efficiency and spatial reasoning. Hence, a new concept of 2D C-string was introduced for efficiency in storage and retrieval [8]. The picture algebra of the local and global operators is developed in [15]. Another type of 2D H-String is introduced which combines the advantages of C-string and quadtree data structures to accommodate more complex picture elements.

The problem of picture information storage and retrieval becomes the problem of 2D string manipulation. In this paper, we concentrate on speeding up the algorithm to compute associated values for picture elements that are used to calculate a perfect hash function for spatially-related picture elements derived from the 2D strings. The 2D C-string, G-string, and H-string algorithms will be handled separately.

The perfect hash function approach was suggested for storage and access of information in [16] and was simplified in [17]. This method was improved to calculate the associated values for the perfect hash function in [18]. However, this latter technique is still inefficient for complex pictures. We have developed some heuristics to prune the search space to determine the associated values for picture elements to compute the hash addresses. The new heuristic algorithm results in an efficient computation of a perfect hash function for storing and retrieving pictures.

The paper is organized as follows. In Section 2, we present the 2D string approach to describe the spatial relationship between picture elements. In Section 3, we describe
the earlier algorithm [19] to compute associated values for picture elements to assign hash addresses to ordered triples [19, 18]. In Section 4, we advance the heuristics to prune the search and present the enhanced algorithm to compute the associated values for picture elements. The experimental results showing the improvement of our algorithm over the earlier algorithm are presented in Section 5.

2 The 2D String Approach

In the 2D string approach, each picture is considered to be made up of a set of picture elements. The entire picture is divided into axis-oriented zones that are known as *minimum bounding rectangles* (MBRs). Each MBR encloses at least one picture element. The centroid of the rectangle is used as reference to the picture element. Assuming that no overlaps occur (overlaps considered separately), the individual picture elements are projected onto the orthogonal axes. The 2D spatial relations among picture elements are expressed in terms of 1D spatial relations (or 1D strings) among the projections on the orthogonal axes. The 2D string is a data structure that develops and maintains the spatial relationship among elements in a picture using the MBRs and allows for spatial reasoning on those elements [19, 13, 8]. It is assumed that the picture elements have been recognized by using standard techniques in image processing.

The two dimensional relations between any two picture elements can be classified into nine categories. For example, starting with one of the picture elements to be a *reference*, the nine categories can be enumerated as “to the north of reference,” “to the north-west of reference,” “to the west of reference,” “to the south-west of
reference,” “to the south of reference,” “to the south-east of reference,” “to the east of reference,” “to the north-east of reference,” and “at the same location as reference.” For brevity, the verbal description is mapped onto integers 1, 2, 3, 4, 5, 6, 7, 8, and 9, as shown in the Figure 1.

![Integer mapping of spatial relations](image)

Figure 1: Integer mapping of spatial relations

The basic idea in a 2D string is to project the picture elements on the horizontal and vertical axis. The axis-oriented symbolic projections are used to determine the spatial relationship among elements. 2D strings employ three operators to describe the spatial relationships among the picture elements. For two symbolic picture elements $S_i$ and $S_j$, the three operators are described in Table 1.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_i &lt; S_j$</td>
<td>$S_i$ is to the left of $S_j$ (horizontal axis), or $S_i$ is above $S_j$ (vertical axis)</td>
</tr>
<tr>
<td>$S_i = S_j$</td>
<td>Projection of $S_i$ coincides with the projection of $S_j$</td>
</tr>
<tr>
<td>$S_i : S_j$</td>
<td>$S_i$ is in the same MBR as $S_j$</td>
</tr>
</tbody>
</table>

Table 1: Relational operators used in 2D strings

The operators in Table 1 are used to describe the relationship between picture elements in two strings. The first string describes the relationship between the elements as seen in the projection on the horizontal axis while the second string describes the relationship in the projection on vertical axis.
Let $S = \{S_1, S_2, \ldots, S_n\}$ be a set of $n$ picture elements in an image and $R = \{:, =, <\}$ be the set of global, infix relational operators as described in Table 1. Then, a 1D string is a string of symbols from the set $S \cup R$ such that each element of $R$ is enclosed between two elements of $S$.

Let $\text{str}(S \cup R)$ be the set of all 1D strings consisting of elements from the set $S \cup R$ satisfying the above constraint. A 2D image is a subset of $\text{str}(S \cup R) \times \text{str}(S \cup R)$. A 2D string is an ordered pair

$$(S_1 r_x S_2 r_x S_3 \cdots S_{n-1} r_x S_n, S_{p(1)} r_y S_{p(2)} r_y S_{p(3)} \cdots S_{p(n-1)} r_y S_{p(n)})$$

where

1. $S_i \in S$,
2. $r_x^i, r_y^i \in R$, and
3. $(p(1), p(2), \ldots, p(n))$ is a permutation of $(1, 2, \ldots, n)$.

The spatial relations among picture elements along $x$-axis are embedded in the first string while those along the $y$-axis are embedded in the second string.

The development of a 2D string is explained with the help of a symbolic picture presented in Figure 2.

<table>
<thead>
<tr>
<th></th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>C</td>
</tr>
<tr>
<td>A</td>
<td>B F</td>
</tr>
</tbody>
</table>

Figure 2: Picture with five symbolic picture elements
The picture $f$ in Figure 2 is described by a 2D string as

$$(A = E < B : F < C = D, A = B : F < E = C < D)$$

A 2D string is used to construct triples that contain the spatial relationship between picture elements as described by the integer mapping of MBRs in Figure 1. The triples are made up of two picture elements in symbolic form and an integer to show the spatial relationship between the two elements. Thus, the relationship $r_{ij}$ of a symbol $S_i$ to a reference symbol $S_j$ is denoted by a *triple* $(S_i, S_j, r_{ij})$. As an example, the triple to describe the relation between $A$ and $B$ in Figure 2 is specified as $(A, B, 3)$ which implies that “$A$ is to the west of $B$.”

It should be noted that a triple $(S_i, S_j, r_{ij})$ is equivalent to $(S_j, S_i, r_{ji})$ such that $|r_{ij} - r_{ji}| = 4$. In Figure 2, $(A, B, 3)$ and $(B, A, 7)$ represent the same triple, implying that “$A$ is to the west of $B$” is the same as “$B$ is to the east of $A$.”

Let a picture be partitioned into $N$ intervals along the $x$-axis and $M$ intervals along the $y$-axis. Mathematically, a picture $f$ is a mapping $f : N \times M \rightarrow 2^S$, where $S$ is the set of all picture elements in the image. The image in Figure 2 is described by

\[
\begin{align*}
  f(1,1) & = \{A\}, \\
  f(2,1) & = \{B,F\}, \\
  f(1,2) & = \{E\}, \\
  f(3,2) & = \{C\}, \\
  f(3,3) & = \{D\}.
\end{align*}
\]
The set $S$ of symbolic picture elements in Figure 2 is given by

$$S = \{A, B, C, D, E, F\}$$

Each picture in a database can be described in terms of triples. The picture elements are used to construct triples that show the relationship between every pair of elements. The set of triples in Figure 2 is given by

$$T = \{(A, B, 3), (A, C, 4), (A, D, 4), (A, E, 5), (A, F, 3), (B, C, 4),
(B, D, 4), (B, E, 6), (B, F, 9), (C, D, 5), (C, E, 7), (C, F, 8),
(D, E, 8), (D, F, 8), (E, F, 2)\}$$

If there are several pictures in a scene, say $f_k$, for $k = 1, 2, \ldots, n$ and $T_k$ is the set of triples corresponding to $f_k$, then the set of all triples is denoted by $T = \bigcup_{k=1}^{n} T_k$. The set of symbols in all the pictures is given by $S = \{S_i : (S_i, *, *) \in T$ or $(*, S_i, *) \in T\}$.

As seen in Figure 3, queries for similar pictures in the picture of Figure 2 can be formulated in terms of 2D strings and triples as

$$q_1 = (A < D, A < D) \quad q_1 = \{(A, D, 4)\}$$
$$q_2 = (E < A, A < E) \quad q_2 = \{(A, E, 6)\}$$
$$q_3 = (A < C = D, A = C < D) \quad q_3 = \{(A, C, 3), (A, D, 4), (C, D, 5)\}$$

The pattern matching is an intricate problem. A perfect hash function can be usefully employed to store and answer these queries. Taking cue from hash functions
in programming languages [17, 18, 16], Chang and Lee advanced the idea of using a perfect hash function to resolve the queries in 2D strings using a triple representation [19]. The picture elements in an entire image database are hashed such that there is a unique address corresponding to each triple. The address leads to a pointer to a linked list of all pictures that contain the triple. This reduces the query to a simple address calculation for the specified triples, and an intersection of retrieved pictures if necessary.

3 Previous Algorithm for Perfect Hash Function

In this section, we describe the computation of a perfect hash function using the picture elements in all the images of a database. The basic idea is to compute a numerical value corresponding to each picture element such that the storage of database and retrieval in response to queries can be quickly satisfied by computing the addresses of triples of picture elements in each image. The algorithm for database indexing described in this section is based on a hashing algorithm presented in [19, 18]. Different steps in the algorithm are presented as procedures and a pseudocode algorithm is described.

Cook and Oldehoeft’s algorithm is based on a slight modification of a perfect hash function defined in [17]. The hash function is given by

\[ h : T \rightarrow \{1, 2, \ldots, k\} \ni h(S_i, S_j, r_{ij}) = r_{ij} + a(S_i) + a(S_j) \]

where \( a(\cdot) \) is an auxiliary function (called the associated value function) that assigns a value to each symbol. The calculation of hash function amounts to determining the associated value function. But the calculation of this hash function [17] is not
efficient because it calculates hash values one triple at a time. Cook and Oldehoeft improved on this algorithm by defining heuristics that regulate search by calculating hash values for sets of triples related by the middle symbol, or second picture element [18]. There are two aspects of this algorithm: (1) assigning a value to a symbol leads to assigning hash values to a set of triples in the picture, and (2) the mapping keeps the storage space size as small as possible. The algorithm is briefly described below.

The first step in the algorithm involves computation of frequency of each picture element in the set of triples. Let $T$ be the set of triples in all the images in a database, and $\{S_i : 1 \leq i \leq n\}$ be the set of all picture elements. If an element occurs twice in a triple (such as $(S_i, S_i, r_{ii})$), it is assigned a higher frequency. The picture elements are sorted such that an element with higher frequency precedes an element with lower frequency. Note that the frequencies are used for only ordering triples and sorting them; they are not used in any other calculations.

```
procedure compute_frequency;
    S ← \{S_1, S_2, \ldots, S_n\};
    T ← \{(S_i, S_j, r_{ij}) : S_i, S_j ∈ S\};
    for all $S_i ∈ S \ni (S_i, S_i, *) \notin T$
        $\phi_i = \text{frequency}(S_i)$ in $T$;
        $\max_\phi \leftarrow \max_{1 \leq i \leq n}(\phi_i)$;
    for all $S_i ∈ S \ni (S_i, S_i, r_{ii}) ∈ T$
        $\phi_i = \max_\phi + \text{frequency}(S_i)$ in $T$;
    sort $S$ in descending order of frequency $\phi_i$ of $S_i$
```

The frequencies computed above are used for ordering the picture elements in triples. For each triple in $T$, we reverse the order of the two picture elements, if necessary, such that the first element has a higher frequency than the second element. In such a case, the relationship $r_{ij}$ is changed to $r_{ji}$. As a result of this ordering, the
symbols in a triple \((S_i, S_j, r_{ij})\) ensure that \(\text{frequency}(S_i) > \text{frequency}(S_j)\). Ties are broken arbitrarily. Note that \((S_i, S_j, r_{ij})\) is equivalent to \((S_j, S_i, r_{ji})\) with \(|r_{ij} - r_{ji}| = 4\), where \(r_{ij}, r_{ji} \in \{1, 2, \ldots, 9\}\).

```
procedure order_triple_elements;
   for each triple \((S_i, S_j, r_{ij}) \in T\) do
      if \(\phi_i < \phi_j\) then
         change triple \((S_i, S_j, r_{ij})\) to \((S_j, S_i, r_{ji})\)
```

The ordered triples are sorted in the descending order of frequencies of the second element in each triple as the key. Because of the higher frequency assigned to elements that appear twice in some triple, the triples of the form \((S_i, S_i, r_{ii})\) appear first after the ordering. That is,

\[(S_i, S_i, r_{ii}) \prec (S_j, S_j, r_{jj}) \prec (S_j', S_j', r_{jj'}) \prec (S_j'', S_j'', r_{jj''})\]

provided \(\text{frequency}(S_j') > \text{frequency}(S_j'')\). Ties are broken arbitrarily.

```
procedure sort_triples;
   sort triples with the second element as key in descending order
```

Once the triples have been sorted, we compute the integer valued mapping \(a(\cdot)\) associated with each picture element so that each triple \((S_i, S_j, r_{ij})\) is assigned a unique address given by \(a(S_i) + a(S_j) + r_{ij}\) in the space of available addresses. The algorithm for the assignment function is further based on depth-first search with backtracking.

The procedure **assign_values** is presented below. In this procedure, a value is assigned to the second symbol for each set of triples that have the same second symbol. The initial value for the symbol begins at zero. This will assign values to all
triples in a set because the first symbol is either the second symbol or the first symbol has already been assigned. The first triple in the ordered list that has two distinct elements may be an exception to this and is handled as a special case. If for any triple, the corresponding hashed location in the database is occupied, the algorithm backtracks to the beginning of this set of triples and tries the next value for this set of triples. This assignment is recursively incremented by one until all triples in the set are assigned, or the algorithm fails to do so. If the assignment fails, the algorithm backtracks to the previous set. The backtracking is based on an abstract stack which holds the temporary values for assigned symbols, the triples relevant to the symbol, and the addresses required for those triples. The recursive process continues until all triples are hashed to unique addresses, or it is impossible to do it within the limits of the available address space.

procedure assign_values; /* Cook and Oldehoeft */
available_address_space ← upper_bound;
address_size_limit ← |T|;
initialize stack;
while address_size_limit ≤ available_address_space and
    symbols_assigned ≠ S do
    used_addresses ← ∅;
    symbols_assigned ← ∅;
    repeat
        find next symbol $S_j$ from $S$;
        value_assigned_to_,$S_j$ ← false;
        $T' ← \{(S_i, S_j, r_{ij}) : (S_i, S_j, r_{ij}) ∈ T\}$;
        $a(S_j) ← 0$;
        repeat
            while $a(S_i) + a(S_j) + r_{ij} ∈ used_addresses$ for any $(S_i, S_j, r_{ij}) ∈ T'$ do
                $a(S_j) ← a(S_j) + 1$;
            if $a(S_j) > address_size_limit$ and stack_not_empty then
                pop($T'$);
                pop(used_addresses);
                pop($S_j$);
\( a(S_j) \leftarrow a(S_j) + 1; \)
\( T \leftarrow T \cup T'; \)
\( \text{symbols Assigned} \leftarrow \text{symbols Assigned} \setminus \{S_j\}; \)
\( \text{value Assigned to } S_j \leftarrow \text{false}; \)
else
\( \text{used Addresses} \leftarrow \text{used Addresses} \cup \text{addresses used for } T'; \)
\( T \leftarrow T \setminus T'; \)
\( \text{symbols Assigned} \leftarrow \text{symbols Assigned} \cup \{S_j\}; \)
\( \text{value Assigned to } S_j \leftarrow \text{true}; \)
\( \text{push}(S_j); \)
\( \text{push(used Addresses)}; \)
\( \text{push}(T'); /* \text{end if} */ \)
until value Assigned to \( S_j \) or
\( (\text{stack empty and } a(S_j) > \text{address size limit}); \)
until symbols Assigned = \( S \) or stack empty;
\( \text{address size limit} \leftarrow \text{address size limit} + 1; \)

This algorithm has several weaknesses that can be removed to improve the speed of the algorithm. During experiments, we observed that the bottleneck of the algorithm lies in the last procedure – assign values – because it searches the possible address space in a depth-first manner to find the associated values. In the next section, we propose some heuristics that prune the exhaustive search for evaluating the assignment function \( a(\cdot) \). The look ahead method adds some overhead, but the improvement in terms of speed far exceeds the cost of overhead.

### 4 Heuristic Algorithm for Perfect Hash Function

We have improved the algorithm to determine the associated values for picture elements. We have devised heuristics to prune the search space to compute the hash function. In this section, we describe an enhanced algorithm and show that it provides a considerable improvement over the previous perfect hashing algorithm.
Initially, we begin with an empty set of symbolic picture elements that have been assigned values. As we proceed, we add the elements to this set that have been assigned values. The assigned value and the spatial relation value in a triple are used to develop an additive. The additive value and the available addresses for the triple are used to calculate a new value for the picture element that has not been assigned a value.

**Heuristic 1** The assignment of a value to a picture element does not have to begin at zero.

The knowledge embedded in the database is used to find the optimal start value. We begin with only those triples that have the current picture element as the second element. The first element is either the same as the second, or already has a value. The triples of interest are identified from all the triples that have second picture elements with unassigned value. The value of first element in each selected triple is now added to the integer relational value to determine the additive corresponding to the triple. This reduces each triple with distinct picture elements to a pair that has an unassigned element and an additive. The pairs are sorted on the additive in ascending order.

Next, we identify the smallest unassigned address that is equal to or larger than the smallest additive. Subtracting the smallest additive from the address gives us the first possible value for the picture element. Thus, the initial value is not necessarily zero.

**Heuristic 2** If a value fails to find valid addresses for all triples of interest, the
increment to be added to current value to determine the next possible value does not have to be unity.

The unit incremental technique in the previous algorithm (Section 3) is a slow process. We propose that the unit increment be replaced with an adaptive increment. The optimum increment step is determined from the knowledge of the elements that have already been assigned a value and the integers that show the relation between picture elements in the triples of interest.

We identify the next available address from the list of unassigned addresses. The additive and the next available address are used to calculate the next possible value for the symbol.

**Heuristic 3** *We do not need to check for all the possible values between 0 and maximum address limit to find the correct value. The search can be stopped much earlier before reaching the address limit.*

In the previous step, we develop a set of Symbol-Additive pairs and sort the set using the additive. This allows us to calculate a new test value for the picture element. If we add the new test value to the last additive, this gives us the maximum possible address for the triples using this value. If this address is greater than the maximum available address, there is no possible solution along this branch of the tree and we need to backtrack. Thus we can avoid testing for a number of solutions that will not exist.

**Heuristic 4** *We can eliminate an entire search subtree by looking ahead at the possible effect of previous symbol assignment at later nodes.*
The effect of this heuristic is best illustrated by an example. Consider the triple set \( T \) to be
\[
\{(G, G, 4), (B, A, 3), \ldots , (B, F, 7), (G, F, 7)\}
\]
where \( \ldots \) represents a number of triples. At the beginning, \( G \) is associated a value 0 and so is \( B \). It is apparent that the value assignment will result in a collision towards the last stage of the tree, and all the intermediate assignments are meaningless. \( G \) and \( B \) occur with high frequency in the triple set but should never be assigned the same value. This example is presented in its entirety as Example 1 in Section 5.

**Heuristic 5** Triples of the form \((S_i, S_i, r_{ii})\) can be converted to a normalized form \((S_i, S_i, r'_{ii})\) such that \(1 \leq r'_{ii} \leq 4\).

The triples \((S_i, S_i, r_{ii})\) and \((S_i, S_i, r'_{ii})\) convey the same information if \(|r_{ii} - r'_{ii}| = 4\).

Therefore, the triples of the form \((S_i, S_i, r_{ii})\), \(5 \leq r_{ii} \leq 8\) should be converted to the form \((S_i, S_i, r'_{ii})\) such that \(r'_{ii} = r_{ii} - 4\). The queries can be adjusted accordingly.

This heuristic not only reduces the number of triples involved in generation of a hash function but also eliminates a virtual duplication of triples.

The heuristics outlined above result in a change in two procedures of the algorithm presented in the last section – **order_triple_elements** and **assign_values**. The new procedure **order_triples_elements** is described as follows:

```plaintext
procedure order_triple_elements;
for each triple \((S_i, S_j, r_{ij}) \in T\) do
    if \( S_i = S_j \) and \( 5 \leq r_{ij} \leq 8 \) then
        change triple \((S_i, S_j, r_{ij})\) to \((S_i, S_i, r'_{ij})\) /* \( r'_{ij} = r_{ij} - 4 \) */
    else
        if \( \phi_i < \phi_j \) then
            change triple \((S_i, S_j, r_{ij})\) to \((S_j, S_i, r_{ji})\)
```
Most of the gain resulting from the heuristics affects the procedure \texttt{assign\_values}.

The new procedure presented below is the result of the new heuristics.

```plaintext
procedure assign\_values; /* Heuristic Algorithm */
available\_address\_space ← upper\_bound;
address\_size\_limit ← |T|;
initialize stack;
while address\_size\_limit ≤ available\_address\_space and symbols\_assigned ≠ S do
  free\_addresses ← \{1, 2, ..., address\_size\_limit\};
  used\_addresses ← ∅;
  symbols\_assigned ← ∅;
  repeat
    find next symbol \(S_j\) from \(S\);
    \(a(S_j) ← -1\);
    value\_assigned\_to\_\(S_j\) ← false;
    \(T' ← \{(S_i, S_j, r_{ij}) : (S_i, S_j, r_{ij}) ∈ T\}\);
    \(P ← ∅; /* Set of pairs of the type (Symbol,Additive) */\)
    free\_addr ← 0;
    repeat
      if \(T' = ∅\) then
        \(a(S_j) ← a(S_j) + 1\)
      else
        for each triple \((S_i, S_j, r_{ij}) ∈ T'\) do
          if \(S_i ≠ S_j\) then
            \(P ← P ∪ \{(S_j, m_j) : m_j ← a(S_i) + r_{ij}\};;\)
            sort \(P\) in ascending order on \(m\);
            next\_m ← first \(m_j\) in \((S_j, m_j) ∈ P\);
            free\_addr ← smallest address in free\_addresses that is greater than the current value of free\_addr;
            if \((S_j, S_j, r_{jj}) ∉ T'\) then
              \(a(S_j) ← free\_addr - next\_m\)
            else
              \(a(S_j) ← (free\_addr - r_{jj}) \text{ div } 2;\)
        if \((T' = ∅ \text{ and } a(S_j) > address\_space\_limit) \text{ or } (a(S_j) + max(m_j) > address\_space\_limit)\) and stack\_not\_empty then
          pop\(T'\);
        pop\(S_j\);
    
```

18
pop(used_addresses);
pop(free_addresses);
\[ T \leftarrow T \cup T'; \]
symbols_assigned \leftarrow symbols_assigned \setminus \{S_j\};
value_assigned_to_{S_j} \leftarrow false;
else
used_addresses \leftarrow used_addresses \cup \text{set of addresses used for } T';
free_addresses \leftarrow free_addresses \setminus \text{set of addresses used for } T';
\[ T \leftarrow T \setminus T'; \]
symbols_assigned \leftarrow symbols_assigned \cup \{S_j\};
value_assigned_to_{S_j} \leftarrow true;
push(free_addresses);
push(used_addresses);
push(S_j);
push(T');
until value_assigned_to_{S_j} \text{ or } (\text{stack_empty and } a(S_j) > \text{address_size_limit});
until symbols_assigned = S \text{ or stack_empty;}
address_size_limit \leftarrow address_size_limit + 1;

The queries are similarly mapped onto the set of triples. The assigned values of symbols are used to determine if the hashed values are in the database.

5 Experimental Results

To evaluate the performance of our heuristic algorithm vis-à-vis the previous algorithm (Section 3), we developed a program corresponding to each algorithm for comparing the performance of the two algorithms. We choose the number of nodes visited in the search tree as our evaluation criterion as it does away with the optimization of code to speed up the programs. If we use time as the evaluation criterion, the optimization of code plays a major role in time statistics which is not of our concern.

The programs are written in Quintus Prolog because of its strong string matching
and back tracking facilities. The programs are implemented on a Sun SPARC 2 workstation running under SunOS 4.1 operating system. The CPU time is provided for the purpose of relative speed measurement though it is not our evaluation criterion.

Several test sets are used to prove the effectiveness of our heuristic algorithm. The first example is taken from [19] and another example is selected from [1] because of their use in the literature. More examples were developed out of the two examples to examine the effect of the algorithm on different types of data. The complexity is measured in terms of the number of times the algorithm backtracks to get to the desired perfect hash function.

In all the following examples, Algorithm 1 refers to the previous algorithm described in Section 3 [19, 18] and Algorithm 2 refers to our heuristic algorithm presented in Section 4.

**Example 1.** The first example consists of 4 subpictures as shown below using 2D string representations and triples [19].

\[
\begin{align*}
f_1 &= (A < B < C, B < C < A) & T_1 &= \{(A, B, 6), (A, C, 6), (B, C, 8)\} \\
f_2 &= (G_1 < G_2 < F, G_2 < G_1 = F) & T_2 &= \{(F, G_1, 3), (F, G_2, 4), (G_1, G_2, 4)\} \\
f_3 &= (B < F, B = F) & T_3 &= \{(B, F, 7)\} \\
f_4 &= (A = C < B : E = D, A = B : E < C = D) & T_4 &= \{(A, B, 7), (A, C, 1), (A, D, 8), (A, E, 7), (B, C, 2), (B, D, 1), (B, E, 9), (C, D, 7), (C, E, 6), (D, E, 5)\}
\end{align*}
\]

The composite image contains 17 unique triples represented below as \(T\). The frequency distribution of symbolic picture elements is: 6 A’s, 7 B’s, 6 C’s, 4 D’s, 4 E’s, 3 F’s, and 4 G’s. The ordered set of symbol-frequency pairs is represented by
\{(G, 4), (B, 7), (A, 6), (C, 6), (E, 4), (D, 4), (F, 3)\}. The set of symbols $S$ and the set of triples $T$ corresponding to 4 subpictures, after ordering the triples according to symbol frequency, and sorting them on the second picture element, is

\[
S = \{A, B, C, D, E, F, G\}
\]

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

This example has been taken from [19]. However, the set of triples $T$ presented above is different from the set of triples $T'$ presented in [19]. After ordering of triples according to symbol frequency, and sorting them on the second picture element, the set of triples as presented in [19] appears as

\[
T' = \{(G, G, 4), (B, A, 6), (B, A, 7), (A, C, 6), (A, C, 1), (B, C, 2), (B, C, 8), (A, D, 8), (B, D, 1), (C, D, 7), (A, E, 7), (B, E, 9), (C, E, 6), (D, E, 5), (B, F, 7), (G, F, 3), (G, F, 4)\}
\]

The discrepancy between $T$ and $T'$ arises because of triples $(B, A, 3)$, $(B, A, 2)$, $(G, F, 8)$, and $(G, F, 7)$. In each of these triples, the relationship $r_{ij}$ in the triple is not adjusted when the triples are ordered in the development of $T'$. The other difference is in the triple $(E, D, 1)$ which is because of different ways of resolving the same frequency during the sorting procedure. Therefore, with triple set as $T'$, the resultant ordering and order of assignments do not match with those in [19], even though the symbol frequency assignments are the same.
The associated values for picture elements using $T$ as the basis are as follows

$$(A, 0), (B, 0), (C, 4), (D, 0), (E, 8), (F, 0), (G, 6)$$

It is easy to see the amount of pruning, due to Heuristic 4, in the above example, especially considering that the root was traversed six times (the associated value of $G$ is computed to be 6). Since $G$ occurs at the root of the search tree, its associated value is always incremented in steps of 1, if no assignment is possible with its previous value. Moreover, the assignment of a value to $B$ occurs just a step away from root and cannot be assigned the same value as $G$, or a value that is one more than $G$, because the triples $(B, F, 7)$, $(G, F, 8)$, and $(G, F, 7)$ will collide. All these triples are pruned at the bottom of the search tree in the depth-first search algorithm, whereas our algorithm prunes them, as soon as a value is assigned to the lower frequency element, by look-ahead technique.

We performed a second experiment by applying the heuristic of Cook and Oldehoeft [18]. This heuristic states that better results can be obtained by allowing a 10% spare address space, and is called the “10%-rule”. The associated values computed with the application of heuristic in this example are the same as the ones computed without the heuristic.

The overall results showing the number of search-tree nodes visited for both the previous algorithm [19] and our heuristic algorithm are shown in Table 2. The first row in this table shows the results without applying the 10%-rule while the second row shows the result with the application of the rule. It is observed that a minimal perfect hashing function does not exist and both the algorithms support this claim. However, the number of search-tree nodes visited shows a remarkable decrease by
about 90% due to our heuristics, with or without the 10%-rule. Also, it may be noted that using the 10%-rule provides us with about 21% reduction in the number of nodes visited in either algorithm.

<table>
<thead>
<tr>
<th>Without 10%-rule</th>
<th>17</th>
<th>18</th>
<th>202465</th>
<th>21749</th>
<th>89.26%</th>
</tr>
</thead>
<tbody>
<tr>
<td>With 10%-rule</td>
<td>17</td>
<td>19</td>
<td>159580</td>
<td>17018</td>
<td>89.34%</td>
</tr>
<tr>
<td>Gain by 10%-rule</td>
<td></td>
<td></td>
<td>21.18%</td>
<td>21.75%</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Number of nodes visited in Algorithm 1 and Algorithm 2 – Example Data 1

**Example 2.** This example illustrates that Cook and Oldehoeft’s 10%-rule is a good start but does not provide an upperbound for the address space in calculating the hash function. The data set in this example is the same as the set $T'$ in Example 1, and appears as column 4 of the example in [19]. In this example, the set of symbolic picture elements $S$ and the triple set $T$ are defined as

$$S = \{A, B, C, D, E, F, G\}$$

$$T = \{(G, G, 4), (B, A, 6), (B, A, 7), (A, C, 6), (A, C, 1), (B, C, 2), (B, C, 8), (A, D, 8), (B, D, 1), (C, D, 7), (A, E, 7), (B, E, 9), (C, E, 6), (D, E, 5), (B, F, 7), (G, F, 3), (G, F, 4)\}$$

The example contains 6 A’s, 7 B’s, 6 C’s, 4 D’s, 4 E’s, 3 F’s, and 4 G’s. G is assigned a higher frequency as it occurs twice in a triple. After ordering the triples and sorting them on the basis of second element, $T$ becomes

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

23
The associated values computed by both the algorithms are given by

\((A, 5), (B, 8), (C, 1), (D, 1), (E, 1), (F, 0), (G, 1)\)

The comparative results are presented in Table 3. The results in the first row are computed by starting with an initial address space of size 17 while the second row, based on the 10%-rule, started with an address space of size 19. The address space size is incremented by one for recomputation if a solution is not possible. It should be noted that in this example, the algorithm could not find a solution even after allowing for a 10% extra space at the beginning and had to add more space to achieve a perfect hashing.

<table>
<thead>
<tr>
<th></th>
<th>Data Size</th>
<th>Address Size</th>
<th>Alg. 1</th>
<th>Alg. 2</th>
<th>Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Without 10%-rule</td>
<td>17</td>
<td>20</td>
<td>335635</td>
<td>51444</td>
<td>84.67%</td>
</tr>
<tr>
<td>With 10%-rule</td>
<td>17</td>
<td>20</td>
<td>187654</td>
<td>29913</td>
<td>84.06%</td>
</tr>
<tr>
<td>Gain by 10%-rule</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>44.09%</td>
</tr>
</tbody>
</table>

**Table 3**: Number of nodes visited in Algorithm 1 and Algorithm 2 – Example Data 2

**Example 3.** This example differs from Example 2 only in two triples where the position of \(A\) and \(B\) is exchanged in the initial data. In this example, the set of symbolic picture elements \(S\) and the triple set \(T\) are defined as

\[
S = \{A, B, C, D, E, F, G\}
\]

\[
T = \{(G, G, 4), (A, B, 6), (A, B, 7), (A, C, 6), (A, C, 1), (B, C, 2), (B, C, 8), (A, D, 8), (B, D, 1), (C, D, 7), (A, E, 7), (B, E, 9), (C, E, 6), (D, E, 5), (B, F, 7), (G, F, 3), (G, F, 4)\}
\]
The example contains 6 A’s, 7 B’s, 6 C’s, 4 D’s, 4 E’s, 3 F’s, and 4 G’s. G is assigned a higher frequency as it occurs twice in a triple. After ordering the triples and sorting them on the basis of second element, T is described as

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

The associated values, as computed by both the algorithms, are given by

\( (A, 4), (B, 1), (C, 0), (D, 4), (E, 7), (F, 11), (G, 0) \)

The comparative number of search-tree nodes visited with and without the application of Cook and Oldehoeft’s 10%-rule are presented in Table 4.

<table>
<thead>
<tr>
<th>Data Size</th>
<th>Address Size</th>
<th>Alg. 1</th>
<th>Alg. 2</th>
<th>Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Without 10%-rule</td>
<td>17</td>
<td>19</td>
<td>226121</td>
<td>29298</td>
</tr>
<tr>
<td>With 10%-rule</td>
<td>17</td>
<td>19</td>
<td>3410</td>
<td>930</td>
</tr>
<tr>
<td>Gain by 10%-rule</td>
<td></td>
<td></td>
<td>98.49%</td>
<td>96.83%</td>
</tr>
</tbody>
</table>

Table 4: Number of nodes visited in Algorithm 1 and Algorithm 2 – Example Data 3

**Example 4.** This example consists of 12 subpictures and has been selected from [1]. It shows that in some cases, the 10%-rule given by Cook and Oldehoeft may overestimate the optimal address space. Even though the decrease in the number of search-tree nodes visited may be significant, the 10%-rule does not always guarantee such gain (Examples 1 and 2).

The composite picture used in this example consists of 24 unique triples. The
symbols and the triples input to the algorithm are given by

\[ S = \{A, B, C, D, E, F, G, H\}\]
\[ T = \{(A, A, 2), (A, B, 2), (A, B, 4), (A, B, 7), (A, C, 1), (A, C, 2), (A, C, 8),
(A, D, 2), (A, D, 4), (A, D, 6), (A, E, 4), (A, F, 2), (B, C, 2), (B, C, 4),
(B, D, 4), (B, D, 6), (B, D, 7), (C, D, 4), (E, F, 1), (E, F, 2), (E, G, 6),
(E, H, 3), (F, G, 3), (G, H, 2)\}\]

The frequency distribution of symbols is: 13 A’s, 8 B’s, 6 C’s, 7 D’s, 5 E’s, 4 F’s, 3 G’s, and 2 H’s. After ordering the triple elements and sorting the triples on the basis of second element, T is described as

\[ T = \{(A, A, 2), (A, B, 7), (A, B, 4), (A, B, 2), (B, D, 7), (B, D, 6), (B, D, 4),
(A, D, 6), (A, D, 4), (A, D, 2), (D, C, 8), (B, C, 4), (B, C, 2), (A, C, 8),
(A, C, 2), (A, C, 1), (A, E, 4), (E, F, 2), (E, F, 1), (A, F, 2), (F, G, 3),
(E, G, 6), (G, H, 2), (E, H, 3)\}\]

Both the algorithms found associated values for the symbols in an address space of size 26. The associated values are given by:

\[(A, 3), (B, 13), (C, 2), (D, 5), (E, 2), (F, 0), (G, 8), (H, 16)\]

In this example, the 10%-rule of Cook and Oldehoeft resulted in different associated values and used an address space of size 27. The values computed by both the algorithms, in this case, are:

\[(A, 0), (B, 3), (C, 12), (D, 2), (E, 12), (F, 13), (G, 7), (H, 9)\]
The comparative number of iterations with and without the application of Cook and Oldehoeft’s heuristic are presented in Table 5.

<table>
<thead>
<tr>
<th>Data Address No. of nodes visited</th>
<th>Net Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Without 10%-rule</td>
<td></td>
</tr>
<tr>
<td>Size</td>
<td>Size</td>
</tr>
<tr>
<td>24</td>
<td>26</td>
</tr>
<tr>
<td>With 10%-rule</td>
<td></td>
</tr>
<tr>
<td>Size</td>
<td>Size</td>
</tr>
<tr>
<td>24</td>
<td>27</td>
</tr>
<tr>
<td>Gain by 10%-rule</td>
<td></td>
</tr>
<tr>
<td>98.64%</td>
<td>98.16%</td>
</tr>
</tbody>
</table>

Table 5: Number of nodes visited in Algorithm 1 and Algorithm 2 – Example Data 4

**Example 5.** The example was developed by replacing the triple \((A, A, 2)\) in Example 4 by an equivalent triple \((A, A, 6)\). The data was expected to measure the effect of changing the triple into an equivalent triple. The symbols and the triples input to the algorithm are given by

\[
S = \{A, B, C, D, E, F, G, H\}
\]

\[
T = \{(A, A, 6), (A, B, 2), (A, B, 4), (A, B, 7), (A, C, 1), (A, C, 2), (A, C, 8),
(A, D, 2), (A, D, 4), (A, D, 6), (A, E, 4), (A, F, 2), (B, C, 2), (B, C, 4),
(B, D, 4), (B, D, 6), (B, D, 7), (C, D, 4), (E, F, 1), (E, F, 2), (E, G, 6),
(E, H, 3), (F, G, 3), (G, H, 2)\}
\]

The frequency distribution of the picture elements is the same as in Example 4, that is, 13 A’s, 8 B’s, 6 C’s, 7 D’s, 5 E’s, 4 F’s, 3 G’s, and 2 H’s. After ordering, the set of triples in the heuristic algorithm appears to be the same as the one in Example 4 because the triple \((A, A, 6)\) is changed back to \((A, A, 2)\) due to Heuristic 5. However, the set of triples in previous algorithm (Section 3) is given by

\[
T = \{(A, A, 6), (A, B, 7), (A, B, 4), (A, B, 2), (B, D, 7), (B, D, 6), (B, D, 4),
\]
(A, D, 6), (A, D, 4), (A, D, 2), (D, C, 8), (B, C, 4), (B, C, 2), (A, C, 8),
(A, C, 2), (A, C, 1), (A, E, 4), (E, F, 2), (E, F, 1), (A, F, 2), (F, G, 3),
(E, G, 6), (G, H, 2), (E, H, 3)}

In this case, each algorithm starts with the same set of triples. However, our
algorithm changed the triple (A, A, 6) to (A, A, 2) due to Heuristic 5. The heuristic
algorithm computed a solution in an address space of 26 while the previous algorithm
computed a solution in an address space of size 27. Our algorithm computed the same
associated values as in Example 4. The associated values computed by the previous
algorithm are given by

(A, 1), (B, 14), (C, 7), (D, 0), (E, 9), (F, 1), (G, 9), (H, 15)

The comparative number of backtracks for the two algorithms, with and without
the heuristic of Cook and Oldehoeft, are presented in Table 6.

<table>
<thead>
<tr>
<th></th>
<th>Data</th>
<th>Address Size</th>
<th>No. of nodes visited</th>
<th>Net Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Size</td>
<td>Alg. 1</td>
<td>Alg. 2</td>
<td>Alg. 1</td>
</tr>
<tr>
<td>Without 10%-rule</td>
<td>24</td>
<td>27</td>
<td>26</td>
<td>96011</td>
</tr>
<tr>
<td>With 10%-rule</td>
<td>24</td>
<td>27</td>
<td>27</td>
<td>15110</td>
</tr>
<tr>
<td>Gain by 10%-rule</td>
<td></td>
<td></td>
<td></td>
<td>84.26%</td>
</tr>
</tbody>
</table>

Table 6: Number of nodes visited in Algorithm 1 and Algorithm 2 – Example Data 5

**Example 6.** The example was designed to observe the effect of two equivalent triples
of the form \((S_i, S_i, r_{ii})\) and \((S_i, S_i, r'_{ii})\) in the set of triples such that \(|r_{ii} - r'_{ii}| = 4\).
The data set was developed by taking a union of the data sets for Examples 4 and 5.
The resultant set of triples was made up of 25 triples two of which were equivalent.
The symbolic picture elements and the set of triples are given by

\[ S = \{A, B, C, D, E, F, G, H\} \]

\[ T = \{(A, A, 2), (A, A, 6), (A, B, 4), (A, B, 7), (A, C, 1), (A, C, 2), \\
              (A, C, 8), (A, D, 2), (A, D, 6), (A, E, 4), (A, F, 2), (B, C, 2), \\
              (B, C, 4), (B, D, 4), (B, D, 6), (B, D, 7), (C, D, 4), (E, F, 1), (E, F, 2), \\
              (E, G, 6), (E, H, 3), (F, G, 3), (G, H, 2)\} \]

The frequency distribution for different picture elements is given by: 15 A’s, 8 B’ s, 6 C’s, 7 D’s, 5 E’s, 4 F’s, 3 G’s, and 2 H’s. After ordering the elements, the set of triples for our algorithm is the same as the one in Example 4 and 5 because one of the equivalent triples is removed due to Heuristic 5. However, the sorted set of ordered triples in previous algorithm (Section 3) is given by

\[ T = \{(A, A, 6), (A, A, 2), (A, B, 7), (A, B, 4), (A, B, 2), (B, D, 7), (B, D, 6), \\
              (B, D, 4), (A, D, 6), (A, D, 4), (A, D, 2), (D, C, 8), (B, C, 4), (B, C, 2), \\
              (A, C, 8), (A, C, 2), (A, C, 1), (A, E, 4), (E, F, 2), (E, F, 1), (A, F, 2), \\
              (F, G, 3), (E, G, 6), (G, H, 2), (E, H, 3)\} \]

Again, like in Example 5, each algorithm works with a separate set of triples. The heuristic algorithm computed a solution using an address space of size 26 while the previous algorithm computes a solution requiring an address space of size 28. Our algorithm computes the same associated values as those in Example 4. The associated values computed by the previous algorithm are given to be

\[(A, 1), (B, 15), (C, 8), (D, 0), (E, 4), (F, 9), (G, 16), (H, 6)\]
The comparative number of search-tree nodes visited for the two algorithms, with and without the 10%-rule of Cook and Oldehoeft, are presented in Table 7.

<table>
<thead>
<tr>
<th>Without 10%-rule</th>
<th>Data</th>
<th>Address Size</th>
<th>No. of backtracks</th>
<th>Net Size</th>
<th>Alg. 1</th>
<th>Alg. 2</th>
<th>Alg. 1</th>
<th>Alg. 2</th>
<th>Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>25</td>
<td>28</td>
<td>26</td>
<td>213396</td>
<td>22625</td>
<td>89.40%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>With 10%-rule</td>
<td>25</td>
<td>28</td>
<td>27</td>
<td>29938</td>
<td>417</td>
<td>98.61%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gain by 10%-rule</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>85.97%</td>
<td>98.16%</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7: Number of nodes visited in Algorithm 1 and Algorithm 2 – Example Data 6

The comparative results on all data sets, with and without the 10%-rule, are summarized in Tables 8 and 9. The two tables also show the computer time used for each data set in minutes.

<table>
<thead>
<tr>
<th>Data Set</th>
<th>Data Size</th>
<th>Address Size</th>
<th>No. of nodes Visited</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>17</td>
<td>18</td>
<td>202465</td>
<td>9:43</td>
</tr>
<tr>
<td>2</td>
<td>17</td>
<td>20</td>
<td>335635</td>
<td>16:23</td>
</tr>
<tr>
<td>3</td>
<td>17</td>
<td>19</td>
<td>226121</td>
<td>11:04</td>
</tr>
<tr>
<td>4</td>
<td>24</td>
<td>26</td>
<td>147754</td>
<td>7:36</td>
</tr>
<tr>
<td>5</td>
<td>24</td>
<td>27</td>
<td>96011</td>
<td>4:42</td>
</tr>
<tr>
<td>6</td>
<td>25</td>
<td>28</td>
<td>213396</td>
<td>11:03</td>
</tr>
<tr>
<td>Average gain</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 8: Number of nodes visited and run-time for both algorithms without 10%-rule

### 6 Conclusion

In this paper, we have presented heuristics to improve the speed of the algorithm to calculate associated values for symbolic picture elements in an image database.
<table>
<thead>
<tr>
<th>Data Set</th>
<th>Data Size</th>
<th>Address Size</th>
<th>Nodes Visited</th>
<th>Gain</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alg. 1</td>
<td>Alg. 2</td>
<td>Alg. 1</td>
<td>Alg. 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>17</td>
<td>19</td>
<td>19</td>
<td>159580</td>
<td>17018</td>
</tr>
<tr>
<td>2</td>
<td>17</td>
<td>20</td>
<td>20</td>
<td>187654</td>
<td>29913</td>
</tr>
<tr>
<td>3</td>
<td>17</td>
<td>19</td>
<td>19</td>
<td>3410</td>
<td>930</td>
</tr>
<tr>
<td>4</td>
<td>24</td>
<td>27</td>
<td>27</td>
<td>2008</td>
<td>417</td>
</tr>
<tr>
<td>5</td>
<td>24</td>
<td>27</td>
<td>27</td>
<td>15110</td>
<td>417</td>
</tr>
<tr>
<td>6</td>
<td>25</td>
<td>28</td>
<td>27</td>
<td>29938</td>
<td>417</td>
</tr>
</tbody>
</table>

Average gain 86.87%

Table 9: Number of nodes visited and run-time for both algorithms with 10%-rule system. The associated values are used to compute a perfect hash function. This hash function is employed to resolve a retrieval query in an image database system. The previous hash function computation, given by Chang and Lee, has several drawbacks: associated value computation starts with zero and on failure, the value is incremented by one; there is no global lookahead mechanism. We have presented heuristics to speed up the computation of associated values resulting in an A* algorithm. This algorithm begins the assignment of associated values with an optimal starting value and upon failure, increments the value in an adaptive manner rather than unity. Moreover, the look-ahead mechanism helps in pruning the search space. Finally, the normalization of triples with identical objects helps to reduce the search and obviates the need for query expansion to account for different possibilities. The experimental results demonstrate that on the average, there is more than 85% saving in the number of search tree nodes examined due to the use of heuristics. It is demonstrated that even though Cook and Oldehoeft’s 10%-rule improves the overall performance of the algorithm, it is neither necessary nor sufficient to guarantee a perfect hashing within 10% extra address space.
References


