Augmented Order Preserving
Minimal Perfect Hash Functions for Very Large
Digital Libraries

Amjad M Daoud, Hussain AbdeJaber, Jafar Ababneh,
WISE University
Amman Jordan

ABSTRACT
Rapid access to information is essential for a wide variety of retrieval systems and applications. Hashing has long been used when the fastest possible direct search is desired, but was considered an exotic [15] and not appropriate when sequential or range searches are also required. To change that, we extend order preserving perfect hash [11] functions to handle sequential access to lexicographically ordered records.

To implement this, the lexicographically sorted key set is compressed with prefix compression and stored in blocks or pages in a similar fashion to the leaf pages of a B+ tree with prefix compression [9]. The block number and the key prefix offset within the block are combined to form the key address. The key address is augmented with a signature of the prefix to form the key data. The key data is blended into the function specification to produce an augmented order preserving minimal perfect hash function.

Our algorithm uses the bipartite graph approach to avoid degenerate edges problems. It relaxes the acyclic requirement in random graphs presented in [7] and can tolerate the presence of cyclic components. Moreover the algorithm is designed to avoid the conditions described in [7] that make the Fox, Chen, Daoud, and Heath approach [11] exponential. Examples of these conditions are given along with how the algorithm overcomes them. The algorithm produces OPMPHFs with much higher success rates than the acyclic hypergraph approach [7] and mostly from the first trial.

Keywords: Perfect Hashing, Digital Libraries, Random Graphs, Access Methods, Order Preserving, Web Maps.

1. INTRODUCTION

1.1 Motivation:
This work was in part motivated by our investigations that deal with tightly integrating information retrieval with relational databases [9]. We have prototyped internal structures in Oracle Rdb 6.1 and 7 which was subsequently implemented as Index Organized Tables in Oracle 8, 8i, 9i and 10g and offered to the IR community under the term cooperative indexing [25]. Using prefix compression and hashing, the internal structures could handle the large requirements of an inverted list [23]. However, cooperative indexing [9] taxes internal structures of a database resulting in many different types of blocks, which in turn slows down the overall performance. Generally, Gray [15] argues that implementing more than one access method is not really beneficial. The choices as we stand are in favor of tree structures such as B-trees and ternary search trees [1] for external storage and hashing for buffer (cache) management [1]. This paper proposes replacing B+ trees with perfect hashing for external storage as it clearly offers many advantages (i.e. minimal, perfect, single access, any priori order, etc.).

1.2 APPLICATIONS
There are numerous applications for our algorithm; some of them are novel such as web maps that are used by Page-rank ranking algorithms [WebGraph][WebBase]. The others are well known but often are associated with tree structures such as partial matching (wild card searching), and near-neighbor searching.

1.2.1 The dictionary
The dictionary membership is well known and is important for digital libraries, information retrieval systems, and compiler optimization.

1.2.2 The Web Map
Web maps such as WebGraph and WebBase are used to study the structure of the World Wide Web. OPMPHF's have been successfully used to manage very large graphs, exploiting modern compression techniques. Storing snapshots of the Web helps the tuning of ranking algorithms such as Google PageRank algorithm.

1.2.3 Partial Match Searching
A third application is partial match searching. A query string may contain both regular letters and the "don't care" character ".". Searching the dictionary for the pattern ".o.o.o" matches the single word rococo, while the pattern "a.a.a" matches many words, including banana, casaba, and pajama. Rivest [22] presents an algorithm for partial match searching in tries: take the single given branch if a letter is specified, for a don't-care character, recursively search all branches. Unspecified positions [21] at the front of the query word are dramatically more costly than unspecified characters at the end of the word.

Order preserving perfect hash function can be used to alleviate positional branching problems that tree structures suffer from. Two OPMPHF's can be computed, one for regular keys and the other for their reverses. When the pattern has many unspecified positions in the front we use the reverse OPMPHF. For a don't-care character at the end of the pattern, strings that match the pattern are likely to be successors or predecessors of the pattern depending on whether the original key set was sorted in ascending or descending order. A class of OPMPHF's should yield better performance than a fixed tree structure.
1.2.4 Near-Neighbor Searching with Common Prefix

Finally, the fourth application is near-neighbor searching in a set of strings that share a common prefix: we are to find all words in the dictionary that are within a given Hamming distance of a query word [1] that share a common prefix of a desired length. For instance, a search for all words within distance two of “soda” with common prefix “s” of length 1 finds. Having our key set sorted, stored in blocks, and compressed with prefix compression indeed proves beneficial for near-neighbor searches as they are likely to be stored in the same block and share the same prefix. Near-neighbor searching can be used to lookup efficiently valid derivational surface forms from roots in languages such as Arabic.

2 AUGMENTED MINIMAL PERFECT HASH FUNCTIONS

There are many popular methods of order-preserving perfect hashing algorithms [11][7]. However, they all suffer from their inefficiency to handle unsuccessful searches and partial match queries [1]. Both approaches need to store the actual keys to check if a key is in the set or not. For unsuccessful search the search time is proportional to the key length. In tries, B+ trees with prefix compression, and ternary search structures unsuccessful searches terminate much earlier and the search time is not proportional to key length.

Fox et. al. published algorithms [11] are proved to have expected exponential running times under certain conditions [7] but almost linear most of the time (validated experimentally). Bad behavior was so rare that it had not been encountered during the experiments with large key sets.

Havas et. al. algorithms [7] have high probability of failure due to degenerate edges (one out of 10 trials succeeds as evident from the MG1 system and MG4J implementations). Both algorithms preserve any priori order not only the lexicographic order.

In this paper, we present a probabilistic algorithm that is linear, designed to produce augmented compact OPMHFs in the presence of conditions described in [7] (see the example and analysis) and has a high success rate. To do this, we do the following:

- We use the bipartite random graph approach and so we do not have degenerate edges. However the graph is almost always cyclic.
- We allow cycles in our mapping step and device ways to break them. This allows us to produce more compact order preserving perfect hash functions than acyclic hypergraphs. The number of unsuccessful trials is greatly reduced (i.e. from 13 trials to 1).
- The ratio of keys (edges) to vertices is chosen such that the cyclic random graph is sparse. Edges forming cycles are remapped to other isolated vertices and other vertices designated as “free choice” vertices. After the remapping, each edge has a vertex which can be used to record data associated with it or its keydata.
- We exploit isolated vertices to remap keys that form a cycle and use a mark bit to distinguish them similar to [11]. For a random bipartite graph with ratio words/key ≥ 1, more than 13% of the vertices in G are isolated. Our marking scheme is much simpler and produces more compact functions.
  - We preserve the lexicographic order of the input key set and use it for prefix compression.
  - Keys prefixes signatures are stored in the OPMHF specification. The keys themselves are not since the mapping is one to one yielding a huge saving over earlier approaches.
  - Unsuccessful searches are terminated by the OPMHF itself.
  - The ordering step enumerates all orderings for cyclic components of length 2 in the random bipartite graph. Since the number of cyclic components is small for ratios slightly higher than 1, the time complexity of the algorithm is the same. Half of these vertices are “free choice” vertices with the flexibility anyone of the two vertices in a cycle of length 2 can be a “free choice”. However once used, the other partner cannot.
  - We also use one vertex in acyclic components (could be any vertex in the component). It can be easily shown that every acyclic component has precisely one vertex that can be used to store key data about indirect edges. Furthermore, this vertex can be any vertex in that acyclic component.
  - For cycles of length 2 that cause the FCDH algorithm [11] to fail (see [7] for a proof), we produce all orderings of the vertices in the cycle. There are exactly two of these for each cycle. The number of cycles in the random bipartite graph is bounded by log (n). So producing all orderings for these 2-cycles does not change the time complexity of our algorithm. This step allows us to use one precious location per cyclic component with the added flexibility of which one of the two can be marked as a “free choice” vertex coined by [Daoud]. A “free choice” vertex can be initialized randomly or assigned any value.
  - If all components are acyclic, the remapping is not required and no mark bits are generated.
  - The classes of functions searched to produce an OPMHF references the g table only twice while the Havas et al. Algorithm [11] references the g table 3 times. This extra “random” access to main memory often takes as much time as executing hundreds of instructions on modern CPUs.

The class of functions searched is:

\[ H_{best}(k) = g(h_0(k) \cdot g(h_1(k)) \cdot g(h_2(k)) \mod 2^r) \]
\[ if \ (mark(h_0(k)) \ OR \ mark(h_1(k)) \ OR \ mark(h_2(k)) \) \]
\[ H_{load}(k) = h_0(k) \cdot g(h_1(k)) \cdot g(h_2(k)) \mod n \]

otherwise.

The dependency graph is traversed so that we partition the set.
of keys into a sequence of levels called a tower. If the vertex ordering is \(v_1, \ldots, v_t\), then the level of keys \(K(v_i)\) corresponding to a vertex \(v_i\), \(1 \leq i \leq t\), is the set of edges incident both to \(v_i\) and to a vertex earlier in the ordering. More formally:

\[
K(v_i) = \{k \mid h_1(k) = v_i, h_2(k) = v_s, v_s < v_i \text{ if } 0 \leq v_i \leq r-1\}
\]

and

\[
K(v_i) = \{k \mid h_1(k) = v_i, h_2(k) = v_s, v_s < v_i \text{ if } r \leq v_i \leq 2r-1\}.
\]

Our algorithm builds on the following four important observations:

- The vertex ordering \(v_1, \ldots, v_t\) of every connected component in the graph \(G\) must start with a vertex \(v\) such that \(|K(v)| = 0\) since it is the first vertex to visit in the ordering. This vertex is a “free choice” vertex.
- Edges that can be directed are always in levels with \(|K(v)| = 1\), while edges that must be remapped occur in levels with \(|K(v)| > 1\).
- Acyclic components when ordered yield a vertex ordering with all levels having a cardinality of one \(|K(v)| = 1\) and all of them qualify as “free choices”.
- Cyclic components with cycles of length of 2 when ordered yield a vertex ordering with first level \(|K(v)| = 0\) and the second level \(|K(v)| = 2\). The two levels are interchangeable. The choice of which one is a “free choice” enhances the success rate during the search step significantly.

3. Augmenting the OPMPHF

An OPMPHF [11][7] preserves any priori order but has no inherent knowledge about keys it hashes. The priori order can be used to store key order in a dictionary or an offset for variable length keys [Havas]. In contrast, our algorithm makes proper use of lexicographical order to aid unsuccessful searches and store bits of the key in the g-table that stores the specification of the OPMPHF.

To implement this, the lexicographically sorted key set is compressed with prefix compression and stored in blocks or pages in a similar fashion to the leaf pages of a B+ tree with prefix compression [9]. The block number and the key prefix offset within the block are combined to form the key address. The key address is augmented with a portion of the prefix to form the key data. The key data is blended into the function specification to produce an augmented order preserving minimal perfect hash function. The key data guides the searching and allows range and membership queries (i.e. is the key \(k\) in the set \(S\)) to be implemented efficiently. The key data size is bounded by \(\log n\) to keep the overall size of the function manageable and equal to \(c n \log n\).

3.1 Approach

For \(n\) keys, if our graph has somewhat more than \(n\) vertices (i.e., if ratio > 1, where ratio is defined as \(2r/m\)), then there should be enough space to specify the OPMPHF. In a random graph of this size, a significant number of vertices will have zero degree. We use indirection for handling some of the keys.

The class of functions searched is:

\[
H_{\text{Rand}}(k) = g[h_0(k) + g[h_1(k)] + g[h_2(k)] \mod 2r] \text{ if } (\text{mark}(h_0(k))) \text{ OR mark}(h_2(k)) \text{ and } H_{\text{Rand}}(k) = h_0(k) + g[h_1(k)] + g[h_2(k)] \mod n \text{ otherwise}.
\]

The algorithm for selecting proper \(g\) values and setting mark (indirection) bits for vertices in \(G\) consists of the three steps: Mapping, Ordering, and Searching. Each step, along with implementation details, will be described in a separate subsection below.

3.1 The Mapping Step

The Mapping step builds random tables the three functions \(h_0, h_1, \text{ and } h_2\) that map each key \(k\) into a unique triple \((h_0(k), h_1(k), h_2(k))\). The \(h_0(k), h_1(k), h_2(k)\) triples are used to build a bipartite graph (called dependency graph [Sager, Fox, Havas]).

Figure 3.1 details the Mapping step. Let \(k_1, k_2, \ldots, k_0\) be the set of keys. The \(h_0(k), h_1(k), \text{ and } h_2(k)\) functions are selected (1) as the result of building tables of random numbers. If triples are not distinct, new random tables are generated (3), defining new \(h_0(k), h_1(k), h_2(k)\) functions. The probability that random tables must be generated more than once is very small [11]. Therefore, the expected time for the Mapping step is \(O(n)\).

3.2 The Ordering Step

The Ordering step orders the connected components of the dependency graph so as to partition the set of keys into an ordered sequence vs called tower [Sager]. Let \(vs = \{v_1, \ldots, v_t\}\) then the levels of the tower \(K(v_i)\) is the set of edges incident both to \(v_i\) and to a vertex earlier in the ordering. More formally [Fox]

\[
K(v_i) = \{k \mid h_1(k) = v_s, h_2(k) = v_s, v_s < v_i \text{ if } 0 \leq v_i \leq r-1\}
\]

and

\[
K(v_i) = \{k \mid h_2(k) = v_s, h_2(k) = v_s, v_s < v_i \text{ if } r \leq v_i \leq 2r-1\}.
\]

Our algorithm builds on the following four important observations:

1. The vertex ordering of every connected component in the graph \(G\) must start with a vertex \(v\) such that \(|K(v)| = 0\). This vertex is a “free choice” vertex.
2. Edges that can be directed are always in levels with \(|K(v)| = 1\), while edges that must be redirected occur in levels with \(|K(v)| > 1\).
3. Acyclic components when ordered yield a vertex ordering with all levels having a cardinality of one \(|K(v)| = 1\) and all of them qualify as “free choices”.

(1) build random tables for \(h_0, h_1, \text{ and } h_2\)

(2) for each \(i \in \{1 \ldots n\}\) do edge[\(i\)] = distinct triple \((h_0(k), h_1(k), h_2(k))\)

(3) if triples not distinct then repeat from step (1).

Figure 3.1: The Mapping Step
4. Cyclic components with cycles of length of 2 when ordered yield a vertex ordering with first level $|K(v)| = 0$ and the second level $|K(v)| = 2$. The two levels are interchangeable. The choice of which one is a “free choice” enhances the success rate during the search step significantly.

Figure 3.2 details the Ordering step. In step (1), STACKS and vertex ordering VS are initialized. In step (2), we choose a vertex $v_1$ of maximum degree. In step (3), all vertices adjacent to $v_1$ are pushed on STACKS according to their degree. In (4), the rest of the vertices in the current component are processed and added to the vertex ordering VS. STACKS are used to identify those vertices that have not been selected and to return an unselected vertex of maximum degree. In (5), we initialize ID to zero. In step (6), we follow the vertex ordering marking every vertex with its component ID. Also every component is marked cyclic or acyclic.

As discussed earlier, the start of a component is found by checking for $|K(v)| = 0$ where $v$ is the current vertex during traversal of the vertex ordering. For acyclic components, all vertices in the component must have $|K(v)| = 1$.

(1) initialize(STACKS)
initialize ordering sequence VS
$v_1 = a$ vertex of maximum degree
(2) mark $v_1$ SELECTED and add to VS list
(3) for each $w$ adjacent to $v_1$ do push($w$, STACKS[deg($w$)])
i = 2
while some vertex of nonzero degree is not SELECTED do
while STACKS are not empty do
$v_i = pop(STACKS)$
mark $v_i$ SELECTED and add to VS list
for $w$ adjacent to $v_i$ do
if $w$ is not SELECTED and $w$ is not in STACKS[deg($w$)] then
push($w$, STACKS[deg($w$)])
i = i + 1
endwhile
endwhile
(5) initialize ID to zero
(6) while not the end of the VS do /* follow the VS sequence */
/* All vertices in a component have the same component ID */
mark vertex $v_i$ with ID
if $|K(v_i)| > 1$ then
vertex[$v_i$.mark] = 1; Component[ID] = cyclic
else
vertex[$v_i$.mark] = 0; Component[ID] = acyclic
if $|K(v_i)| = 0$ then increment ID
VS = vertex[i].succ
Endwhile

3.3 The Searching Step

The Searching step takes the levels produced in the Ordering step and tries to assign hash values to indirect edges according to the ordering. Assigning hash values to $K(v_i)$ amounts to assigning a value to $g(v_i)$.

When a value is to be assigned to vertex[i].g, there are usually several choices for vertex[i].g that place all the indirect keys in $K(v_i)$ into “free choice” vertices. Once the “free choice” vertex The direct edges in these acyclic components are hashed to their desired locations as well. After hashing all indirect edges according to the vertex ordering, the rest of the direct edges in unassigned acyclic components are processed to finish the searching. In looking for a value for vertex[i].g, the Searching step uses a random probe sequence to access the slots 0, …, n-1 of the g table.

Figure 3.3 gives the algorithm for the Searching step. A random probe sequence of length n is chosen in step (3). At the beginning of the Searching step, the current implementation chooses a set of 20 small primes that do not divide n. Each time (3) is executed, one of the primes q is chosen at random to be $s_1$ and is used as an increment to obtain the remaining $s_j$, $j \geq 2$. Thus, the random probe sequence is 0, q, 2q, 3q, …, $(2r-1)q$. In step (4), direct edges are handled. In step (5), we compute h(k). Then in (6) we check if the indirect vertex address is cyclic or assigned. If so, we try another random probe value $s_i$. If the indirect vertex address is in an acyclic component and it is not assigned, then we store the key data as detailed in step (6). The Searching step fails if all random probe values result in collision.

(* edge[k].keydata is the key data concat (blockno, k prefix offset, portion of prefix) of the Key k *)
(1) for i [0 … n-1] do
vertex[i].assigned = false \n(2) for i = 1 to t (number of levels in VS) do
establish a random probe sequence $s_0, s_1, ..., s_{t-1}$ for [0 … n-1]
for each k in $K(v_i)$ do
j = 0
collision = false
if $v_i \in$ [0 … t-1] then w = 1 else w=2
for each k in $K(v_i)$ do...
if |K(v)| = 1 and vertex[edge[k].h].mark = 1 then
    a = [edge[k].h + vertex[edge[k].h].g] % n
    if edge[k].keydata ≥ a then
        vertex[vi].g = edge[k].keydata - a
    else vertex[vi].g = n - a + edge[k].keydata
else if component[vertex[h(k)]] is 2-cyclic
    a = [edge[k].h + vertex[edge[k].h].g] % n
    if edge[k].keydata ≥ a then
        vertex[vi].g = edge[k].keydata - a
    else vertex[vi].g = n - a + edge[k].keydata
    reorder 2-cyclic component in the tower
else
    if component[vertex[h(k)]] is 2-cyclic or assigned then
        collision = true
    else
        h(k) = [edge[k].h + vertex[edge[k].h].g + sj] % 2r
        if component[vertex[h(k)]] is cyclic or assigned then
            collision = true
        else
            j = j + 1
        if j > n-1 then fail

Figure 3.3: The Searching Step

4 Example

In this section, we give examples of our algorithm that illustrate the concepts discussed earlier.

Let S = {“JAN”, “FEB”, “MAR”, ……,”DEC”}

5 EXPERIMENTAL RESULTS

To provide further insights into our algorithm, we provide some experimental and timing statistics for various key set sizes and ratios, collected from random dependency graphs and their corresponding orderings. These statistics include the number of acyclic components Nacyclic, and the number of indirect edges Ninirect. The timing statistics include Mapping time, Ordering time, Searching time, and total time. Tables 4.1, 4.2 and 4.3 show these statistics for various key set sizes.

Table 4.1 OPMHF Timing Results, Key Set Size n=1024

<table>
<thead>
<tr>
<th>Ratio</th>
<th>Nacyclic</th>
<th>Ninirect</th>
<th>N2-cycle</th>
<th>Mapping (sec)</th>
<th>Ordering(sec)</th>
<th>Searching(sec)</th>
<th>Total(sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.4</td>
<td>463</td>
<td>108</td>
<td>2.63</td>
<td>0.63</td>
<td>0.93</td>
<td>4.19</td>
<td></td>
</tr>
<tr>
<td>1.3</td>
<td>364</td>
<td>113</td>
<td>2.60</td>
<td>0.63</td>
<td>0.98</td>
<td>4.21</td>
<td></td>
</tr>
<tr>
<td>1.2</td>
<td>280</td>
<td>152</td>
<td>2.63</td>
<td>0.57</td>
<td>1.10</td>
<td>3.30</td>
<td></td>
</tr>
<tr>
<td>1.18</td>
<td>282</td>
<td>196</td>
<td>2.58</td>
<td>0.58</td>
<td>1.23</td>
<td>4.39</td>
<td></td>
</tr>
<tr>
<td>1.16</td>
<td>272</td>
<td>216</td>
<td>2.60</td>
<td>0.58</td>
<td>3.97</td>
<td>6.75</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2 OPMHF Timing Results, Key Set Size n=10000

<table>
<thead>
<tr>
<th>Ratio</th>
<th>Nacyclic</th>
<th>Ninirect</th>
<th>N2-cycle</th>
<th>Mapping(sec)</th>
<th>Ordering(sec)</th>
<th>Searching(sec)</th>
<th>Total(sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.4</td>
<td>4355</td>
<td>710</td>
<td>6.02</td>
<td>6.00</td>
<td>5.60</td>
<td>17.62</td>
<td></td>
</tr>
<tr>
<td>1.3</td>
<td>3514</td>
<td>1027</td>
<td>5.98</td>
<td>5.73</td>
<td>5.72</td>
<td>17.43</td>
<td></td>
</tr>
<tr>
<td>1.2</td>
<td>2859</td>
<td>1706</td>
<td>5.95</td>
<td>5.38</td>
<td>7.27</td>
<td>18.60</td>
<td></td>
</tr>
<tr>
<td>1.18</td>
<td>2685</td>
<td>1754</td>
<td>5.98</td>
<td>5.35</td>
<td>8.25</td>
<td>19.58</td>
<td></td>
</tr>
<tr>
<td>1.16</td>
<td>2574</td>
<td>1936</td>
<td>5.97</td>
<td>5.33</td>
<td>10.35</td>
<td>21.65</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.3 OPMHF Timing Results, Key Set Size n = 130198

<table>
<thead>
<tr>
<th>Ratio</th>
<th>Nacyclic</th>
<th>Ninirect</th>
<th>N2-cycle</th>
<th>Mapping(sec)</th>
<th>Ordering(sec)</th>
<th>Searching(sec)</th>
<th>Total(sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.4</td>
<td>56649</td>
<td>9137</td>
<td>53.58</td>
<td>78.63</td>
<td>68.80</td>
<td>201.01</td>
<td></td>
</tr>
</tbody>
</table>
6 CONCLUSIONS

In this paper, a practical algorithm for finding augmented order-preserving minimal perfect hash functions is described. The method is able to find augmented OPMPHFs for various sizes of key sets in almost linear time with the function size remaining within reasonable bounds. The hash functions are optimal in terms of time (perfect) and hash table space utilization (minimal), and preserve the sorting of the key set. The application of the method to dictionary membership, inverted file storage, range searches is also illustrated. Furthermore, the resulting augmented perfect hash functions can be found in linear time and is 20% the space required to store functions generated by [7] methods that use acyclic random hyper graphs with c=3; a saving of five folds and is clearly more practical for very large collections.

7. REFERENCES