Technical Report of Order Preserving Minimal Perfect Hashing Function

Yingfeng Zhang

December 5, 2008

Abstract

This document presents the technical report for one component of *iZENELib* that could provide minimal perfect hashing with order- preserving. Order-preserving can guarantee the order of keys to be hashed, therefore a wildcard search ability could be provided based on it.

Contents

1 Document History

Date	Author	Description
2008-12-05	Yingfeng	Created.

2 Introduction

In order to better support wildcard search based on minimal perfect hashing, the ability of order-preserving should be provided. Order-preserving minimal perfect hash function (*OPMPHF*)means the hashed keys could preserve the order of input key set, to make clear what is implied, consider Figure1.

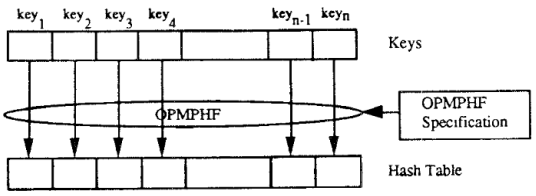

Figure 1: Order-preserving minimal perfect hash function.

3 Algorithm

Not so many algorithms have been published to illustrate how to implement *OPMPHF*, Czech[1] and Fox[2]'s work are the major of them, both of which are going to build the hash in memory at first, therefore a drawback of costing much memory does exist. Since searching process could be finished within linear time after bulding hash, and we can tolerate the cost for building, therefore it is reasonable to choose a relative simple method to build *OPMPH*. There are three ways to build *OPMPH* as summarized in [2]:

- Acyclic Graphs.
- Two-Level Hashing.
- Using Indirection.

Indirection is the approach chosen by Fox[2], while Czech[1] has implemented an improved acyclic graph, since acyclic graph is much simpler, that is the way we chose and is described as follows:

For a given undirected graph $G = (V, E)$,$|E| = m$,$|V| = n$,find a function $g : V \rightarrow [0, m-1]$ such that the function $h : E \rightarrow [0, m-1]$ defined as

$$h(e = (u, v)\epsilon E) = (g(u) + g(v))mod\ m \tag{1}$$

is a bijection. In other words we are looking for an assignment of values to vertices so that for each edge the sum of values associated with its endpoints taken modulo the number of edges is a unique integer in the range $[0, m-1]$.

If the graph G is acyclic, a very simple procedure can be used to find values for each vertex: Associate with each edge a unique number $h(e)\epsilon[0, m-1]$ in any order. For each connected component of G choose a vertex v. For this vertex, set $g(v)$ to 0. Traverse the graph using a depth-first search beginning with vertex v.If vertex w is reached from vertex u, and the value associated with the edge $e = (u, w)$ is $h(e)$, set $g(w)$ to $(h(e) - g(u))mod\ m$. Apply the above method to each component of G. To prove the correctness of the method it is sufficient to show that the value of function g is computed exactly once for each vertex. This property is clearly fulfilled if G is acyclic. The solution to this graph problem becomes the second part of the algorithm and is called the assignment step. Now we are ready to present the new algorithm for generating a minimal perfect hash function. We denote the length of the word w by $|w|$ and its i-th character by $w[i]$. The algorithm comprises two steps: mapping and assignment. In the mapping step a graph $G = (V, E)$ is constructed, where $V = 0, ..., n-1$ with n determined later, and $E = (f1(w), f2(w)) : w\epsilon W$. We introduce auxiliary functions f1 and f2 which are designed to be two independent random functions mapping W into $[0, n-1]$. There are various possibilities. Here we choose the functions to be:

$$f_1(w) = (\sum_{i=1}^{|w|} T_1(i, w[i]))mod\ n \tag{2}$$

$$f_2(w) = (\sum_{i=1}^{|w|} T_2(i, w[i]))mod\ n \tag{3}$$

where T_1 and T_2 are tables of random integers modulo n for each character and for each position of a character in a word. The space required by tables T_1 and T_2 is $O(logn)$ bits, since each entry is a number in the range $[0, n-1]$ and there is in effect a constant number of entries (actually dependent on the length of keys and the size of character set).As long as n fits into one computer word this is $O(1)$ words. If n is not less than the alphabet size, by treating each character $w[i]$ as a number we obtain another suitable pair of mapping functions:

$$f_k(w) = (\sum_{i=1}^{|w|} T_k(i, w[i]))mod\ n \tag{4}$$

These can be stored in less space at the expense of greater time for hash function evaluation on common machine architectures (since table lookups are replaced by multiplications). In fact we can

characterize suitable functions by as little as one random number, at the expense of even greater computation time. However our space requirements for increasing m are dominated by the space for storing the function g, so such considerations are of interest only for small m. Our goal is to find values of T_1 and T_2 so that the graph G is acyclic.Because we have no easy deterministic method for doing this, we randomly generate tables repeatedly, until we obtain an acyclic graph. Once an acyclic graph is generated the assignment step is executed. Notice that generating a minimal perfect hash function can be reduced to the problem described at the beginning of this section. For an acyclic graph, each edge $e = (u, v)\epsilon E$ corresponds uniquely to some word w (such that $f_1(w) = u$ and $f_2(w) = v$) so the search for the desired function is straightforward. We simply set $h(e = (f_1(w), f_2(w))) = i - 1$ if w is the i-th word of W . Then values of function g for each $v\epsilon V$ are computed by the assignment step. The function h is a minimal perfect hash function for W . Evaluation of the hash function is done in fast, constant time, involving little more than two standard hashes.

4 Design

As the component of *iZENELib*, *OPMPHF* is designed with generic structure, therefore the algorithm could be easily replaced. As shown in 2, an iterator is provided to support sequential access. Key-value pair storage layer is easily implemented because with minimal perfect hashing mechanism,

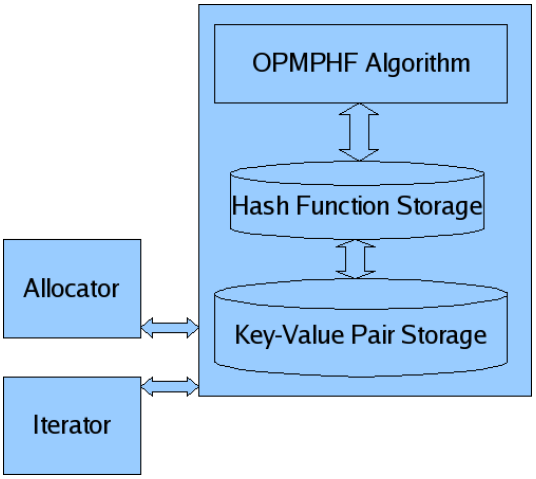

Figure 2: Design framework.

we could just store the key-value pair in a linear array. The role of the allocator is to make the storage easily managed, in future, if we have got a file based allocator, then the key-value pair could be stored on files easily.

5 Usage

```
////////////////////////////////////
#include <string>
#include "ph.h''
int main()
{
        ////////////////////////////////////
        ///choose std::string as the KeyType and ValueType
        PH<std::string,std::string,Opmph>* ph = new PH<std::string,std::string,Opmph>;
        //build_hash will call the algorithm to build OPMPHF, the input key set
        //should not contain repeated keys, or else the hash function can not be successfuly built.
        int ret = ph->build_hash(argv[1]);
        //Dump the hash function to a file for future use.
        ph->dump("ph.hash");
        delete ph;
        . . .
        ph = new PH<std::string,std::string,Opmph>;
        //Load the hash function before we proceed with the insert/search operation on the hash.
        ph->load("ph.hash");
        . . .
```

```
        //Adopt the iterator to access the hash sequentially.
        for(PH<std::string,std::string,Opmph>::const_iterator iter = ph->begin(); iter != ph->end(); ++iter)
        . . .
}
```

References

[1] Z. Czech, G. Havas, and B. Majewski. An Optimal Algorithm for Generating Minimal Perfect Hash Functions. *Information Processing Letters*, 43(5):257–264, 1992.

[2] E. FOX, Q. CHEN, A. DAOU13, and L. HEATH. Order-Preserving Minimal Perfect Hash Functions and Information Retrieval. *ACM Transactions on Information Systems*, 9(3), 1991.

6 Appendix

Table 1: Implement schedule

MileStone	Finish Date	In Charge	Description
Algorithm	2008-12-2	Yingfeng	Make the *OPMPHF* algorithm work
Wrapper	2008-12-3	Vernkin	Provide a wrapper for different minimal perfect hashing algorithm
Initial version	2008-12-5	Yingfeng	Initial version can be used
Better wrapper	?	Yingfeng	Wrap to keep consistent with *iZENELib*