

Perfect Hash Functions for Large Web Repositories

Amjad M. Daoud *

Abstract. *We describe a new practical algorithm for finding perfect hash functions with no specification space at all, suitable for key sets ranging in size from small to very large. The method is able to find perfect hash functions for various sizes of key sets in linear time. The perfect hash functions produced are optimal in terms of time (perfect) and require at most computation of $h_1(k)$ and $h_2(k)$; two simple auxiliary pseudorandom functions.*

1. Introduction

There is a growing demand for fast access to large text collections such as repositories of Web pages. For a repository to be searched fast, it requires an index. To construct an index, the set of unique words and URLs need to be identified. The dictionary constructed may not fit in main memory and may need to be stored on slow external media. Hashing has long been used when the fastest possible direct access to random locations is desired. More recently, a range of linear algorithms for producing quality order preserving and minimal perfect hashing algorithms for static sets were introduced [11] [8] [13] [14]. The specifications of the minimal perfect hash functions require $O(n)$ words where n is the number of keys. For example, The WebGraph research group [1] used the Czech, et. al. algorithms [8] to access 18.5 million URLs. The perfect hash function specification comprised of a g table required 88 MB to hash the UK web graph. For a world wide web snapshot of 118 million URLs, the g table size is 563 MB mostly random numbers that can not be compressed any further. Clearly, the PHF specification for large sets cannot be stored in the L2 cache and computing the final hash address would require three expensive random lookups to the g table. Random accesses even to main memory often take as much time as executing hundreds of instructions on modern CPUs.

In this paper, we present a new faster perfect hashing approach that maps the key set into a bipartite graph using two functions $(h_1(k), h_2(k))$, arranges the vertices of the graph according to incident edges cardinality in ascending order, and assigns the keys to their final locations simply according to $h_1(k)$ or $h_2(k)$. Ideally the graph would be compact and acyclic. The algorithm produces perfect hash functions that require *no specification space at all*; however, functions produced are only perfect; not minimal and not order preserving. Our algorithm improves upon earlier schemes in that it uses the bipartite graph approach to avoid degenerate edges problems and so that two functions are used. Compact acyclic bipartite graphs are harder to construct but produce perfect hashing schemes that require at most two accesses. Nevertheless, our approach relaxes the acyclic requirement in random graphs presented in [8] and can tolerate the presence of the most common cyclic components; and thus is easier to construct. The algorithm produces perfect hash functions with much higher success rates than the acyclic hypergraph approach [8] and mostly from the first trial. Our algorithms have been tested in the MG4J system [1] to accumulate URLs collected and are currently used to store efficiently large web maps [11].

*Department of Computer Science, University Of Petra, Amman, Jordan email: amjad@uop.edu.jo

1.1. Motivation

This work was in part motivated by our investigations that deal with tightly integrating information retrieval with relational databases [12] and construction of efficient web maps for search engines [11]. We investigated two popular methods for producing order preserving and minimal perfect hashing algorithms as described in [13] and [8]. We found that they have large main memory requirements or high probability of failure due to degenerate edges (one out of 10 trials succeeds as evident from the MG system and MG4J implementations). Both algorithms require substantial storage for their hash functions parameters and are inefficient when memory is scarce.

1.2. Applications

There are numerous applications for our algorithm in digital libraries and information retrieval systems: some of them are novel such as web maps; the others are well known such as the dictionary membership [13] [11]. Efficient perfect hash functions have been successfully used to manage very large web graphs. Storing snapshots of the Web helps the tuning of ranking algorithms such as Google PageRank [3] algorithm. The WebGraph research group [1] used order preserving perfect hashing [8] to access URL sets as large as 18.5 million URLs and the perfect hash function specifications required 88 MB.

2. Previous Methods to Find PHFs

Hashing has been a topic of study for many years, both in regard to practical methods and analytical investigations [17]. A less extensive literature has grown up, mostly during the last decade, dealing with perfect hash functions; it is that subarea that we consider in this section.

Given a key k from a static key set S of cardinality n , selected from a universe of keys U with cardinality N , our objective is to find a function h_f that maps each k to a distinct entry in the hash table T containing m slots. If the $ratio_T m/n = 1$, h is said to be minimal.

For a function h to be perfect, it must map each key k in S to a unique integer bounded by m using $O(1)$ operations. Early algorithms by Sprugnoli [22], Jaeschke [16], and Fredman, Komlós, and Szemerédi [15]. Chang [5] used four tables based on the first and second letters of the key. Cichelli [6] used the length of the key and tables based on the first and last letters of the key. Note, however, that the length of a key, its first letter, and its last letter are sometimes insufficient to avoid collisions; consider the case of the words ‘woman’ and ‘women’ in Cichelli’s method.

Cercone et al. [4] enhanced the discriminating power of transformations from strings to integers by generating a number of letter to number tables, one for each letter position. Clearly, if the original keys are distinct, numbers formed by concatenating fixed length integers obtained from these conversion tables will be unique. In practice, it often suffices to simply form the sum or product of the sequence of integers.

While in some schemes (e.g., [6]) the resulting integer is actually the hash address desired, in most algorithms, the h function must further map from the integer value produced into the hash table.

Brain and Tharp [2] presented a new approach to MPH hashing. Their scheme first maps the keys onto a 2-dimension relatively sparse array and then compacts the array onto a one-dimension array.

Author	Time Efficiency	Space Efficiency	Minimal	Calc Efficiency
Sprugnoli	$O(2^n)$	$O(n \log n)$	no	$O(1)$
Jaeschke	$O(2^n)$	$O(n \log n)$	no	$O(1)$
Cichelli	$O(2^n)$	$O(n \log n)$	no	$O(1)$
Brain	$O(2^n)$	$O(n \log n)$	no	$O(1)$
Chang	$O(n^2 \log n)$	$O(n \log n)$	no	$O(n \log n)$
Sager	$O(n^4)$	$O(n \log n)$	yes	$O(1)$
Fredman	$O(n)$	$O(n \log n)$	no	$O(1)$
Cormack	$O(n)$	$O(n \log n)$	yes	$O(1)$
Daoud90	$O(n^{1+\eta})$	$O(\frac{n}{\eta \ln 2})$	yes	$O(1)$
Schmidt	$O(n)$	$O(n)$	yes	$O(1)$
Czech	$O(n)$	$O(n \log n)$	yes	$O(1)$
current	$O(n)$	$O(1)$	no	$O(1)$

Table 1. Comparison of Different Perfect Hashing Algorithms

If compaction could be made collision free, the indices of the resulting one-dimension array can be used as hashing addresses. Their algorithm can generate PHFs for up to 5, 000 keys.

Schmidt and Siegel [21] presented tight bounds on the spatial complexity of perfect hash functions. They described a variation of an explicit $O(1)$ time single probe perfect hash function that can be specified in $O(n + \log \log m)$ bits.

In [14], Fox et. al. presented two algorithms to find MPHFs, both based on the notion of Mapping-Ordering-Searching (MOS). In general, MOS approach calls for mapping keys in a particular key set into a space of representative of keys, ordering the subsets of representatives (of keys) and finally searching the MPHf specification space for each representative subset so that corresponding keys fit onto the hash table. The space used to specify the MPHf is related to all three steps. Ideally, the representative space should help ordering step to produce a proper ordering so that searching step could easily fit the keys. Since identifying an optimal ordering is NP-complete [20], heuristics is usually sought to obtain suboptimal orderings in practice.

In [13],[11], and [10] we presented algorithms that make use of the dependency graph where vertices are the range of h_1 and h_2 functions (generated in mapping step) and edges are keys. The ordering step heuristically arranges the vertices in the dependency graph to get a vertex sequence. The subsets of keys induced by the sequence are handled according to their size in ascending order and such that subsets of size one (the majority) are assigned in the hash table during the searching step first. These algorithms are capable of producing order preserving minimal PHFs for very large sets while keeping the specification space close to theoretical bounds. The ordering heuristics may fit large subsets in the table later than the smaller ones. This increases the difficulty of fitting the table when the number of subsets is small. Another drawback of the algorithm is that each vertex contributes $\log n$ bits to the MPHf specification. For vertices associated with small key subsets appearing earlier in the sequence, this is a large overhead.

In Table 1, we compare the different perfect hashing algorithms.

In the next section, we extend these algorithms to build dependency graphs that are more sparse and

easier to work with (i.e. acyclic). We propose a new ordering algorithm that allows as many keys as possible to be assigned during the search step in parallel.

3. The new algorithm

In this section, we introduce a new algorithm to generate perfect hash functions that require no specification space at all and has a high success rate.

The new algorithm builds dependency graphs that are more sparse and easier to work with (i.e. acyclic). Motivated by the fact that an acyclic dependency graph is a perfect mapping of the key set to an array of size m , the two random functions used to build dependency graphs $h_1(k)$ and $h_2(k)$ are used to generate PHFs that require no specification space at all with high success rates and in linear time. The final hash function is simply $h_1(k)$ or $h_2(k)$. So to find k we would check the two T table entries: $T[h_1(k)]$ or $T[h_2(k)]$.

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 l_1, \dots, l_t , then the level of keys $K(l_i)$, $1 \leq i \leq t$, corresponding to a set of vertices v_i , $1 \leq i \leq 2r - 1$, is the set of edges incident both to v_i and to a vertex earlier in the ordering. The first level $K(l_1)$ is the set of edges incident to each vertex v_i , $1 \leq i \leq 2r - 1$, in the dependency graph chosen such that the vertex has only one incident edge or has an edge that would break a cycle involving v_i . If a component is cyclic and has more than one cycle, the ordering step fails and we try a different mapping again .

Clearly, our hashing scheme allows us to have cycles of size two in the dependency graph. In section 5., we show that cyclic components can have cycles of size two only, since the probability of having vertices with more than two incident edges drops to zero as n approaches 0 as $m = 2r \rightarrow \infty$ [8]. Consequently, a vertex cannot have two cycles of size two. Clearly, our algorithm has a strong mathematical background. In fact, relaxing the acyclic requirement in random bipartite graphs helps lower m and increases the success rates of our algorithm.

The algorithm 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 takes a set of n keys and produces the two auxiliary hash functions h_1 , and h_2 . The h_1 , and h_2 values are used to build a bipartite graph called the *dependency graph*. Half of the vertices of the dependency graph correspond to the h_1 values and are labeled $0, \dots, r - 1$. The other half of the vertices correspond to the h_2 values and are labeled $r, \dots, 2r - 1$. There is one edge in the dependency graph for each key in the original set of keys. A key k corresponds to an edge labeled k between the vertex labeled $h_1(k)$ and the vertex labeled $h_2(k)$. Notice that there may be other edges between $h_1(k)$ and $h_2(k)$, but those edges are labeled with keys other than k . There are two data structures that constitute the dependency graph, one for the edges (keys) and one for the vertices (determined by the h_1 and h_2 values). Both are implemented as arrays. The vertex array is

```
vertex:  array [0..2r-1] of record
         firstedge: integer;
```

- (1) build random tables for h_1 , and h_2
- (2) **for each** $v \in [0 \dots 2r - 1]$ **do**
 $\text{vertex}[v].\text{firstedge} = 0$
 $\text{vertex}[v].\text{degree} = 0$
- (3) **for each** $i \in [1 \dots n]$ **do**
 $\text{edge}[i].h_1 = h_1(k_i)$
 $\text{edge}[i].h_2 = h_2(k_i)$
 $\text{edge}[i].\text{nextedge}_1 = 0$
 add $\text{edge}[i]$ to linked list with header
 $\text{vertex}[h_1(k_i)].\text{firstedge}$
 increment $\text{vertex}[h_1(k_i)].\text{degree}$
 $\text{edge}[i].\text{nextedge}_2 = 0$
 add $\text{edge}[i]$ to linked list with header
 $\text{vertex}[h_2(k_i)].\text{firstedge}$
 increment $\text{vertex}[h_2(k_i)].\text{degree}$

Figure 1. The Mapping Step

```

degree:  integer;
end

```

`firstedge` is the header for a singly-linked list of the edges incident to the vertex. `degree` is the number of edges incident to the vertex. The edge array is

```

edge:  array [1..n] of record
    h1, h2: integer;
    nextedge1: integer;
    nextedge2: integer;
end

```

h_1 , and h_2 contain the h_1 , and h_2 values for the edge (key). Also, nextedge_i , for side i ($= 1, 2$) of the graph (corresponding to h_1, h_2 , respectively), points to the next edge in the linked list whose head is given by `firstedge` in the vertex array.

Figure 1 details the Mapping step. Let k_1, k_2, \dots, k_n be the set of keys. The h_1 , and h_2 functions are selected (1) as the result of building tables of random numbers. The construction of the dependency graph in (2) and (3) is straightforward. Therefore, the expected time for the Mapping step is $O(n)$.

3.2. The Ordering Step

The Ordering step explores the dependency graph so as to partition the set of keys into a sequence of levels. The step actually produces an ordering of the vertices in levels having a degree of one when all preceding levels are processed.

```

(1) initialize (STACKS)
    initialize ordering sequence VS
(2) select all vertices of degree = 1
    mark them as SELECTED, and add them to  $K(l_1)$ 
(3) for each  $w$  adjacent to vertices in  $K(l_1)$  do
    push ( $w$ , STACKS [ $\text{deg}(w)$ ])
     $i = 2$ 
(4) while some  $v$  is not SELECTED do
    while STACKS are not empty do
         $v = \text{popmin}(\text{STACKS})$ 
        if  $v$  has one edge left to process then
            mark  $v$  SELECTED
            add  $v$  to level  $i$  in VS list
        for  $w$  adjacent to  $v$  do
            if  $w$  is not SELECTED and
                 $w$  is not in STACKS [ $\text{deg}(w)$ ] then
                    push ( $w$ , STACKS [ $\text{deg}(w)$ ])
             $i = i + 1$ 
        endwhile
    endwhile
endwhile

```

Figure 2. The Ordering Step

Since the vertex degree distribution is decidedly skewed and the graph is relatively sparse, the first level would contain more than 70% of the keys. All these keys can be assigned in parallel in the search step. Next we visit vertices that are of minimum degree as they are likely to have only one unprocessed edge left. The algorithm uses a sufficient number of stacks to identify the next minimum degree processed and the unprocessed vertices. These stacks accelerate choosing the next vertex with the required degree. Figure 2 details the Ordering step. In step (1), STACKS and vertex ordering VS are initialized. In step (2), we choose all vertices v of degree = 1 and add them to $K(l_1)$. In step (3), all vertices adjacent to vertices in $K(l_1)$ 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 minimum degree.

Clearly the Ordering step can be finished in $O(n)$ time because we traverse the vertices in the dependency graph only once.

3.3. The Searching Step

The Searching step takes the levels produced in the Ordering step and tries to assign hash values to edges according to the ordering. Assigning hash values to vertices that have degree of one or one key k in $K(l_i)$ amounts to assigning $T[h_1(k)]$ to the key k if the vertex is on the h_1 side and assigning $T[h_2(k)]$ to the key k if the vertex is on the h_2 side. Recall that the hash table T has the same size as the dependency graph and that $m = 2r$.

```

(1) for  $i = 1$  to  $t$  (number of levels in  $VS$ ) do
    if  $v_j.degree$  in  $K(l_i) = 1$  then
        for each  $v \in K(l_i)$  do
             $T_j = k_{v_j}$ 
            remove  $k_{v_j}$ 
    else
        fail

```

Figure 3. The Searching Step

Figure 3 gives the algorithm for the Searching step. Clearly the Searching step requires only $O(n)$ time to finish.

4. Example

We show an example of finding a PHF for the 20 key set listed in Table 2. The ratio used is 1.2 and thus $r = 12$ and the size of the hash table $T = 24$. The h_1 , and h_2 values (see Table 2) are used to build a bipartite graph. Half of the vertices of the dependency graph correspond to the h_1 values and are labeled $0, \dots, r - 1 = 11$. The other half of the vertices correspond to the h_2 values and are labeled $r, \dots, 2r - 1 = 23$. There is one edge in the dependency graph for each key in the original set of keys.

We notice that we have four acyclic components (trees) and Table 3 verifies that we indeed have a PHF for the example key set.

The tower levels dictates the order of assignments: vertices with $degree = 1$ are $\{v_0, v_2, v_3, v_5, v_8, v_{10}, v_{11}\}$ on the h_1 side and $\{v_{12}, v_{13}, v_{14}, v_{15}, v_{16}, v_{18}, v_{19}, v_{22}\}$ on the h_2 side. So $K(l_1) = \{v_0, v_2, v_3, v_5, v_8, v_{10}, v_{11}, v_{12}, v_{13}, v_{14}, v_{15}, v_{16}, v_{18}, v_{19}, v_{22}\}$ has the following set of keys: “tussle”, “oculomotor nerve”, “deposited”, “Han Cities”, “Chungking”, “meridiem”, “Lagomorpha”, “antennae”, “sodium lamp”, “Euclidean”, “quibbles”, “ethyl ether”, and “x-rays” that can be assigned to their final addresses according to h_1 or h_2 without collision. Here, we favor placing keys according to their $h_1(k)$ value since we would compute $h_1(k)$ first when searching for k . In our example, the l_1 can be visualized as the edges that are incident on the nodes of the first level. Notice that we have an acyclic bipartite graph which is equivalent to four random trees. In fact, the algorithm assigns the edges incident on their leaf nodes first (each has only one incident node). Next, moves to the next level till all edges (keys) are assigned to unique vertices.

When all assigned keys are removed; $K(l_2) = \{v_1, v_6, v_7\}$, has the set of keys “treacherous”, “calc-”, “dentifrice”; $K(l_3) = \{v_{20}, v_{23}\}$ has two keys {“imprecise”, “Clouet”}, and finally $K(l_4) = \{v_4\}$ has two keys {“Bulwer”, “rotundus”},

5. Analysis

We bound the expected total length of cycles in the graph. Let C_{2k} denote the number of cycles of length $2k$. To build a cycle of length $2k$, we select $2k$ vertices and connect them with $2k$ edges in

Key	h_1	h_2
x-rays	7	22
Euclidean	1	14
ethyl ether	6	19
Clouet	4	23
Bulwer	4	17
dentifrice	7	20
Lagomorpha	11	17
Chungking	8	15
quibbles	7	18
Han Cities	5	16
treacherous	1	23
calc-	6	23
deposited	3	23
rotundus	9	17
antennae	1	12
sodium lamp	7	13
oculomotor nerve	2	21
tussle	0	20
imprecise	9	20
meridiem	10	21

Table 2. The Key Set Used in Example, $n=20$, $m=24$, ratio = 1.2

key	h_1	h_2	h_{final}	$T[h_{final}]$
x-rays	7	22	h_2	T[22]
Euclidean	1	14	h_2	T[14]
ethyl ether	6	19	h_2	T[19]
Clouet	4	23	h_2	T[23]
Bulwer	4	17	h_1	T[04]
dentifrice	7	20	h_1	T[07]
Lagomorpha	11	17	h_1	T[11]
Chungking	8	15	h_1	T[08]
quibbles	7	18	h_2	T[18]
Han Cities	5	16	h_1	T[05]
treacherous	1	23	h_1	T[01]
calc-	6	23	h_1	T[06]
deposited	3	23	h_1	T[03]
rotundus	9	17	h_1	T[09]
antenna	1	12	h_2	T[12]
sodium lamp	7	13	h_2	T[13]
oculomotor nerve	2	21	h_1	T[02]
tussle	0	20	h_1	T[00]
imprecise	9	20	h_2	T[20]
meridiem	10	21	h_1	T[10]

Table 3. Keys and Their Final Hash Addresses, $n=20$, $m=24$, ratio = 1.2

any order. There are $\binom{r}{k}^2$ ways to choose $2k$ vertices out of $2r$ vertices of a graph, $k!k!/2k$ ways of connecting them into a cycle, and $(2k)!$ possible ordering of the edges. The cycle can be embedded into the structure of the graph in $\binom{r}{r-2k}r^{2(r-2k)}$ ways. Hence, the number of graphs containing a cycle of length $2k$ is

$$\binom{r}{k}^2 ((k!)^2/2k)(2k)! \binom{r}{r-2k} r^{2(r-2k)}$$

Also, the expected number of cycles of length $2k$ in the graph is

$$\sum_{k=1}^{n/2} 2k E(C_{2k}) = \sum_{k=1}^{n/2} \frac{\binom{r}{k}^2 ((k!)^2/2k)(2k)! \binom{r}{r-2k} r^{2(r-2k)}}{n^{2k}}$$

And the expected number of cycles in the graph c is

$$\sum_{k=1}^{n/2} E(C_{2k}) = \sum_{k=1}^{n/2} \frac{\binom{r}{k}^2 ((k!)^2/2k)(2k)! \binom{r}{r-2k} r^{2(r-2k)}}{n^{2k}}$$

According to [8], the number of cycles is bounded by

$$\sum_{k=1}^{n/2} E(C_{2k}) < \ln(2k)$$

Next, we count the number of tree components in G , excluding zero-degree vertex components. We have the number of different trees in a bipartite graph G' :

$$R_{ij} = j^{i-1} \cdot i^{j-1}$$

Let t be expected number of trees of distinct edges of size from 1 to n in a bipartite graph G with r vertices on each side is

$$E(\text{TR}) \leq \sum_i \sum_j \frac{\binom{r}{i} \binom{r}{j} \cdot R_{ij} \cdot \binom{n}{n} \cdot (n)!}{r^{2n}}$$

where i and j should satisfy the constraints $n-i \geq 1$ and $n-j \geq 1$ when $i+j-1 < n$, or $n-i \geq 0$ and $n-j \geq 0$ when $i+j-1 = n$.

Moreover, the maximum number of components that have one cycle is bounded by $2r-t$ and the probability of having a cyclic component with more than one cycle approaches 0 as $m = 2r \rightarrow \infty$. This is a direct result from [8] Lemma 4.3. Consequently, and almost surely a vertex cannot participate in two different cycles of size two or higher.

	$ratio_T=1.2$			
Keys	Map	Order	Search	Total
1M	0.55	0.05	0.04	0.64
2M	1.05	0.09	0.06	1.20
4M	2.12	0.20	0.11	2.43
8M	5.85	0.37	0.19	6.21
19.5M	10.40	0.72	0.40	11.52
	$ratio_T=1.4$			
Keys	Map	Order	Search	Total
1M	0.56	0.04	0.03	0.63
2M	1.20	0.08	0.05	1.33
4M	2.32	0.17	0.11	2.60
8M	5.05	0.33	0.23	5.61
19.5M	11.90	0.60	0.45	11.95

Note: Machine = Sony Vaio P4-3200MHz;
Time (CPU) is in seconds.

Table 4. Running Time Summary of the New Algorithm

6. Experimental Results

In this section, we present the running results of the new algorithm on a Sony VAIO workstation equipped with 3.2 GHz Pentium 4, a 3Ware 8-port 8506-8 SATA RAID controller, and four Western Digital 250gb SATA drives (8MB cache) in RAID0 configuration. The system runs the Linux OS with the 2.6 kernel. The xfs filesystem is used due to its prefetching of sequential files. The system has 4Gig of dual DDRAM, with 1GB dedicated to filesystem cache. Table 4 describes the performance of the algorithm on different key sets for different ratios $ratio_T = 1.2$, $ratio_T = 1.35$, and $ratio_T = 1.5$.

Table 5 shows the time for a large set 19.5M keyset that consist of 18.5M URLs taken from [1] and 1M keywords when $ratio_T$, is varied from 1.2 to 1.5. It can be seen that mapping time is the dominant factor. The mapping step can actually skip all the keys that map to isolated vertices as those are assigned to the first level during the ordering step. The first level nodes need not be ordered. So we can skip them during the mapping step; and this improves the performance of the mapping step significantly over other schemes. Similar optimization can be applied to algorithm that process acyclic hypergraphs described in [8].

7. Conclusion

This paper describes a new practical algorithm for finding perfect hash functions with no specification space at all, suitable for key sets ranging in size from small to very large. The method is able to find PHFs for various sizes of key sets in linear time. The hash functions are optimal in terms of time (perfect) and requires at most computation of $h_1(k)$ and $h_2(k)$. Moreover, to help access methods researchers understand the algorithm, we have dedicated a WWW page to our algorithms and provided Java applets to visualize how they work [23].

$ratio_T$	Mapping	Ordering	Searching	Total
1.2	10.40	0.72	0.40	11.52
1.3	10.95	0.66	0.47	12.08
1.4	11.90	0.60	0.45	11.95
1.5	12.31	0.55	0.44	13.25

Note: Machine = Sony Vaio P4-3200MHz;
Time (CPU) is in seconds.

Table 5. Running Time Summary of the New Algorithm on a UK Web Graph: 19.5M Key Set of URLs

References

- [1] Boldi P., and Vigna S. The WebGraph framework I: Compression techniques, *Proc. of the Thirteenth International World Wide Web Conference, Manhattan, USA* (2004).
- [2] Brain, M.D., and Tharp, A.L. Perfect hashing using sparse matrix packing. *Information Systems 15* (1990), 281-290.
- [3] Brin, S. and Page, L. <http://www-db.stanford.edu/backrub/google.html>, (2002).
- [4] Cercone, N., Krause, M., and Boates, J. Minimal and almost minimal perfect hash function search with application to natural language lexicon design, *Computers and Mathematics with Applications 9* (1983), 215-231.
- [5] Chang, C.C. The study of an ordered minimal perfect hashing scheme, *Communications of the ACM 27* (1984), 384-387.
- [6] Cichelli, R.J. Minimal perfect hash functions made simple, *Communications of the ACM 23* (1980), 17-19.
- [7] Cormack, G.V., Horspool, R.N.S., and Kaiserswerth, M. Practical perfect hashing, *The Computer Journal 28* (1985), 54-58.
- [8] Czech, Z.J., Havas G. and Majewski, B.S. Perfect hashing, *Theoretical Computer Science*, Vol. 182 (1997) 1-143.
- [9] Daoud, A.M. Efficient Data Structures for Information Retrieval, PH.D. dissertation, Department of Computer Science, Virginia Polytechnic Institute & State University (1993).
- [10] Daoud, A.M. Efficient Data Structures for Search Engines, Technical Report TR-2005-3, DiTech (2005).
- [11] Daoud, A.M. Augmented Order Preserving Perfect Hashing for Information Retrieval, *to appear* (2004).
- [12] DeFazio, S., Daoud, A.M., Smith, L., Srinivasan, J., Croft, W.B. and Callan, J. Integrating IR and RDBMS using cooperative indexing, *SIGIR* (1995) 94-92.
- [13] Fox, E.A., Chen, Q., Daoud, A.M. and Heath, L. Order preserving minimal perfect hash functions and information retrieval, *SIGIR* (1990) 279-311.

- [14] Fox, E.A., Heath, L., Chen, Q., and Daoud, A.M. Practical minimal perfect hash functions for large databases, *Communications of the ACM* (1992).
- [15] Fredman, M.L., Komlós, J. and Szemerédi, E. Storing a sparse table with $O(1)$ worst case access time, *Journal of the ACM* 31 (1984), 538-544.
- [16] Jaeschke, G. Reciprocal hashing—a method for generating minimal perfect hash functions. *Communications of the ACM* 24 (1981), 829-833.
- [17] Knuth, D.E. *The Art of Computer Programming*, Volume 3, *Sorting and Searching*, Addison-Wesley Publishing Company, Reading, MA, 1973.
- [18] Raghavan S., and Garcia-Molina, H. Representing Web Graphs, *Stanford University* (2002).
- [19] Zobel, J., Heinz, S., Williams, H.E. In-memory Hash Tables for Accumulating Text Vocabularies, *Information Processing Letters*, 80(6), (2001) 271-277.
- [20] Sager, T.J. A polynomial time generator for minimal perfect hash functions, *Communications of the ACM* 28 (1985), 523-532.
- [21] Schmidt, J.P., and Siegel, A. On aspects of universality and performance for closed hashing. *Proceedings of the 21st ACM Symposium on Theory of Computing*, 1989, 355-366.
- [22] Sprugnoli, R. Perfect hashing functions: a single probe retrieving method for static sets, *Communications of the ACM* 20 (1978), 841-850.
- [23] <http://www.omlet.org/phf/phf2005.html>