A More Cost Effective Algorithm for Finding Perfect Hash Functions

Edward A. Fox, Qi-Fan Chen, Lenwood Heath and Sanjeev Datta

TR 88-30

September 1988

A More Cost Effective Algorithm for Finding Perfect Hash Functions

Edward A. Fox
Qi-Fan Chen
Lenwood Heath
Sanjeev Datta

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, VA 24061-0106 USA

Electronic mail to:
fox%fox.cs.vt.edu@dcssvx.cc.vt.edu

ABSTRACT

As the use of knowledge-based systems increases, there will be a growing need for efficient artificial intelligence systems and methods to access large lexicons. In the COmposite Document Expert/extended/effective Retrieval (CODER) system we have, in order to provide rapid access to data items on CD-ROMs and to terms in a lexicon built from machine readable dictionaries, investigated the construction of perfect hashing functions. We have considered algorithms reported earlier in the literature, have made numerous enhancements to them, have developed new algorithms, and here report on some of our results. This paper covers an $O(n^3)$ algorithm that has been applied to building hashing functions for a collection of 69806 words on a CD-ROM. Most recently we have developed a much better algorithm and have succeeded in finding a perfect hash function for a set of 5000 words taken from the *Collins English Dictionary*.

CR Categories and Subject Descriptors: E.2 [**Data Storage Representations**]: *hash-table representations;* H.2.2 [**Database Management**]: Physical Design - *access methods;* H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing - *indexing methods;* H.3.2 [**Information Storage and Retrieval**]: Information Storage - *file organization*

General Terms: Algorithms, Experimentation

Additional Keywords and Phrases: perfect hashing, random graph, minimal perfect hashing, indexing

1. Introduction

In the COmposite Document Expert/extended/effective Retrieval (CODER) system we have investigated the construction of a large lexicon from machine readable dictionaries [FOXE86], and we are also developing methods to access large databases on CD-ROMs. Furthermore, we are producing such CD-ROMs with suitable access software [FOXE88]. In all of these contexts, there are large static collections of records which must be indexed by keys, usually English words or phrases. Hashing is one method that can provide the desired rapid access, with little overhead in space, but unless the hash function chosen is suitable, there can be a considerable loss in performance due to collisions. With static data sets, however, it is possible to build so-called "perfect" hashing functions that require minimal space for the hash table, and which avoid the problem of collisions.

There are mainly two ways to obtain perfect hash functions that have been considered. The first type of approach involves directly searching for a proper function — i.e., using a "search-only" strategy [SPRU78], [JAES81], [CHAN86]. A class of functions is often chosen, and constants are sought such that hashing can take place without collisions. While in some of these approaches it can be shown that a perfect hash function will always be found, it is usually the case that this direct search is prohibitively expensive (e.g., takes time related exponentially to the number of distinct keys) [JAES81]. The method is clearly only practical for very small data sets. Thus, another class of approaches has been explored.

The second way to obtain perfect hash functions involves "mapping-ordering-searching" methods [CICH86], [CERC83] and [SAGE85]. To begin with, the key space is mapped to a new space where an ordering heuristic can be applied to reduce the cost of the search. Then a much less expensive searching methods can be applied. While this general approach seemed to allow construction of perfect hashing functions for much large data sets than does the search-only approach, there were no reported results of building suitable functions for very large data sets.

In our on-going knowledge base project, fast access to a very large static lexicon data base of size on the order of at least 10,000 is required, and methods to handle 100,000 or more items are desired. An effort was made to compare various perfect hashing methods [DATT88] and it was concluded that Sager's algorithm [SAGE85] might be a good candidate, but its efficiency on large data sets was not adequate. In the following sections, a more efficient algorithm for finding a perfect hash function is outlined along with some experimental results. To aid comparison, an overview of Sager's algorithm and some test results using both methods are presented as well.

2. Terminology

Key Space: Each record in the data file has a key field, and the set of keys in the file forms the key space. We use W to denote the key space and M to denote the cardinality of W.

Hash Table: The hash table stores virtually the real addresses for records given their keys. The size of the table is denoted by N.

Load factor of a hash table: $LF = M / N$.

Bipartite Graph: A bipartite graph G is a graph where the vertex set R can be divided into two disjoint subsets R_1 and R_2 such that every edge in G has one of its vertices in R_1 and the other in R_2. We define $R = R_1 \cup R_2$, and by our construction of R have $|R|$ even and $r = |R| / 2$.

To reduce a graph: To reduce a graph G is to take an edge $e=(n_1,n_2)$ of G and collapse two vertexes n_1 and n_2 of G to a single new vertex n'. After reduction, e disappears, and all edges adjacent to n_1 or n_2 connect to n'.

Perfect hash function: A perfect hash function assigns each key to an unique slot in the hash table. A minimal perfect hash function has LF equal one.

3. Sager's algorithm and test results

3.1 Outline of Sager's algorithm

Since Sager's algorithm [SAGE85] is a mapping-ordering-searching algorithm, we present it based on its mapping, ordering, and searching stages.

3.1.1 Mapping stage

In this stage, the set W of N words is mapped to three sets by three independent functions h0, h1 and h2 which will hopefully each lead to a uniform distribution of result values for a given input set. The functions recommended by Sager are:

h0: W -> I (a finite set of integers defined only by h0 and the data);
h1: W -> R_1, $R_1 = \{ x \mid x \in [0 .. r\text{-}1] \}$ so $|R_1| = r$;
h2: W -> R_2, $R_2 = \{ x \mid x \in [r .. 2r\text{-}1] \}$ so $|R_2| = r$.

Each word is associated with the tuple (h0(w), h1(w), h2(w)). No two words may be associated with the same tuple; if such a collison occurs, new h0, h1, and h2 functions must be selected. A bipartite graph is formed with vertices labelled with each of the values in the range of the functions h1 and h2; an edge for each word w connects the nodes labelled h1(w) and h2(w). Note that it is possible for two vertices to have multiple edges (words) between them.

3.1.2 Ordering stage

In this stage, an ordering $w_0, w_1, ..., w_k$ on words in W that totally determines the search sequence is found. The ordering is based upon a simple heuristic: always select next an edge in the bipartite graph that is in a maximal number of cycles of minimal length. One such selected edge is called the **canonical edge.** Any remaining edges that have the same endpoints as the canonical edge are also selected; all the selected edges constitute the next **level** of the search sequence. The strategy intends to produce a search sequence, called the **word tower**, where the levels that contain many edges occur early in the search.

In Sager's algorithm, the selecting procedure is done by reduction on the bipartite graph. The procedure is repeated until the graph becomes empty. This stage requires $O(r^4)$ time.

3.1.3 Searching stage

Searching starts by systematically assigning U values to the canonical edges. This follows the order from the previous stage. At the same time, final hash addresses are computed for all edges in that level of the word tower. In the case of a collision, another value for the canonical edge (i.e., one more than the previous value) is tested. If all possible values fail (i.e., we attempt N tries), the algorithm backtracks to the previous level of the tower to try assigning new values there.

If all canonical edges are successfully assigned values then a perfect hash function, g, is constructed by a final simple scan of the graph. (Further details can be found in [SAGE85] and [DATT88].)

3.2 Experimental results and discussion

A careful study of Sager's algorithm using our data sets was undertaken and is discussed in [DATT88]. We have carried out additional experiments as well, and give some of our recent results in Figure 1 and in Tables 1 and 2. However, we also show results using our own algorithms in those charts and so suggest to the reader that these charts be examined later or that unexplained aspects of the charts be ignored.

Our experimental investigation has been carried out using a Macintosh II system with 2 megabytes of Apple memory and 8 megabytes of slower speed National Semiconductor memory. The Macintosh was run using the A/UX version of UNIX, with all programs written in the C language and compiled with the standard software available.

While there are many observations that can be made about Sager's algorithm, two points we feel are especially important are:

- There was no backtracking observed in any of the sets, which suggests that the ordering heuristic is quite good.
- The entire computation is dominated by the process of tower-building which is of time complexity $O(r^4)$. In case $|R| = M$, the complexity is $O(M^4)$.

However, while the algorithm is polynomial, the complexity $O(r^4)$ means that it is still not practical for large sets of words. Thus, the tower building process for a set of size 120 required 0.29 hours, suggesting that a set of size 240 would require 4.72 hours (i.e., 16 times as long).

4. New algorithm

We decided that based on our analysis of Sager's method, a new algorithm could be developed which would be more efficient. We make the following observations about Sager's algorithm and suggest why and how it could be improved.

- It is crucial to have the freedom to map the key space to a different (I, R_1, R_2) space, through the adjustment of h0, h1, and h2 functions. Doing this causes the corresponding bipartite graph to vary. Since the later ordering and searching depend on the bipartite graph, a measure of the quality of the graph may indicate how successful the search is likely to be. Thus, it is worthwhile to improve the quality of the graph that is constructed to reduce the cost of later processing.

- It is important to place any large levels early in the tower. Sager's ordering strategy works well but is expensive for large sets. We believe a good word order can be found using more efficient ordering strategies.

- Searching for values for the canonical edges starts at 0 and proceeds incrementally to N-1. This can lead to undesirable clustering of hash values. Using a random search of the set of possible values should have advantages.

Based on the above considerations, we have proposed and tested a new algorithm which is able to find a perfect hash function for a fairly large set (up to 1000 words) yet using less time than Sager's algorithm. We present the idea of the algorithm in three parts: §4.1 describes the process for setting up (I, R1, R2); §4.2 explains building the tower; and §4.3 describes the search stage.

4.1 Setting up (I, R1, R2)

The h0, h1 and h2 function suggested by Sager are:

$$
\begin{aligned}
h0(w) &= (\mathrm{length}(w) + \textstyle\sum \mathrm{ord}(\ w[i],\ i= 1 \text{ to } \mathrm{length}(w) \text{ by } 3)\ )\\
h1(w) &= (\textstyle\sum \mathrm{ord}(w[i], i =1 \text{ to } \mathrm{length}(w) \text{ by } 2) \bmod r\\
h2(w) &= ((\ \textstyle\sum \mathrm{ord}(w[i],\ i = 2 \text{ to } \mathrm{length}(w) \text{ by } 2) \bmod r\ ) + r
\end{aligned}
$$

In our implementation, ord(w[i]) returns the ASCII value of the i^{th} letter of the word.

Though these functions work well for many small sets, there have been cases where they lead to collision among (h0, h1, h2) pairs for a set of words taken from the *Collins English Dictionary* [HANK79]. In general, to handle arbitrary sets it is necessary to vary h0, h1, and h2. In our implementation, the user (i.e., the person using our system to build a perfect hash function for a given set) can choose from among the following families of more general h0, h1 and h2 functions:

a) h0, h1 and h2 in the form of

$$
\begin{aligned}
h0(w) &= (\ \mathrm{length}(w) + \textstyle\sum \mathrm{ord}(\ w[i], i=ih0 \text{ to } \mathrm{length}(w) \text{ by } step0)),\\
h1(w) &= (\ \textstyle\sum \mathrm{ord}(w[i], i=ih1 \text{ to } \mathrm{length}(w) \text{ by } step1) \bmod r,\\
h2(w) &= ((\ \textstyle\sum \mathrm{ord}[w[i], i=ih2 \text{ to } \mathrm{length}(w) \text{ by } step2) \bmod r\ ) + r,
\end{aligned}
$$

where the iho, ih1, ih2, step0, step1, and step2 are parameters defined by the user.

b) h0, h1 and h2 in the form of

$$
\begin{aligned}
h0(w) &= (\mathrm{length}(w) + \textstyle\sum \mathrm{ord}(\ f(w[i]), i = ih0 \text{ to } \mathrm{length}(w) \text{ by } step0)),\\
h1(w) &= (\ \ \textstyle\sum \mathrm{ord}([f(w[i]), i = ih1 \text{ to } \mathrm{length}(w) \text{ by } step1) \bmod r,\\
h2(w) &= ((\ \textstyle\sum \mathrm{ord}(f(w[i]), i = ih2 \text{ to } \mathrm{length}(w) \text{ by } step2) \bmod\ ) + r,
\end{aligned}
$$

where f(w[i]) maps ord(w[i]) to a pre-defined random number; as before, ih0, ih1, ihw step0, step1 and step2 are parameters.

We used the chi-square statistical test to measure the randomness of the sequences $h1(w_1)$, $h1(w_2)$, ... $h1(w_n)$ and $h2(w_1)$, $h2(w_2)$, ..., $h2(w_n)$. Analysis in a later paper will show that a random bipartite graph saves searching effort. Getting a randomness measure of the two sequences will indicate indirectly the randomness of the bipartite graph. We have found that when we follow Sager's original algorithm, the chi-square measure indicates that the sequences are not nearly random. When we use functions from the family described in (b) above, we have much more random sequences.

4.2 Building the tower

In this stage, we build a tower of levels of words, just as in the ordering stage of Sager's algorithm. The time required for our ordering stage is $O(r^3)$.

The ordering is done in two steps. In the first step, a maximum spanning forest Tsp is found for the bipartite graph, using Prim's algorithm; edge weights are given by edge multiplicities. The edges in Tsp will be the canonical edges of the tower. Building such a Tsp is based on the observation that when $M=|R|$, there are not many edges with multiplicity greater than 1. Thus Tsp will contain almost all of these multiple edges. These edges with multiplicity greater than 1 each stand for a set of words that are dependent on each other. Putting such a set in the tower early tends to decrease searching time.

Once Tsp is built, the second step constructs a search sequence. Let $e_0, e_1, \ldots, e_{i-1}$ be the canonical edges selected before, and e_n the canonical edge chosen in the current step. Then the level of tower W_i is given by

$$\{e_i\} \cup \{\, x \mid x \text{ is non-spanning edge and } x \text{ and } e_i \text{ and any subset of } \{e_0, e_1, \ldots, e_{i-1}\} \text{ form cycle(s)} \}.$$

Let ClassI be the set of canonical edges with weight > 1, classII the set of canonical edges lying in at least one cycle of length more than 2, ClassIII all canonical edges not in ClassI and ClassII, and w(n) the number of edges in the subgraph that has been reduced (see below) to vertex n. We produce ordered lists from these three sets as follows:

For ClassI, the list is sorted by the multiplicity value.
For ClassII, the list is sorted by the number of cycles.
For ClassIII, the list is sorted by $(w(n_1) + w(n_2) + X(n_1,n_2))$
where n_1, n_2 are the vertices of the edge and $X(n_1,n_2)$ is the number of edges between the subgraphs represented by n_1 and n_2.

Note that the sorting is done in non-increasing order.

Then the ordering heuristic is to follow the three steps below, in sequence:

a) Select edges from ClassI one by one based on the sorting order until the list is empty.
b) Select edges from ClassII one by one based on the sorting order until all of the edges left have the same value (assigned during the list sort phase above)
c) Select edges in order from ClassIII one by one until the list is empty.

4.3 Search Stage

This stage is almost the same as Sager's algorithm, except that initial values for canonical edges are randomly set.

5. Experimental results

In our recent experimental study we have selected word sets of varying sizes from the *Collins English Dictionary* [HANK79]. We compare two versions of our algorithm, PHF0 and PHF1, against the result of using Sager's algorithm. PHF0 is our perfect hashing function where the assigned U values for hashing are constructed without the use of randomization in the searching stage. PHF1 varies only in our use of randomization during search.

Figure 1 shows that Sager's algorithm indeed has complexity $O(n^4)$ while our method PHF1 has lower complexity (i.e., we have claimed complexity $O(n^3)$). The detailed data in Table 1 demonstrates this more completely, by showing times for various sets. We further give the break down of times for each stage of the processing. It can be seen that backtracking is generally avoided (PHF1 is slightly better than PHF0, as expected). We were unable to build perfect hashing functions using Sager's algorithm or PHF0 for the largest sets.

Table 2 adds additional details regarding the interaction of set size (M) and graph size (measured in terms of 2*r). We see that small graphs cause the algorithm performance to degrade.

We observe:

- Our algorithm works very well when M=2r and LF=1. No backtracking occurs for set sizes from 10 to 1000. The entire computation time is dominated by the tower-building process which is $O(M^3)$. Therefore, our algorithm is more practical than Sager's in terms of running time.
- Randomizing the initial U values makes the searching time of our algorithm less than Sager's.
- The algorithm degrades when 2r is set somewhat smaller than M. For example, when $2r / M < 0.63$, the search did not finish within 20 backtracks.

We have used method PHF1 to build a number of hash functions for the words in the Collins dictionary. A demonstration program for 69806 words has been built — we constructed a number of each with 256 words and an associated hash function.

Most recently we have developed a new, much better algorithm and have built a perfect hashing function for a set of 5000 word.

6. Conclusion

In this paper, a new perfect hash function finding algorithm is described which adopts a more efficient ordering heuristic. The experiment data shows that the ordering heuristic is able to incur no backtracking for data sets up to 1000 words, when the parameters are properly set. Thus, the algorithm may achieve the same effect as Sager's algorithm yet keep the execution time small enough to be practical. We have applied this work to our research in CD-ROM and have developed a new more efficient algorithm that will be described in a future paper.

References

[CERC83] N. Cercone, M. Krause, J. Boates, "Minimal And Almost Minimal Perfect Hash Function Search With Application To Natural Language Lexicon Design," Computers and Mathematics With Applications, 1983, V9, N1, pp.215-231.

[CICH80] R.Cichelli, "Minimal Perfect Hash Functions Made Simple," Comm. ACM, 1980, V23, pp.17-19.

[CHAN86] C. Chang, "Letter Oriented Reciprocal Hashing Scheme," Information Science, 1986, V38, N3, pp.243-255

[DATT88] S. Datta, "Implementation Of A Perfect Hashing Function Scheme," MS. report, Department of Computer Science, Virginia Polytechnic Institute & State University, Blacksburg, VA, March 1988.

[ENBO88] R. J. Enbody and H. C. Du. "Dynamic Hashing Schemes," ACM Computing Surveys, June 1988, V20, N2, pp. 85-114.

[FOXE86] E. Fox, R. Wohlwend, P. Sheldon, Q. Chen, and R. France. "Building the CODER Lexicon: The Collins English Dictionary and Its Adverb Definitions," TR-86-23, Department of Computer Science, Virginia Polytechnic Institute & State University, Blacksburg, VA, October 1986.

[FOXE87] E.Fox, "Development Of The CODER System: A Testbed for Artificial Intelligence Methods In Information Retrieval, "Information Processing And Management, 1987, V23, N2, pp.341-366.

[FOXE88] E. Fox, editor and project manager. Virginia Disc One. CD-ROM developed at Virginia Polytechnic Institute & State University, Blacksburg VA, and produced by Nimbus Records, Ruckersville VA, September 1988.

[HANK79] P. Hanks, ed. Collins English Dictionary. William Collins Sons & Co., London, 1979.

[JAES81] G.Jaeschke, "Reciprocal Hashing — A Method For Generating Minimal Perfect Hash Functions," Comm. ACM, 1981, V24, N12, pp.829-833.

[SAGE85] T. Sager, " A Polynomial Time Generator For Minimal Perfect Hash Function," Comm. ACM, May 1985, V28, N5, pp.523-532.

[SAGE84] T. Sager, "A New Method for Generating Minimal Perfect Hashing Functions," TR CSc-84-15, University of Missouri-Rolla, Rolla, Mo, November 1984.

Fig. 1 Tower Build Time, Sager's vs. PHF1

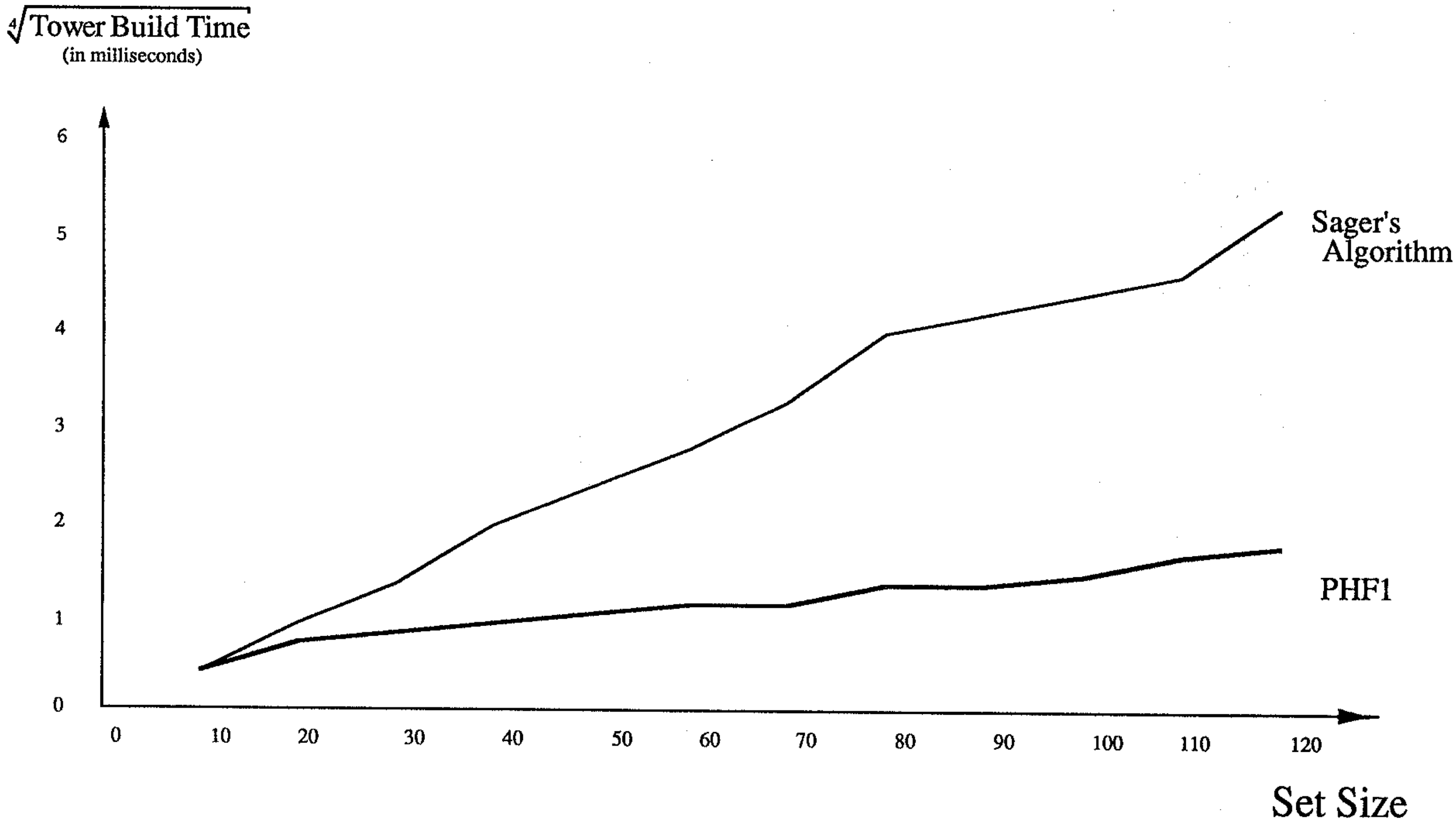

Table 1
Performance Comparison of Algorithms on Different Size Sets

Set	Tower Build Time			Search Time			Height of Tower			No. of Backtracks		
Size	Sager	PHF0	PHF1	Sager	PHF0	PHF1	Sager	PHF0	PHF1	Sager	PHF0	PHF1
10	49	99	99	16	66	83	7	8	8	0	0	0
20	1049	249	266	66	316	316	17	17	17	0	0	0
30	3399	433	433	166	399	383	23	21	21	0	0	0
40	16549	716	716	416	416	499	34	33	33	0	0	0
50	28198	1060	1049	683	616	433	41	38	38	0	0	0
60	54231	1416	1399	1099	816	999	44	52	52	0	0	0
70	120261	1983	1999	1466	1099	849	56	57	57	0	0	0
80	241723	2566	2566	2333	1599	1366	70	67	67	0	0	0
90	323320	3433	3433	3683	2899	1299	74	72	72	0	0	0
100	519845	4266	4233	6883	11299	2149	81	82	82	0	16	0
110	385801	5783	5766	9399	7666	2616	77	82	82	0	0	0
120	861415	6466	6466	11340	10566	2633	93	97	97	0	1	0
240	n/a	n/a	32682	n/a	n/a	33382	n/a	n/a	168	n/a	n/a	0
500	n/a	n/a	189475	n/a	n/a	264385	n/a	n/a	395	n/a	n/a	1
1000	n/a	n/a	2101032	n/a	n/a	413133	n/a	n/a	615	n/a	n/a	0

Key:
Set size = number of words
PHF0 = new algorithm without randomization of initial U values
PHF1 = new algorithm with randomization of initial U values

Notes:
Times are measured in milliseconds using the A/UX system routine “clock.”
LF = 1, M = 2r.

Table 2
Performance Comparison of Algorithms on Different Size Sets with various values of r

Set Size	Graph Size	Tower Build Time Sager	Tower Build Time PHF0	Tower Build Time PHF1	Search Time Sager	Search Time PHF0	Search Time PHF1	Height of Tower Sager	Height of Tower PHF0	Height of Tower PHF1	No. of Backtracks Sager	No. of Backtracks PHF0	No. of Backtracks PHF1
40	40	16549	716	716	416	416	499	34	33	33	0	0	0
40	30	1999	483	466	383	483	fail	26	-	-	2	-	-
40	26	c	399	383	-	483	fail	-	-	-	-	-	-
40	20	c	c	c	-	-	-	-	-	-	-	-	-
60	60	54231	1416	1399	1099	816	999	44	52	52	0	0	0
60	46	9382	933	849	1116	733	716	38	39	40	3	0	0
60	38	2949	720	716	1533	766	fail	34	35	-	13	0	-
60	30	c	540	533	-	fail	fail	-	-	-	-	-	-
80	80	241723	2566	2566	2333	1599	1366	70	67	67	0	0	0
80	60	42048	1449	1466	1660	1616	799	54	54	54	0	16	0
80	52	19049	1166	1133	2016	3199	fail	49	51	-	7	11	-
80	40	c	816	799	-	2966	fail	-	38	-	-	12	-
100	100	519845	4266	4233	6883	11299	2149	81	82	82	0	16	0
100	76	108862	2199	2233	7483	fail	1149	69	-	71	0	-	0
100	64	21999	1649	1716	4333	fail	fail	60	-	-	1	-	-
100	50	5283	c	c	536795	-	-	47	-	-	3598	-	-

Key:
Set size = number of words
PHF0 = new algorithm without randomization of initial U values
PHF1 = new algorithm with randomization of initial U values
c= (h0, h1, h2) collision
fail = backtracking more than 20 times while searching

Notes:
Times are measured in milliseconds using the A/UX system routine "clock."
LF = 1.