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]; 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 [FOX88]. 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 $h_0$, $h_1$ and $h_2$ which will hopefully each lead to a uniform distribution of result values for a given input set. The functions recommended by Sager are:

$$h_0: W \rightarrow I \text{ (a finite set of integers defined only by } h_0 \text{ and the data);}$$
$$h_1: W \rightarrow R_1, R_1 = \{ x \mid x \in [0..r-1] \} \quad \text{so } |R_1| = r;$$
$$h_2: W \rightarrow R_2, R_2 = \{ x \mid x \in [r..2r-1] \} \quad \text{so } |R_2| = r.$$

Each word is associated with the tuple $(h_0(w), h_1(w), h_2(w))$. No two words may be associated with the same tuple; if such a collision occurs, new $h_0$, $h_1$, and $h_2$ functions must be selected. A bipartite graph is formed with vertices labelled with each of the values in the range of the functions $h_1$ and $h_2$; an edge for each word $w$ connects the nodes labelled $h_1(w)$ and $h_2(w)$. 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 $h_0$, $h_1$, and $h_2$ 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:

\[
\begin{align*}
    h0(w) &= (\text{length}(w) + \sum \text{ord}(w[i]), \ i = 1 \text{ to } \text{length}(w) \text{ by } 3) ) \\
    h1(w) &= (\sum \text{ord}(w[i]), \ i = 1 \text{ to } \text{length}(w) \text{ by } 2) \mod r \\
    h2(w) &= ((\sum \text{ord}(w[i]), \ i = 2 \text{ to } \text{length}(w) \text{ by } 2) \mod r ) + r
\end{align*}
\]

In our implementation, \(\text{ord}(w[i])\) 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 \(h0, h1, \text{ and } h2\).

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

\[
\begin{align*}
    h0(w) &= (\text{length}(w) + \sum \text{ord}(w[i]), \ i = \text{ih}0 \text{ to } \text{length}(w) \text{ by step0}) , \\
    h1(w) &= (\sum \text{ord}(w[i]), \ i = \text{ih}1 \text{ to } \text{length}(w) \text{ by step1}) \mod r, \\
    h2(w) &= ((\sum \text{ord}(w[i]), \ i = \text{ih}2 \text{ to } \text{length}(w) \text{ by step}2) \mod r ) + r,
\end{align*}
\]

where the \(\text{ih}0, \text{ih}1, \text{ih}2, \text{step}0, \text{step}1, \text{and step}2\) are parameters defined by the user.

b) \(h0, h1\) and \(h2\) in the form of

\[
\begin{align*}
    h0(w) &= (\text{length}(w) + \sum \text{ord}(f(w[i]), \ i = \text{ih}0 \text{ to } \text{length}(w) \text{ by step0}) , \\
    h1(w) &= (\sum \text{ord}(f(w[i]), \ i = \text{ih}1 \text{ to } \text{length}(w) \text{ by step1}) \mod r, \\
    h2(w) &= ((\sum \text{ord}(f(w[i]), \ i = \text{ih}2 \text{ to } \text{length}(w) \text{ by step}2) \mod r ) + r,
\end{align*}
\]

where \(f(w[i])\) maps \(\text{ord}(w[i])\) to a pre-defined random number; as before, \(\text{ih}0, \text{ih}1, \text{ih}w\ \text{step}0, \text{step}1\) and \(\text{step}2\) 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 $T_{sp}$ is found for the bipartite graph, using Prim's algorithm; edge weights are given by edge multiplicities. The edges in $T_{sp}$ will be the canonical edges of the tower. Building such a $T_{sp}$ is based on the observation that when $M=|R|$, there are not many edges with multiplicity greater than 1. Thus $T_{sp}$ 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 $T_{sp}$ is built, the second step constructs a search sequence. Let $e_0, e_1, ..., e_{i-1}$ 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

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

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


Table 1
Performance Comparison of Algorithms on Different Size Sets

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

**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

<table>
<thead>
<tr>
<th>Set Size</th>
<th>Graph Size</th>
<th>Tower Build Time</th>
<th>Search Time</th>
<th>Height of Tower</th>
<th>No. of Backtracks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Sager PHF0 PHF1</td>
<td>Sager PHF0 PHF1</td>
<td></td>
<td>Sager PHF0 PHF1</td>
</tr>
<tr>
<td>40</td>
<td>40</td>
<td>16549 716 716</td>
<td>416 416 499</td>
<td>34 33 33</td>
<td>0 0 0</td>
</tr>
<tr>
<td>40</td>
<td>30</td>
<td>1999   483 466</td>
<td>383 483 fail</td>
<td>26 - -</td>
<td>2 - -</td>
</tr>
<tr>
<td>40</td>
<td>26</td>
<td>c      399 383</td>
<td>- 483 fail</td>
<td>- - -</td>
<td>- - -</td>
</tr>
<tr>
<td>40</td>
<td>20</td>
<td>c      c c</td>
<td>- - -</td>
<td>- - -</td>
<td>- - -</td>
</tr>
<tr>
<td>60</td>
<td>60</td>
<td>54231 1416 1399</td>
<td>1099 816 999</td>
<td>44 52 52</td>
<td>0 0 0</td>
</tr>
<tr>
<td>60</td>
<td>46</td>
<td>9382   933 849</td>
<td>1116 733 716</td>
<td>38 39 40</td>
<td>3 0 0</td>
</tr>
<tr>
<td>60</td>
<td>38</td>
<td>2949   720 716</td>
<td>1533 766 fail</td>
<td>34 35 -</td>
<td>13 0 -</td>
</tr>
<tr>
<td>60</td>
<td>30</td>
<td>c      540 533</td>
<td>- fail fail</td>
<td>- - -</td>
<td>- - -</td>
</tr>
<tr>
<td>80</td>
<td>80</td>
<td>241723 2566 2566</td>
<td>2333 1599 1366</td>
<td>70 67 67</td>
<td>0 0 0</td>
</tr>
<tr>
<td>80</td>
<td>60</td>
<td>42048 1449 1466</td>
<td>1660 1616 799</td>
<td>54 54 54</td>
<td>0 16 0</td>
</tr>
<tr>
<td>80</td>
<td>52</td>
<td>19049 1166 1133</td>
<td>2016 3199 fail</td>
<td>49 51 -</td>
<td>7 11 -</td>
</tr>
<tr>
<td>80</td>
<td>40</td>
<td>c      816 799</td>
<td>- 2966 fail</td>
<td>- 38 -</td>
<td>- 12 -</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>519845 4266 4233</td>
<td>6883 11299 2149</td>
<td>81 82 82</td>
<td>0 16 0</td>
</tr>
<tr>
<td>100</td>
<td>76</td>
<td>108862 2199 2233</td>
<td>7483 fail 1149</td>
<td>69 - 71</td>
<td>0 - 0</td>
</tr>
<tr>
<td>100</td>
<td>64</td>
<td>21999 1614 1649</td>
<td>4333 fail fail</td>
<td>60 - -</td>
<td>1 - -</td>
</tr>
<tr>
<td>100</td>
<td>50</td>
<td>c      c 536795</td>
<td>- - 47 -</td>
<td>- 3598 -</td>
<td>- - -</td>
</tr>
</tbody>
</table>

**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.