1987

Signature file access methodologies for text retrieval: a literature review with additional test cases

Karen Caviglia

ROCHESTER INSTITUTE OF TECHNOLOGY
SCHOOL OF COMPUTER SCIENCE AND TECHNOLOGY

SIGNATURE FILE ACCESS METHODOLOGIES FOR TEXT RETRIEVAL A LITERATURE REVIEW WITH ADDITIONAL TEST CASES

by Karen Caviglia

Submitted to
the Faculty of the School of Computer Science and Technology
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

Approvals:

Jeffery Lasky
Dr. Jeffrey Lasky, Chair

Chris Comte
Chris Comte

Andrew Kitchen
Dr. Andrew Kitchen

Peter Anderson
Dr. Peter Anderson

SIGNATURE FILE ACCESS METHODOLOGIES FOR TEXT RETRIEVAL
A LITERATURE REVIEW WITH ADDITIONAL TEST CASES

by Karen Caviglia
in partial fulfillment of the requirements of the
Master's Degree in Computer Science
Rochester Institute of Technology
Rochester, New York

April, 1987

Thesis Committee
Dr. Jeffrey Lasky, Chair
Chris Comte
Dr. Andrew Kitchen
Dr. Peter Anderson

ABSTRACT

Signature files are extremely compressed versions of text files which can be used as access or index files to facilitate searching documents for text strings. These access files, or signatures, are generated by storing "hashed" codes for individual words. Given the possible generation of similar codes in the hashing or storing process, the primary concern in researching signature files is to determine the accuracy of retrieving information. Inaccuracy is always represented by the false signaling of the presence of a text string. Two suggested ways to alter false drop rates are: 1) to determine if either of the two methologies for storing hashed codes, by superimposing them or by concatenating them, is more efficient; and 2) to determine if a particular hashing algorithm has any impact.

To assess these issues, the history of suprimposed coding is traced from its development as a tool for compressing information onto punched cards in the 1950s to its incorporation into proposed signature file methodologies in the mid-1980's. Likewise, the concept of compressing individual words by various algorithms, or by hashing them is traced through the research literature.

Following this literature review, benchmark trials are performed using both superimposed and concatenated methodologies while varying hashing algorithms. It is determined that while one combination of hashing algorithm and storage methodology is better, all signature file mehods can be considered viable.

TABLE OF CONTENTS

FIGURES LIST

CHAPTER 1. INTRODUCTION

1.1 FORMATTED VS. UNFORMATTED RETRIEVAL

In the area of computer storage of data, much research and development has focused on the concept of database, a term which has become nearly synonymous with three classical database models: hierarchical, CODASYL, and relational. The distinguishing characteristic of data stored in any of these three models is that user access is by means of an abstract, logical model which organizes the underlying physical data representations into one or more user views or data representations. These logical models are constructed by extracting attributes and keys from data and then defining relationships between them. Groupings of attributes are referred to as entities and the relationships between entities (one to one, one to many, and many to many) constitute the basic design of the database.

For each of the three standard models there is a unifying data structure: a tree for the hierarchical model, a graph for the network model, and n-ary relationships (depicted as tables or sets) for the relational model. Such database structures are well-developed both commercially and theoretically and therefore are attractive mechanisms for the storage and retrieval of data in formatted

representations. Each database model has its own tradeoffs between the additional storage overhead of access files needed by the abstract model, ease of retrieval of data, and query processing efficiency.

On the other hand, an extremely large body of data exists for which such formatted models are inappropriate. This unformatted data, primarily text, constitutes a potentially larger body of stored information than formatted data. Letters, memos, reports, journal indexes and abstracts, complete journal articles, books, and even encyclopedias are good examples of text files. Extraction of attributes and keys from such data is limited to identifying common fields such as the addressee of a memo, the book title, or the author's name. However, attribute extraction still leaves the body of the memo, or diverse content of the book unrepresented in a direct way. Textual information can be represented in a database system only to the extent that easily identifiable fields adequately describe the total contents.

Indexing to create additional description fields is a traditional approach to solving this problem of content addressability. For example, a library card catalog offers more content access points than a list of book titles because of the subject indexing. And book indexes augment

the table of contents enormously. However, indexing is labor intensive effort, even when aided by automatic indexing. Commercially available computerized indexing systems typically offer users little more than a way to flag keywords for inclusion in an index. These terms are then formatted automatically into an index with the page number supplied as a final step for the word processing software. There are a few experimental systems which attempt true automated indexing by statistical analysis of the text in order to identify good index terms. In either case, whether the index is generated by manual effort or by complex machine textual analysis, indexing is a costly method for providing content addressability of text files.

Furthermore, indexing itself is an imperfect tool for content addressability since it limits access to the specified list of index terms. Queries which might be resolved by an exhaustive search of the complete original document may go unanswered if limited to index terms or subject headings.

With this understanding of the nature of textual data and the problems inherent in providing access to it, several methodologies for text retrieval will be reviewed.

1.2 TEXT RETRIEVAL METHODOLOGIES: A BRIEF REVIEW

Existing text retrieval methodologies may be classified into four groups: 1) Full text scanning; 2) Inverted files; 3) Clustering; and 4) Signature files. Of these, only the first two methods have been widely used in commercial systems and environments.

1.2.1 Full Text Scanning

Full text scanning works by searching sequentially through a file for a particular character pattern. Such patterns are matched exactly, although some systems provide partial match capability for substring searching. The advantage of this method is that it requires no additional storage overhead, as files are searched in whatever form they already exist. However, the processing time for text scanning, even with optimized string searching algorithms, can be a major constraint. For large files in an interactive environment, full text scanning is generally not feasible.

1.2.2 Inverted Files

An inverted file is a secondary access file which contains an ordered list of words taken from the original file, coupled with pointers to the original file. (If desired, such pointers may also include more exact word addresses such as the the line number or word number in the

original file.) At the extreme, every word in the original file can be stored in the access file to create a completely inverted file. Alternatively, a list of keywords may be used to create the inverted file, which then resembles a data dictionary. Most commonly, a stop list of common words may be excluded from the inversion process. Inverted files can also be maintained in a manner that preserves information about the order of words in a document. For example, if words have been stored with their relative position (word number), this information can be used for proximity or phrase searching. This means that rather than searching for a phrase "free text" as a whole, the fact that "free" is in posiion 3 and "text" is in position 4 indicates that the term "free text" is present in the document.

The advantage of the inverted file is that the ordering of the words in the access file provides extremely fast query processing. However, such rapid access is at the expense of a large storage overhead. Such systems require as much as 50-300% additional space for the inverted file. Furthermore, appending, deleting, or inserting data into inverted files is time consuming and costly, since the entire inverted file must be reorganized and rewritten. Thus, inverted file methodology is often limited to archival or at least stable files whose update requirements allow

batch processing to take place during periods of low system use.

1.2.3 Clustering

Clustering as a retrieval methodology is an attempt to provide computerized classification of documents. The classification schema is derived from similarity measures between documents. Retrieval, and sometimes storage location, is then based upon document clusters. The object of clustering is to provide a range of possibly relevant documents in response to a given request, whether or not there is an exact match between the vocabulary of the request and the vocabulary of the document. While full text scanning and inverted files may be used to place a phrase within a document, clustering normally occurs only at the document level with no attempt to cluster smaller sub-units of information. Like automated indexing, there is a need for computer text analysis (or human input) to determine significant keywords or attributes in order to produce a similarity matrix. As a technique clustering has been mostly of theoretical interest and there are few commercial applications at this time.

1.2.4 Signature Files

A less common text retrieval methodology is signature files. As will be shown in the historical reviews in Chapters 2-4, the underlying techniques for signature files have been in existence since the 1950's, but the specific development of the concept of signature files is attributable to a research group at the University of Toronto in very recent years [Faloustsos,1984,1985; Christodoulakis,1984,1985; Tsichritzis,1985].

The basic idea is to create an extremely compressed version of the original file, called a signature file, which can serve as an access file. Since this signature is much smaller than the original, it can be searched more efficiently and quickly than the original document. The major issue in creating and using signatures is that the extreme compression of the original document can introduce errors into the retrieval process. As will be seen in detail later, the errors generated are always indications that requested words exist in the original document when in fact they do not. Such errors are referred to as false drops, a term which has remained from the punched card era, and the prediction and limiting of false drop rates is the primary concern in the development of signature file methodologies.

1.3 AN INTRODUCTION TO SIGNATURE FILE METHODOLOGY

Because signature file techniques are the least known of the technologies under discussion here, a more complete introduction to signature file methodologies is provided below. The basic processes of encoding and retrieving information involve hashing individual words, then either building the signature or searching it. The major types of signature files, superimposed signatures and concatenated word signatures, are two different ways of building the signature file itself.

1.3.1 Encoding and Retrieving from Signature Files

Both approaches to building signature files begin by hashing each original document word. A simple hash algorithm (figure 1.1) for a word might be based on the first four characters of the word and return four values:

```
for i = 1 to 4
 hash[i] = (ord(word[i]) * i * prime) mod signature_size
```

<Simple Hash Algorithm - Fig. 1.1>

Unlike traditional hashing for hash table storage, this algorithm makes no attempt to resolve hash collisions. Different original text words may generate the same hash values. Since the original word is not stored in the signature itself, there is no way to determine if colliding hash codes represent repeating words or different words

which have produced the same numerical hash values.

Once the hashed codes for individual words have been derived, the signature itself may be built by either: 1) word signatures, the concatenation of individual hashed words or 2) superimposed coding, the superimposing of all the hash values into one large coding field. Both of these methods will be discussed in more detail below.

The retrieval process for signature file access reverses the encoding/storage process. The query word is hashed to produce codes, which are then searched for in the signature. However, the storing process did not resolve hash collisions; and if superimposed coding is being used, additional valid codes may have been generated in the superimposition process. The retrieval process, then, is likely to produce a number of false drops; i.e., the presence of a query word is signaled, where in fact that word does not exist in the original text. Note that the failure to find a match is a guarantee that the word does not exist in the text. There are no "false dismissals" in this methodology.

Thus, unlike full text scanning or inverted files, signature file retrieval is probabilistic and the primary concern for the investigation of the feasability of signature files becomes the determination of the false drop

probability for signature file methodologies. Any implementation of a signature file methodology must also provide for the screening of these false drops.

1.3.2 Superimposed Coding Storage vs. Word Signature Storage

The following example (figure 1.2) demonstrates how a superimposed signature is created for a document of four words: "free text retrieval methods". In this example each word is hashed to generate four codes using an algorithm such as fig. 1.1. Assuming that the word "free" generated the values of 3, 7, 10 and 15, the corresponding bits of an "intermediate" signature are set to 1. The codes for each word are then logically ORed to produce the resulting signature of 16 bits.

Word	Signature
bit no.	111111 5432109876543210
Free	1000010010001000
Text	0011000000001010
Retrieval	1010010010000000
Methods	0010000111000010
SUPERIMPOSED SIGNATURE	1011010111001010

<Creating a Superimposed Signature - Fig. 1.2>

For the purposes of illustration, the results of the hashing for each word have been presented above using an "intermediate" signature. In actual usage it would be more

efficient to "OR" each individual hash code with the final signature, rather than build intermediate signatures.

The following example (figure 1.3) shows the concatenation of individual word signatures into a resulting signature.

Word:	Free	Text	Retrieval	Methods
Signature:	00000000	10000100	00100111	1111101:

WORD SIGNATURE 00000000/10000100/00100111/11111011

<Creating a Concatenated Word Signature - Fig. 1.3>

For retrieval, a query word is processed by the same hashing algorithm. The result is then searched for in either the superimposed coding field, or in the concatenated word signature. For example, in the case of superimposed coding, if the hashed word "free" set bits 3, 7, 10, and 15, then the fact that these bits are set to one in the superimposed signature indicates that the word "free" may be in the original document. In the case of word signatures a hash code of 132 (10000100) for the word "retrieval" would be searched for sequentially through the concatenated word signatures. If found, then, again, the word "retrieval" may be in the original.

A false drop could be produced in the following way. In superimposed coding, if the word "full" produced hashed

values of 1, 7, 10, and 11, the bits for these values would be found to be ones in the superimposed field. However, the word "full" is not in the original text. In the case of word signatures, if the word "full" produced a hashed value of 0, it would match with the word signature for the word "free." With only 256 possible, distinct combinations, it is quite likely that even a hash code with a theoretically even distribution would produce such collisions.

The two examples presented above have used trivial sized "documents" and "signatures." But even these show that the possible number of variables for superimposed coding is greater. One can vary the total size of the signature field as well the number of hashes per word. On the other hand, for word signatures, one is left primarily with the size of the individual word signature as the only variable. In practical applications, the word signature is best searched if the individual word signatures fit on computer word, or half-word boundaries. This effectively restricts word signatures to 16, 32, or 64 bits. Once the size of the individual word signature is determined, the final signature file grows with the number of words in the document. Superimposed signatures do not increase in size with the number of words, but rather become more dense (a higher percentage of the bits set to one). As will be amply

demonstrated in the review of the research literature, superimposed fields which set fewer than 50% of the bits to one are viable.

The examples above show that superimposed coding used 4 bytes to represent a total of 24 bytes of character data (not counting blanks between words). Concatenated word signatures used 8 bytes to represent the 24 bytes. As the test cases in Chapter 5 demonstrate, superimposed fields can actually be used which are only about 10% of the size of the original documents they represent. Word signatures never achieve such a savings, and in fact they are likely to be 25% to 30% of the original document in size.

1.3.3 Evaluation of Signature Files

In terms of overhead space, signature files are economical. If a signature file is estimated to take only 10%-30% of the original text in size, it is significantly smaller than an inverted file, which is always larger than the original document. It should be noted that signature files, like inverted files, can be reduced in size by the exclusion of stop words; that is, common, insignificant English words. Further, an optimization of superimposed signature files has been suggested which would use "run-length variable coding" to compress potentially sparse signatures [Faloustsos, 1985b]. Under this system,

superimposed signature sizes could be expanded in virtual size to reduce the number of collisions created by both the hashing algorithm and the superimposed coding, and thereby reduce the false drop rates. Yet the variable length encoding would not increase the actual stored size of the signature

Maintenance of signature files requiring additions to a document or file signature can be handled readily for superimposed coding fields, and with only minor difficulty for concatenated word signatures. Deletions, on the other hand, require the complete regeneration of the access file unless the false drops generated by leaving the signature file untouched could be tolerated or handled in other ways. Therefore, signatures would not be a preferred methodology for a rapidly changing file environment, but rather for more stable or archival environments where additions are more common than updates or deletions.

1.3.4 Current Applications and Research

Like clustering, there are no significant commercial applications of signature files. The closest implementations have been a hashed signature used as a line header in a word processor [Tharp] and a spelling checker which uses hash codes for words [McIlroy]. In the case of the word

processor, a signature was appended to each line and the signature was searched first to resolve queries. This search then resulted in fewer lines to be searched via string searching. In the case of the spelling checker, the original list of text words was discarded altogether. In this case, the existence of false drops would allow a few misspelled words to pass undetected, but the words returned as incorrect could be relied upon not to be in the original dictionary. These applications will be discussed in detail in Chapter 2.

As was mentioned earlier, the primary work done on signature files has been by a research group centered at the University of Toronto [Faloustous, 1984,1985; Christodoulakis, 1985; Tsichritzis, 1984]. Their work has been primarily analytical in nature, and while they posit the office automation environment as a particularly suitable use of signature files, they have not yet implemented any software using signature files for this application area. They propose that signature files are particularly attractive in the office automation area because of: 1) the amount of document handling in an office; 2) the archival nature of a large percentage of documents and letters; 3) the relatively smaller memory size of office automation machines such as microcomputers and workstations; and 4) the

possibility of creating signatures for non-text, graphical, or voice data to be integrated in an office system.

1.4 THESIS ORGANIZATION

The overall format of the thesis is to present a thorough literature review of the issues involved with signature files. As has already been mentioned there is no significant literature on word signatures themselves. The issues which will be covered are: 1) superimposed coding, and 2) hashing algorithms for textual material. Both of these areas are the points at which false drops are introduced into the signature file methodology. While there has been significant work done in both areas, the two areas have not been brought together in the research literature. For example, researchers developing equations for the false drop behavior of superimposed coding have normally done so by assuming completely random assignment of hash codes. It is not that these researchers are unaware of the nature of English text and the distribution of letter frequencies, but rather that to mathematically accommodate the behavior of text hashing would be extremely tedious, if not impossible.

The research on text hashing has also been unrelated to superimposed coding for the most part. The object of such research has been no more than the generation of code words which are smaller than the original text word. In the

process of reviewing these methods it will be seen that while some text hashing algorithms seem to perform better than others, none is capable of providing completely random hash codes for English text.

Following a review of these two areas, results from the author's own test cases will be presented. Simple programs were written to determine the overall utility of the signature file approach to text retrieval in terms of its false drop rates. Both superimposed coding programs and concatenated word signature programs were run against a sample text file, and results have been charted. The primary function of these test programs is to verify the general ratios of signature size to original document size, and the false drop ratios as presented in the literature. However, these programs have also varied their hashing algorithms in order to determine the importance of good hashing algorithms on false drop rates.

Chapter 2. SUPERIMPOSED CODING

2.1 INTRODUCTION TO SUPERIMPOSED THEORY

The history of superimposed codes, as well as the attendant need to devise strategies for translating subject descriptors, terms or codes into compact words begins with punched cards. While the concepts of mapping an item from a document to index terms for the purposes of retrieval began much earlier, the idea of further reduction of terms to extremely short codes was triggered by the development of punched or notched-edge cards.

In tracing this history there is a shift in terminology. Punches and holes will become bits. The codefield or codeword becomes a signature. And depending upon the particular application, the items mapped into the coding field will be index terms, descriptors, subjects or even just words. These terms have not been standardized in the following chronological review of the literature.

Of the various performance issues which are involved in superimposed coding, the calculation of false drop probabilities or rates is clearly the one which carries forward most easily across various generations of mechanized literature retrieval. Other issues, such as disk access, become implementation details. Many authors have speculated

on these issues, and some have been prone to suggest future architectures, such as associative processors, as well-suited to the process. The algorithmic performance issues depend on the data structures used to store the superimposed field. Few authors discuss this. For these reasons, the emphasis of this review will be on the false drop issue.

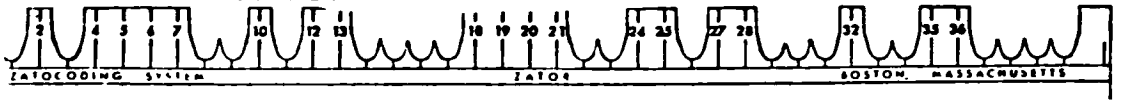
It will be seen that the development of superimposed coding methodology has been more theoretical than applied. Most authors spend significant time deriving complex mathematical formulas which will predict the performance of superimposed coding. Some of these equations will be repeated in the review. Some of these equations have been used to generate tables of data, which will also be reported.

There are only a few working applications of superimposed coding, and they will be presented. By reviewing both theory and applications, an attempt will be made to determine if there are some general rules or procedures to be used in developing a superimposed coding application.

2.2 ZATORCODING

One of the first suggested uses of superimposed coding [Mooers,1951] was for a system called the Zator machine.

This was a machine designed to mechanically sort notched edge cards. Calvin Mooers, President of the Zator Company, and the acknowledged originator of the superimposed coding concept, suggested that rather than dividing the number of code positions on a card into fields and assigning specific terms or descriptors to each field, that codes randomly assigned to descriptors be superimposed into one field. Mooers envisioned using such a system for a collection of documents where each document was represented by one card. Information about the document was typed on the card, as were the descriptor terms. The edges were then notched using the randomly assigned codes for faster retrieval. Instead of reading through all cards, needles could be inserted in notches, causing desired cards to drop out. Figure 2.1 shows an example of a notched edge card taken from the index system to a collection of 4000 documents with a descriptor list of 1000 terms.



Descriptors	Zatocodes				Reference
selective device	3	11	15	39	U. S. Patent No. 2,295,000
film tally	14	17	22	50	Rapid Selector-Calculator
photo-electric sensing	1	11	34	40	Richard S. Morse, Rochester, N. Y.
audio frequency code	9	16	29	31	
	1	8	29	34	one claim
flash	17	23	34	38	
counting	8	26	33	37	

<Example of a Zator Notched Edge Card - Fig 2.1>

Beginning with the assumption that all retrieval queries will be conjunctive queries of three or more terms, and working backwards, Mooers concluded that a choice among 4000 documents involves only 12 decision points (2**12) [Superscripted exponents are represented as ** when an in-line expression is warranted.] Mooers theorized that each decision halves the remaining documents. In essence a binary search is proposed even though this approach to an unordered set of documents is dubious. Nonetheless, if 12 terms are needed to decide and each inquiry will have three descriptors, four Zatorcodes will be needed for each descriptor. Given a card of 40 positions, there are 91,390 possible combinations (40 positions taken four at a time), which is more than adequate for 1000 terms if terms are assigned uniquely. However, it is suggested that the actual codes be assigned to the descriptors (four to each) randomly, rather than uniquely, since assigning 1000 terms manually and assuring uniqueness would be difficult. Lacking a computerized random number generator, balls numbered 1 to 40 were suggested. It should be noted from the example above that the Zatorcodes assigned to each document have no meaningful relationship to the descriptor itself.

Using the example above, it can be seen that more than one descriptor contains the code 11 in its Zatorcode. Given

this overlap, it was determined that seven different Zatorcodes (of four numbers each) could be marked before a density of more than 50% of the positions were used in the field of 40 positions. The rule derived for this was that the sum of the marks of all codes equals 69% when the average density equals 50%.

While Mooers had no empirical evidence for his example (the mechanics of typing, coding and notching 4000 cards would certainly not have been trivial), his insistence on two principles has persisted in the development of superimposed codes:

1) That no more than 50% of the coding field be marked
2) That input codes be generated randomly.

2.3 MATHEMATICAL COMPARISONS OF PUNCHED CARD SUPERIMPOSED CODING

A good review of the many applications of punched cards to information retrieval is found in Punched Cards: Their Application to Science and Industry [Case,1951]. In the chapter on the mathematical analysis of several of these, Wise [Wise,1951] discusses several types of coding.

Direct coding is the simplest coding system in which each hole represents one term. The obvious limitation is that the number of terms is set at the number of holes available to be punched. Using direct coding and a system of

seven descriptor fields with 26 positions each, it is predicted that if a card has one punch in each field, and if a needle is inserted at random in a field, the needle has a one in 26 chance of entering a punched position. If all seven fields are needled there is only a one in 26**7 chance (one in 8,031,810,000) that all seven random needles would enter punched positions.

If another item is added on the card, and the seven codes assigned to it are superimposed in the coding fields, then the probability of a single needle in one field becomes two out of 26, and for all seven fields needled it is 2**7 out of 26**7 (128 out of 8,031,810,000). However, the process of adding another term and its seven codes has a certain probability that some of these codes will overlap with the previous ones. Thus, coding a second item in each field has a probability (one out of 26) that it will occur in a hole already punched. Or, stated in reverse, it will have 25 chances out of 26 of punching a new hole.

Deriving equations based on these principles, Mooers calculates tables of punched values which yield dropping fractions. He concludes that if nine names or terms are coded in one 26-position field, the average number of positions punched out will be 0.29 not 0.346 (9/26). This means that a random needle in 100 cards would drop out 29 cards.

The author then turns his attention to the problem of unwanted cards (false drops). Using the same approach, if a card is wanted which has two fields (two descriptors) with particular punches, the dropping fraction will be 2(1/26)**2 or 2/676. On the other hand, without accounting for duplicate, or overlapping punches in the same hole, the dropping fraction for one needle is (2/26)**2 or 4/676. Subtracting the number of wanted cards from the total of random cards (4/676 - 2/676) = 2/676 unwanted cards. Thus, 50% of the cards would be wanted and 50% unwanted.

Finally Mooers compared his theoretical values to a test case and found that the number of positions actually punched was about 5% lower than the theoretical value of 29 he predicted and that the unwanted card fraction was near the theoretical values of 50%. However, the average percentage of total cards dropped was significantly higher than the theoretical values. Mooers presumes that this is because the searches were not based on random needling but were always on codes known to have been used.

It is noted that this false drop rate is extremely high compared with later systems. Unfortunately, Mooers did not state how many items were coded on each card or give the density of the resulting code field. However, the test cases in the final chapter would indicate that such a field of 26 positions with one code in each is small. Even viewing the

system as one code field of 26*7 with seven descriptors is still a relatively small field.

2.4 RANDOMIZING SQUARES

Wise's work was based on codes which, although they contained 26 characters in a field, were not specifically alphabetic and therefore did not suffer from the geometric distribution of character frequency distributions. (This will be covered in detail in the chapter on text hashing.) In an attempt to bypass the problems of unequal letter frequencies, Luhn [Luhn,1958] suggested the use of randomizing squares to represent alphabetic words. While the author is specifically concerned with representing words which have already been reduced via various coding methods (Significant Letter Code and Eliminate and Count Code Words) his process is in some sense an early form of hashing and could be used to code English words directly.

The objective is to group letters so that the frequencies of each group are roughly equal. These groups are then used as the indexes to a matrix and bigrams, two-letter pairs, are encoded by choosing the appropriate rows and columns. Figure 2.2 illustrates this.

	A	B	D	E	H	J	K	L
	Q	C	I	F	N	O	R	M
	S	U	G	P	V	T	Y	W
	Z			X				
ABD								
CGOZ					x			
ENK	x						x	
FLY								
HMP				x				
IJSX						x		
RVW								
TQU				x				

<Randomizing Matrix - Fig. 2.2>

In the above example CHESTER is represented by the x's given that the word is encoded by chain spelling, or redundant coding as CH,HE,ES,ST,TE,ER

Additional words would be coded in the matrix, and finally the matrix would be represented in a 1-dimensional form on the index card. The author does not attempt any analysis of overlapping or colliding codes, nor does he offer test data in support of this system.

Stiassny [Stiassny,1960] does, however, develop theoretical probabilities for false drops using Luhn's concept of randomizing squares. Like Wise's process of estimating probabilities based first on the probability of random needle punches, Stiassny first estimates the

probability that repeating the process of choosing a series of positions in the randomized square will produce a word in the vocabulary. He then estimates that the number of vocabulary words created in the field by punching w words is:

V = Vocabulary,
L = no. of characters,
m = no. of letters per word
Ze(w) = no. of patterns created in the field

$$\left[Ze(\omega)_{-\omega}\right]\frac{V}{L_m}t\omega$$

<Eq. 2.1>

From this, the number of punched "sure" patterns which are subtracted and the number of vocabulary words created in addition to the w original ones is:

$$Y(\omega) = Ze(\omega)/V/L_m$$

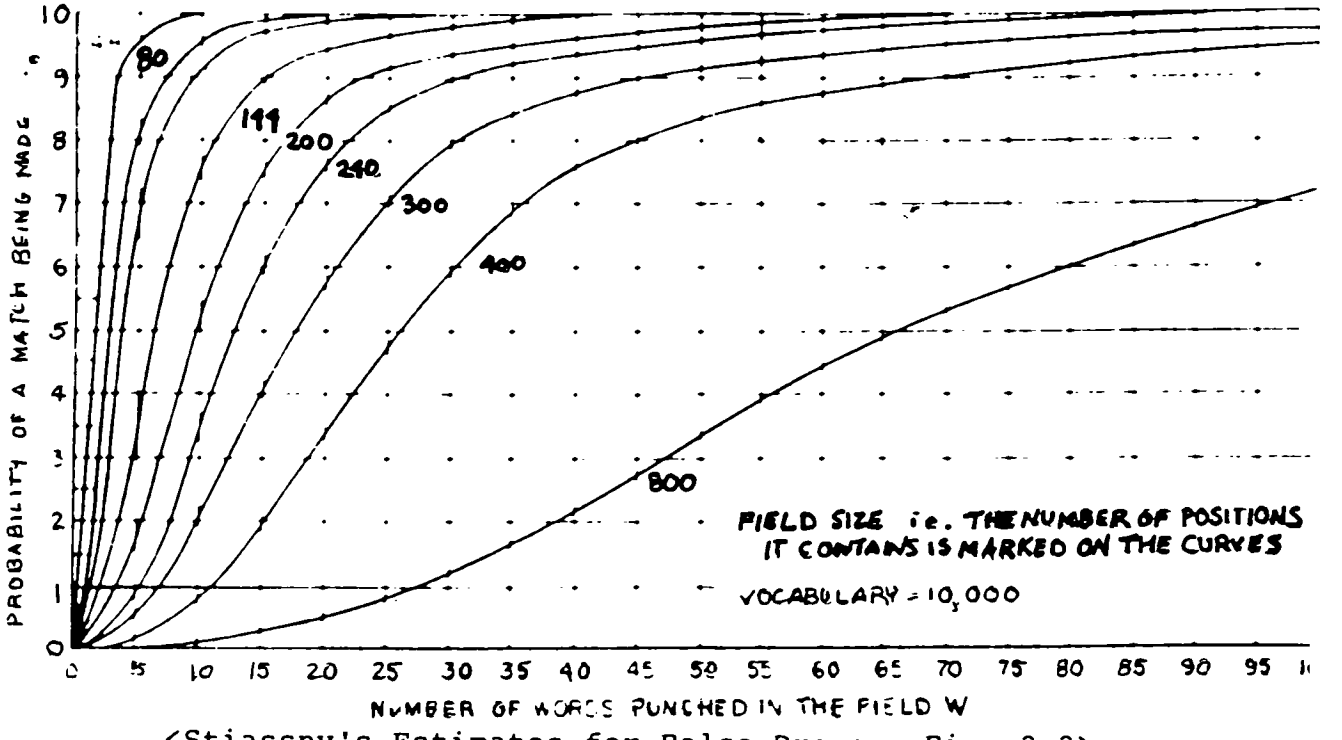
<Eq. 2.2>

and further, the probability (P(w)) of a false drop is:

$$P(\omega) = Y(\omega)/[Ye(\omega)+\omega]$$

<Eq. 2.3>

Stiassny also proves that these formulas hold approximately for chained coding (also known as redundant bigrams) with or without word-end wraparound, as well as for what he calls the Random Number method, where codes are chosen without overlap (also known as non-redundant bigrams). Figure 2.3 presents Stiassny's estimates for false drops.



<Stiassny's Estimates for False Drops - Fig. 2.2>

Stiassny determines the optimal number of punches per word for both chained and random number coding. In simple terms, he concludes that optimum results are obtained when the holes cover half the field. However, "this does not mean that the number w of words punched should be increased until half the field is punched, but it does means that for any

given w, the number m of punches per word should be increased until one-half of the field is punched."

This concept of filling half the field, or having half the coding bits set to one continues to appear in the literature of superimposed coding. The test cases, which will be discussed in the final chapter, do not necessarily support the idea that half the field must be filled, but only that performance is degraded when more than half the field is used.

2.5 FALSE DROP THEORY FOR SUPERIMPOSED CODES

Continuing the theoretical work of Wise and Stiassny, Bird [Bird,1975] derives false drop probabilities for superimposed coding. Bird's orientation and terminology also assume a punched card environment where each document is related to one card and hence one code field. The terms that Bird uses are:

a = length of codeword (typically 2-5)

n = size of codefield (typically 10-100)

i = number of coded descriptors in a document (typically 0-40)

m = average no. of coded descriptors per document (averaged over a group of documents)

If a document is indexed with a fixed level of indexing, that is, a pre-determined number of descriptors per document, then m and i should be the same. If the document is allowed to have variable numbers of descriptors, m and i are not the same.

In later discussions, the codeword length will be seen as the number of bits set per word, the size of the codefield will be given in bits, and i and m will be the number of words mapped to one signature. So despite his orientation to the card environment, Bird's results can be generalized.

Bird begins with Stiassny's idea that:

$$f = \frac{\text{average unwanted items recoved}}{\text{total items in system - wanted items recovered}}$$

After deriving several equations, he states that for practical purposes the following expressions give a fair approximation for the more complex ones. For false drops with a fixed depth of indexing and where K = the number of marked codes, and kCa is the binomial coefficient (k!/(a!(k-a)!))

$$fm = \frac{kCa - n}{nCa}$$

<Eq. 2.4>

when

$$k = n\left(1 - (1 - a/n)^{a}\right)$$

<Eq. 2.5>

For a variable number of indexing terms with a Poisson distribution, the approximate expression is:

$$f_m = \sum_{i=0}^{\infty} e^{-m} \frac{m^i}{i!} f_i$$

<Eq. 2.6>

Tabulating values for both cases for an indexing field of 26 positions, Bird presents the following false drop probabilities.

$n = 26$ (*Cont.*)

a	2		3		4		5		6	
INDEXING DEPTH	Fixed	Poisson	Fixed	Poisson	Fixed	Poisson	Fixed	Poisson	Fixed	Poisson
2	0·0110	0·0191	0·0056	0·0152	0·0032	0·0142	0·0022	0·0152	0·0018	0·0182
3	0·0307	0·0401	0·0213	0·0359	0·0169	0·0370	0·0158	0·0425	0·0169	0·0524
4	0·0570	0·0665	0·0479	0·0650	0·0463	0·0714	0·0495	0·0852	0·0600	0·1060
5	0·0883	0·0969	0·0846	0·1012	0·0899	0·1161	0·1060	0·1411	0·1337	0·1755
6	0·1233	0·1304	0·1302	0·1430	0·1457	0·1688	0·1801	0·2069	0·2300	0·2555
7	0·1603	0·1661	0·1706	0·1889	0·2118	0·2270	0·2654	0·2786	0·3369	0·3402
8	0·1987	0·2031	0·2321	0·2376	0·2819	0·2882	0·3545	0·3525	0·4436	0·4249
9	0·2406	0·2409	0·2865	0·2876	0·3537	0·3504	0·4442	0·4256	0·5424	0·5059
10	0·2761	0·2790	0·3413	0·3378	0·4242	0·4118	0·5243	0·4956	0·6297	0·5805
15	0·4713		0·5894		0·7079		0·8108		0·8875	
20	0·6277		0·7615		0·8649		0·9318		0·9688	
Vocabulary	325		2600		14950		65780		230230	

<Bird's False Drops for 26-Position Field - Fig. 2.4>

2.5.1 Conclusions

For our purposes it is worthwhile to note the closeness of the Poisson or average values to the fixed indexing values. When using running text for a signature file where the number of original words per signature varies, the case of variable levels of indexing is approximated.

For larger code sizes, Bird presents only the fixed indexing values (Figure 2.5).

$n = 100$

a	2	3	4	5	6
INDEXING DEPTH	Fixed	Fixed	Fixed	Fixed	Fixed
1	0·0000	0·0000	0·0000	0·0000	0·0000
2	0·0008	0·0001	0·0000	0·0000	0·0000
3	0·0023	0·0005	0·0001	0·0000	0·0000
4	0·0045	0·0012	0·0004	0·0001	0·0001
5	0·0074	0·0025	0·0009	0·0004	0·0002
6	0·0108	0·004	0·0018	0·0010	0·0006
7	0·015	0·006	0·0031	0·0020	0·0014
8	0·019	0·009	0·0051	0·0035	0·0028
9	0·024	0·012	0·0077	0·0058	0·0049
10	0·030	0·016	0·011	0·009	0·008
15	0·063	0·047	0·040	0·041	0·044
20	0·104	0·091	0·092	0·103	0·120
Vocabulary	4950	$1·62 \times 10^5$	$3·92 \times 10^6$	$7·53 \times 10^7$	$1·19 \times 10^9$

<Bird's False Drops for 100-Position Field - Fig. 2.5>

Bird concludes that increasing the codefield size always has a beneficial effect upon the false dropping fraction, as does decreasing the average depth of indexing. The length of the codeword (the number of codes per descriptor), however, can be increased up to a certain value before the false

dropping fraction deteriorates.

The test cases discussed in the final chapter all reached a plateau of false drop rates beyond which increasing the codefield size did not decrease the false drop rate.

2.6 BLOOM FILTER FOR A HYPHENATION DICTIONARY

In almost a separate direction, and without reference to any of the research cited here, the "Bloom Filter" concept was proposed by Burton Bloom [Bloom,1971]. Without specifically referring to superimposed coding, Bloom suggests superimposing hash codes into one field for applications where not finding a given item is the most likely and the most desirable result. Hence the concept of a filter. He compares this method with traditional, error-free hash methods which require a fair number of table accesses to reject an item which is not a member of the table.

The example Bloom uses is that of a hyphenation application where a dictionary of 500,000 words may have 50,000 words (10%) that cannot be hyphenated in the usual way. Such an exception dictionary could be hashed using superimposed codes, and if a query word is not found in the tionary (the most likely case) it can be safely hyphenated using the routine algorithms since there are no

false dismissals. There are, of course, false drops, so a match in the hash field is inconclusive.

Bloom's main concern, however, is not with the number of false drops, but with the average reject time. In other words, how many operations will need to be performed on average before an empty bit is found in the hash table. (When looking for a subset in a membership test, the zero bit is the sign of a non-match.) Bloom does give a false drop probability as a first step to calculating his reject time. Thus the false drop probability, or the case where all bits tested are ones is:

$$P = (1 - e)^d$$

<Eq. 2.7>

where e is the fraction of empty cells and d is the number of bits set for each message

Beginning with E, the expected fraction of bits still set to zero after n messages are stored in a hash table of N bits is:

$$\varepsilon = (1 - d/N)^N$$

<Eq. 2.8>

Bloom then derives the relationship for the normalized time measure T when x bits will be tested and the first x-1 bits are 1 and the xth bit is zero. (The time for a reject.)

$$T = \sum_{k=1}^{\infty} x.e(1-e)^{x-1} = \frac{1}{e}$$

<Eq. 2.9>

The following table (Figure 2.6) clearly shows the results of these equations when applied to the hyphenation case mentioned earlier. The comparison implied with the "percentage of disk accesses saved" column is with a conventional hash table in non-resident memory. In the latter case, each table access would involve a disk access. What Bloom then is comparing is the number of accesses in order to reject a word, as well as the effort involved in making them. (The formulas for the reject time for a conventional hash table lookup have not been reported here, but are roughly equal to the worst case insertion rate in a hash table.)

TABLE I. SUMMARY OF EXPECTED PERFORMANCE OF HYPHENATION APPLICATION OF HASH CODING USING METHOD 2 FOR VARIOUS VALUES OF ALLOWABLE FRACTION OF ERRORS

P = Allowable Fraction of Errors	N = Size of Hash Area (Bits)	Disk Accesses Saved
1/2	72,800	45.0%
1/4	145,600	67.5%
1/8	218,400	78.7%
1/16	291,200	84.4%
1/32	364,000	87.2%
1/64	509,800	88.5%

<Bloom Filter Disk Accesses - Fig. 2.6>

2.7 PEEK-A-BOO CARDS

In the suggested card applications presented so far, the document has been represented by one card with the appropriate codes or descriptors encoded on them. In the Peek-A-Boo System [Uhlmann,1964], this is reversed. Each descriptor or code term is represented by a card and the documents are represented by holes on the card. For example, if document number 11 has codes for positions 2, 35, 43 and 72 of a randomized square, then position number 11 is punched on these four cards. This reversal then is a direct analogy of the randomizing square where each card is the equivalent of one square, even though the square is represented in one dimension.

While there are limitations on the number of documents which can be encoded with such a system, the resulting cards can be easily manipulated by hand. Beginning with the first term in the query, all cards for the appropriate terms are held up to a light source and all holes which are transparent all the way through represent the documents for the query terms. (Conversely, cards which block any particular hole are not matches and are discarded.)

Uhlmann's concern in designing the Peek-A-Boo system is in determining the number of descriptors which can be assigned to any document, given a randomizing square with a set number of code positions and the number of optimal punch

codes per descriptor word. He arbitrarily assumes the number of resulting punches should be no more than 33%. Thus for 104 unique punches (the number of positions in the randomizing square) which represents 104 cards, and for four punches per descriptor, there can be nine descriptors per document. A smaller randomizing square of only 65 positions and three punches per descriptor still results in only seven descriptors per document.

2.8 PEEK-A-BIT CARDS

One of the more successful implementations of superimposed coding was called Peek-A-Bit [Hutton,1968] in deference to its roots in Peek-A-Boo cards as well as superimposed coding. However, the documents, not the descriptors, are once again represented by the cards.

Using 33,000 subject index entries from Nuclear Science Abstracts, a working system was maintained for several years using the following coding methods for fast retrieval.

Each subject index contains an average of nine alphabetical words with a total of 60 characters. The values to be set in the superimposed field are determined by the values of the individual characters. However, the BCD (Binary Coded Decimal) values for characters were deemed wasteful since a BCD byte of six bits represents 64 characters when only 36 (26 letters and 10 digits) are

needed. The author devises his own binary character values and in so doing is careful to give representations with fewer 1 bits to more frequent characters. (Hutton notes that "in the master mask 1 bits are wanted since only the 0 bits have stopping power...while in the question mask it is the 1 bits that have penetrating power.")

Words are coded directly into the coding field using these values, with the exception that word lengths determine the positions of the codes:

words <= 6 characters	occupy bytes 1->6
words 7-12 characters	occupy bytes 7->18
words 13-18 characters	occupy bytes 13->18,1->12
words > 18 characters	occupy bytes 1->18

The phrase "Analytical Quality Control Contamination Survey," when translated into the coding field becomes:

```
                A N A L Y T I C A L
                Q U A L I T Y
                C O N T R O L
I N A T I O N           C O N T A M
S U R V E Y
```

Using their code representations creates a mask (in octal):

```
                50 10 50 06 30 40  04 11 50 06
                15 21 50 06 04 40  30
                11 20 10 40 02 20  06
04 10 50 40 04 10  10              11 20 10 40 50 24
03 21 02 44 01 30
```

with a logical sum of:

with a logical sum of:

07 31 52 44 05 30 55 31 50 46 36 60 37 31 50 46 50 24

A query mask for the single word ACTIVATION (bits 7-18) is:

00 00 00 00 00 00 50 11 40 04 44 50 40 04 20 10 00 00

The logical product (and) of this with the stored mask is:

00 00 00 00 00 51 11 40 04 04 40

The changes in the last two bytes indicate the absence of the query word, and there is no need to process the third set of bytes.

Hutton gives a sample of the performance figures: a search for the word "activation" in 618,616 entries took 29 seconds, with 8,527 hits of which 5,646 actually contained the word "activation". Unfortunately for the purposes of this review, more detailed analysis of false drop rates were not given. Hutton's own concerns were for speedy searching through the 600,000 subject terms and false drops were evidently not seen as burdensome. The system had been in existence for five years at the time of the article.

2.9 INFORMATION RETRIEVAL

Another implementation of superimposed coding, also used for subject retrieval in an information system [Files, 1969], was tested on a bibliography of computer science as well as bibliographic material taken from The Computer News Group of the IEEE. The signature files were generated by reading a word, and, if it was greater than three characters, processing it by trimming it (not necessarily resulting in a grammatically correct stem) and then checking it through a stop list of common words.

Each entry consisted of approximately 12 words with about 300 characters total. Each word after trimming generated seven different pseudo-random hashes between 1 and N and each of these different numbers was used to create one superimposed code word of 24 bits. Thus each entry of 12 English words produced seven superimposed code words (and one 32-bit address pointer).

In such a system, Files and Huskey predicted that a query of three words would have a probability of 3 x 10**-10 of matching all seven code words for each word. But the authors paid scant attention to either deriving a probability function or demonstrating it empirically. Their false drop probability is:

$$Nx(bd)^{cw.qbits}$$

<Eq. 2.10>

where Nx = no. records coded; bd = bit density; cw = code words; and qbits = no. of ones in the query word.

They suggest that optimum bit density is 1/e and that the number of bits to use for the codeword when there are M words to be coded is 2.2M. In their test case, they had 12 words x 2.2 = 24.4 bits.

Again, the authors' main concerns were not with false drops but with storage size and search speed so that appropriate comparison with an inverted file structure could be made. Their initial test file was 100,000 bytes which was coded into 3,000 bytes, since the test run produced only one codeword, not seven, for each entry. Increasing the number of codewords to seven per entry would have given a total size of 2.5 x 10**8 bytes for a 10-to-1 reduction. In comparison, an inverted file would require twelve 24-bit words (one for each entry) plus the 32-bit address pointer for a total of 4 x 10**8 bytes.

2.9.1 Conclusions

In attempting to evaluate the search speed, Files and Huskey note the inherent difficulties due to the various

system dependent functions. While the inverted file search is conceded to be faster because of the ordered nature of the inverted file, the authors suggest that the logical "AND" process of query searching could be handled as a hardware implementation and even in parallel for all seven words. Furthermore, there is a high probability that with three query words, 90% of the records will be rejected after only one comparison, in which case the next two words do not need to be compared. The test cases which will be reported in the final chapter would support this assertion. A 10% false drop is easy to achieve. Finally, they note that while the inverted file must be stored on a random access device for efficient searching, the sequential nature of superimposed coding allows the use of sequential devices without loss of efficiency.

In an appendix, the authors also suggest that the search space of superimposed codewords could be further reduced if a second level code were introduced. For example, each 24-bit code would have each three bits logically "OR"ed producing an 8-bit second level code word. Since there are 256 possible second level code words, the file could be partitioned or ordered. Searching the file would then involve first determining which partition to search. For example, a query mask of 000 010 000 010 000 001 000 000 would be "OR"ed to the 8-bit code 01010100 (84) which then

would search only those signatures with 84 as a subset (11111111, 11111110, 11111100, but not 11111011) which would eliminate all but 32 of the 256 sets. They do not implement this improvement.

2.10 TEXT EDITING

Signatures have been suggested and tested as a way to enhance text searching in word processing or text editing environments. Alan Tharp and Kuo-Chung Tai [Tharp, 1981] tested adding a 64-bit or 128-bit signature to the front of each 80-character line. The object is to reduce the search space for global search commands in the text editing process.

Based on letter frequencies, they create a table of individual letters such that each class of letters in the table occurs with equal frequency. That is, that the sum of the individual frequencies for each class is roughly equal. A table with eight classes is:

T(y)	y
0	blank character
1	E B & ' " ?
2	T X Z W G 5 ; 1 * <
3	A F Y P 4 ,) ! > ^
4	O L C . @ [
5	I K D M J 2 9 #] |
6	N V S 1 8 - $ {
	U R 0 7 + % }

<Tharp's Frequency Table for Bigrams - Fig. 2.7>

Using bigrams they implement the following hash algorithm for the pair (y1, y2):

hash(y1,y2) = 8 * T(y1) + T(y2)

(8 equals the number of classes in the table)

The pair "ab" produces a value of 8 * 3 + 1 = 25. Note that all the values from the table using the above algorithm will result in values from 0 to 63. For a larger signature size, the number of classes needs to be further divided. The authors suggest that the number of classes squared should equal the signature size. For a 128-bit signature, 11 classes were developed and since 11 squared is only 121, seven bits are left unused.

The authors tested their system using a 202-line journal article and a 862-line PL/1 program. The data was encoded using both 64- and 128-bit signatures. Search patterns were generated 1) completely randomly and 2) from merged lines of the input data which were then parsed into varying length character strings. The latter method took 1200 characters of merged data and used the first 200 characters as 100 2-character strings, the first 300 characters as the 100 3-character strings and so on. (This method was the preferred method, as the substring and strings produced were somewhat more valid than the completely random strings.)

The results expressed in percentages for each are shown in the following two charts.

SEARCH RESULTS ON THE TEXT FILE USING SUBSTRINGS AS PATTERNS

Pattern length	Lines Searched (%)		True Drops (%)		False Drops (%)	
	Eight byte signature	Sixteen byte signature	Eight byte signature	Sixteen byte signature	Eight byte signature	Sixteen byte signature
2	70.29	53.31	52.44	69.15	47.56	30.85
3	52.98	35.46	25.99	38.82	74.01	61.17
4	41.63	23.37	14.21	25.31	85.79	74.69
5	31.16	14.19	10.33	22.67	89.67	77.33
6	25.50	11.30	9.14	20.64	90.86	79.36
7	18.91	7.07	8.72	23.32	91.28	76.68
8	14.51	5.35	7.74	21.00	92.26	79.00
9	11.95	3.92	9.20	28.07	90.80	71.93
10	8.77	2.69	10.39	33.89	89.61	66.11
11	7.17	2.22	11.25	36.22	88.75	63.78
12	6.14	1.70	10.39	37.61	89.61	62.39

<Tharp's Search Results for Text File - Fig. 2.8>

SEARCH RESULTS ON THE PROGRAM FILE USING SUBSTRINGS AS PATTERNS

Pattern length	Lines Searched (%)		True Drops (%)		False Drops (%)	
	Eight byte signature	Sixteen byte signature	Eight byte signature	Sixteen byte signature	Eight byte signature	Sixteen byte signature
2	43.06	32.77	39.29	51.64	60.71	48.36
3	22.30	15.98	22.50	31.41	77.50	68.59
4	13.74	8.46	17.71	28.78	82.29	71.22
5	7.91	6.75	22.98	29.90	77.02	70.10
6	5.66	4.00	18.06	25.54	81.94	74.46
7	3.92	2.79	26.16	36.75	73.84	63.25
8	3.14	1.99	29.61	46.60	70.39	53.40
9	2.45	1.62	27.73	41.95	72.27	58.05
10	1.52	1.02	30.73	45.42	69.27	54.58
11	1.11	0.92	24.90	31.40	75.10	69.90
12	1.00	0.66	34.56	32.72	65.44	47.27

<Tharp's Search Results for Program File - Fig. 2.9>

While some of the false drop percentages, particularly for the 64-bit signature, seem high, reductions of even 50% of a file to be searched, as the worst case for 3-character

strings implies, is still worthwhile. As has been suggested by probability results, longer queries (whether expressed as 12-character strings, or as in previous examples of 3-word queries which may map to 12 or so characters) produce fewer false drops.

2.11 DICTIONARIES

One area which seems to be a particularly good application of superimposed coding is the use of spelling dictionaries. There are at least two instances of such implementations; one at AT&T, [McIlroy,1982] and one for the Yale Screen Editor, [Nix, 1981].

McIlroy recounts many of the considerations in choosing the terms (stems, affixes, prefixes, words) and size of an appropriate spelling dictionary. Of interest is the use of superimposed hash codes to store the resulting dictionary of 25,000 words in a size small enough to fit in main memory. He determines the size of the storage table by first using the following as the probability q that a bit is empty

$$q = \left(1 - \frac{1}{N}\right)^{km} \approx e^{-km/N}$$

<Eq. 2.11>

where N = the size of the table, k = the number of independent hashes; and m = the number of entries.

The optimal storage capacity of any such table is found when half the bits are one and half are zero. The probability q becomes 1/2 when

$$k = \frac{N}{m} \log 2$$

<Eq. 2.12>

When this happens the probability of a false drop is

$$(1-q)^k = 2^{-k}$$

<Eq. 2.13>

To hash 25,000 words into 400,000 bits, the optimal number of hash functions for k is 11 and the false drop is less than 1 in 2000. McIlroy may be particularly concerned with a low false drop rate because he may not retain the original dictionary but only the hashed version, leaving in that case no backup against which to check false drops.

However, McIlroy discovered that having 25,000 terms in the dictionary was not adquate and that increasing the hash table size would make it too big for main memory. At that point the author decided to use a variable length code such as Huffman code to compress the hash table values.

The same problem was encountered by Nix. But rather than increase the size of the hash table, Nix decided to accept a higher probability of false drops (which in the case of a

spelling dictionary amounts to letting a misspelled word pass.) Beginning with the concept that a 1,000 word dictionary could be hashed into 20,000 bits with 10 independent hashes (setting somewhat fewer than half the bits because of overlapping hashes) would yield a false drop probability of less than (1/2)**10. Thus, a dictionary of 34,958 words can fit in 699,160 bits.

Nix tested his false drop rates by transforming 5000 words in the manner of common spelling mistakes: 1) changing one letter; 2) inserting one new letter; 3) deleting one letter, and 4) swapping two adjacent letters. The resulting 2,208,158 words were checked against the spelling dictionary. The algorithm reported that 13,316 were in the dictionary. Of these, 11,281 of them truly were there, while 2,035 were false acceptances (or false drops). The overall false drop rate was .093%. Further examing this error rate by word length reveals a .83% rate for 4-letter words, .17% for 5-letter words, and .08% for 8-letter words. Nix deemed such error rates acceptable in this application.

2.12 PARTIAL-MATCH RETRIEVAL AND SUPERIMPOSED CODES: THEORY

Another application which has been suggested for superimposed coding is partial-match retrieval; that is, file retrieval based on key retrieval, where a query may

request a match from field/key 1 and a match from field/key 2. Roberts [Roberts,1979] is the first to suggest superimposed coding as a way to use a preselection process and narrow the search space in the partial match environment.

In Roberts' system, a file of superimposed code words replaces the keyed access file which is often part of the partial match system. In this case, the keys from each record are hashed and superimposed into one superimposed code word. The set of these codewords is stored sequentially in a record selector field in an access file. It is important that the codewords and the records they correspond with are stored in the same sequence, since the only mapping mechanism between the two files is the sequential ordering. Searching the codeword file then is similar to searching the rows of a matrix where the index to the matrix indicates a similar index value into the record file. The ultimate matching is still done by accessing the original records or files and comparing the keys to the queries.

Like many others, Roberts derives a false drop probability which he then uses for finding the optimum values for the numbers of bits per key, and the optimum number of keys per signature. First he describes the probability (p) of one bit in the signature being set as:

$$P(1) = 1 - (1 - k/b)^s$$

<Eq. 2.14>

where k = the number of bits actually set to 1 by a key, b = the number of bits in the signature or codeword, and s = the number of s-tuples of codewords (or the number of records).

To estimate the probability that t number of bits will be set to 1, Roberts uses the following approximate equation:

$$P(t) \simeq [1 - (1 - k/b)^s]^t$$

<Eq. 2.15>

This equation is valid for sufficiently large S and for $k << b$.

After mathematically proving that the optimum values occur when half the bits in the signature are set to 1, Roberts derives equations for the optimal values for k and b. For k they are:

$$k = (1/\log 2)\log(1/F)$$

<Eq. 2.16>

where F is the desired false drop rate and b is:

$$b = k/(1 - \exp(-\log 2/s))$$

<Eq. 2.17>

or by substituting for k

$$b = (1/\log 2)^2 s \log(1/F)$$

<Eq. 2.18>

Using the following notation Roberts computes values for the following table:

N = no. of records total

$\bar{D}_f$ = no. of false drops

r^i = no. of keys per record

$\hat{r}^Q$ = no. of keys per query

k = no. of bits per descriptor (key)

b = no. of bits in codeword

TABLE II
EXAMPLES OF k AND b COMPUTED FROM (3.17)

Example	N	$\bar{D}_f$	r^i	$\hat{r}^Q$	k	b
a	2048	2	4	1	10	63
b	49 152	6	7	1	13	138
c	65 536	4	8	1	14	169
d	1 048 576	4	8	1	18	217
e	1 048 576	4	8	2	9	109
f	1 048 576	32	8	1	15	181

<Roberts' Bits Per Codeword/Descriptor - Fig. 2.10>

Roberts' hashing algorithm is interesting in that he uses a pseudo-random number generator. Using the values of the letters as seed numbers to the generator, successive numbers are generated to obtain the desired number of values

for each mapping. For example, he gives the first letter in a word three bits (three hashes), the second two and the third and fourth, one. The results of the random number generator are, of course, trimmed to table size.

But what is most interesting is his suggestion as to how to store these multiple word signatures in what he refers to as a bit-slice organization. While he describes this system in terms of sets and tuples, the organization may be more easily seen as a 2-dimensional matrix. For example:

	bit0	bit1	bit2	bit3....bitn
rec1	0	0	1	1 1
rec2	1	1	0	1 0

In this way, each row represents one superimposed codeword. If there is a query mask which has 1's in positions 0,2 then the usual way to search would be to logically "AND" the query mask with row 1, row2 and so on, discarding any rows that produced zeros. However, Roberts suggests that it would also be possible to process the array by columns rather than rows. That is, knowing that matches in the 0 and 2 columns are required means just those columns are searched for ones. This way is, of course, feasible only if the file has been stored in such a way that the bits are accessible by column, which in a sense may imply storing the matrix with the coordinates reversed.

While Roberts does implement a test database of 47,252 records from the Suffolk telephone directory, he does not present elaborate results. He does not give false drop rates, but rather he calculates the average weights, or number of bits set, for each superimposed code word. Code words were 143 bits and the average number of bits set was 71.

As will be seen in the test cases in the final chapter, the average number of bits set by a hash codes is of primary importance in determining performance. Roberts is the first to discuss this.

2.13 PARTIAL-MATCH QUIK SYSTEM

Based on the theoretical work of Roberts, one of the most well-developed and potentially useful applications was designed at Bell Labs by J.D. Gabbe, et al. [Gabbe,1978]. Continuing the concept of partial match retrieval for a telephone directory, an application called QUIK was written.

In the QUIK System, approximately 60,000 words of business and professional listings from a telephone book are encoded in a superimposed code file. Each record consists of several keys such as the name, address, or town, and the superimposed code is built from hashes from individual keys. The file consists of one superimposed code word per record

with a fixed number of bits used for each descriptor, except that a very few keys are given fewer, or zero bits because of their excessively high frequency.

The authors do not give the actual size of their superimposed code word, nor do they state the number of bits set by any particular key, and they do not present the actual false drop rates generated by the retrieval system. One may assume that since their work is based on the work of Roberts, that the approximate relationships he suggests hold for their work.

The QUIK system itself is an interactive one where a user may enter a query such as "Name = Mary, Street = Main." The query mask is derived in a fashion similar to the superimposed code word and the results are logically "AND"ed. The resulting set of records is then string-searched for exact matches. It is presumably in this stage that limitations to particular fields are checked. Since there is an implicit nesting involved in the coding field (JO is contained in the signature JOHN) the retrieval process allows for truncation. This truncation implies that the retrieval mask is not required to have a "full set" of bits, but only as many bits as are set by the individual letter, phrases or bigrams. So, the query "name = YA BR U" yields the "Yankee Used Lumber and Bricks" company.

Since the system is interactive, the user is given the

total number of drops before the final string searching phase. Thus, if the initial query drops too large a set, the user can stop the process.

In addition to the telephone system, the authors implemented an office filing system. Each document in the system is given a cover record with key fields such as Date, To, From, Third Parties, Document type, Keywords, Abstract and a filing location. Each document corresponds to one record and one superimposed code word. Approximately 2000 documents have been put into the system, which had been in use for one year at the time of the article's publication.

2.14 CONCLUSIONS FOR SUPERIMPOSED CODING THEORY

It is apparent that each of the authors reviewed has individual purposes for using superimposed coding and has a different assessment of the cost/benefit of superimposed coding. Their various objectives were to fit an access file in core memory, to reduce a search space, to produce a reject filter, to compete with inverted files, or to provide fast disk accesses to data. In deriving equations for false drops, their ultimate purpose may have been to measure some other performance aspect of superimposed coding. Furthermore, it is clear that the implementations, or theoretical bases for the false drop figures presented are different. Therefore, the following summary figures should

be viewed only as rough comparisons. But even given the disparity of the measures used, it appears that a false drop rate of 1% can be achieved.

False drop rates	words-documents	Field Size	Hashes/bits in word
Mitre,			
less than 1 in 2000	25,000	400,000	11
Sit			
.692	24,953	599,167	10
Bird			
ranges-fixed indexing			
.0045 -- .001	4	100	2--6
.164 -- .120	20	100	2--5
Tharp			
percent of retrieved %			substrings searched
48.36 of 22.77%	80 characters	128 bits	3 -- 12
47.27 of .66%			
50.71 of 43.0c%	80 characters	64 bits	3 -- 12
47.27 of 1.00%			
Files			
1 word query			
1 / 10 ** -12	12 words(200 char)	24 bits	
Hutton			
(1 query tested)			
8527-oldest retrieved	80 characters	12 bits	each character
8527-5646 false drops			variable
Stiassny			
estimates			
1 -- 7	25 -- 100	311	variable with
8 -- > 10	7 -- 16		word length

Chapter 3. RECENT WORK ON SIGNATURE FILES

Two researchers at the University of Toronto, Chris Faloutsos and Stavros Christodoulakis have extended the concepts of superimposed coding as we have seen them so far. They introduce the term "signature file" to cover both the concept of superimposed coding as well as their own concept of concatenated word signatures. Both of these methodologies have been covered in the introduction. Furthermore they extend the use of signature files for larger files and vocabularies than have been used so far. They propose signature files for full text applications rather than for index descriptors only.

3.1 False Drop Probabilities

Like previous authors, Christodoulakis and Faloutsos [Faloutsos, 1984], provide analytical proofs for false drop probabilities. They cover both superimposed coding techniques and word signatures in their formulations. Because they intend to cover files of a size which would require more than one signature to represent it, they have also chosen to break up their file into blocks. Each block represents a number of words to encode into one signature. Given the following notation:

Symbol	Definition
F	Signature size in bits
Dbl	Distinct, noncommon Words per logical Block
V	Vocabulary size: total number of distinct noncommon words in the database
M	Number of logical blocks in the database
aj	Number of blocks that the jth word appears in ("selectivity" of the word)
a	Average number of appearances per word
m	Number of bits that each word sets to "1"
f	Number of bits in a single "word signature"
Smax	Maximum possible number of distinct word signatures

Faloutsos makes the following assumptions:

1) Large number of possible signatures: Smax >> Dbl
2) Large vocabulary: V >> Dbl
3) Large database: M >> 1
4) Small selectivities: M >> aj
   (This assumption is based on using a stop list of common words)
5) Occurrence frequencies follow the geometric distribution
6) Randomly selected words appear independently of each other in the blocks of the database.

Such assumptions can be represented in the following "typical values"

V	10,000	
F	600	
M	10,000	
Dbl	40	
av	40	(=MDbl/V)
m	10	(= F ln 2/Dbl)
f	15	(= F/Dbl)
Smax	32,768	(=2**(F/Dbl)

<Typical Values for Faloutsos' Files/Vocabularies - Fig. 3.

Using the case of the unsuccessful search (the word is not in the database so that all drops are false drops), the authors derive the following:

1) False drop probability for word signatures is:

$$\text{False Drop} \approx 1 - \left[1 - \frac{1}{Smax}\right]^{Dbl}$$

<Eq. 3.1>

$$\ln \text{False Drop} \approx \ln Dbl - \frac{F}{Dbl} \ln 2$$

<Eq. 3.2>

2) False drop probability for superimposed coding is:

$$\text{False Drop} = \left[1 - e^{-(m\, Dbl / F)}\right]^{m}$$

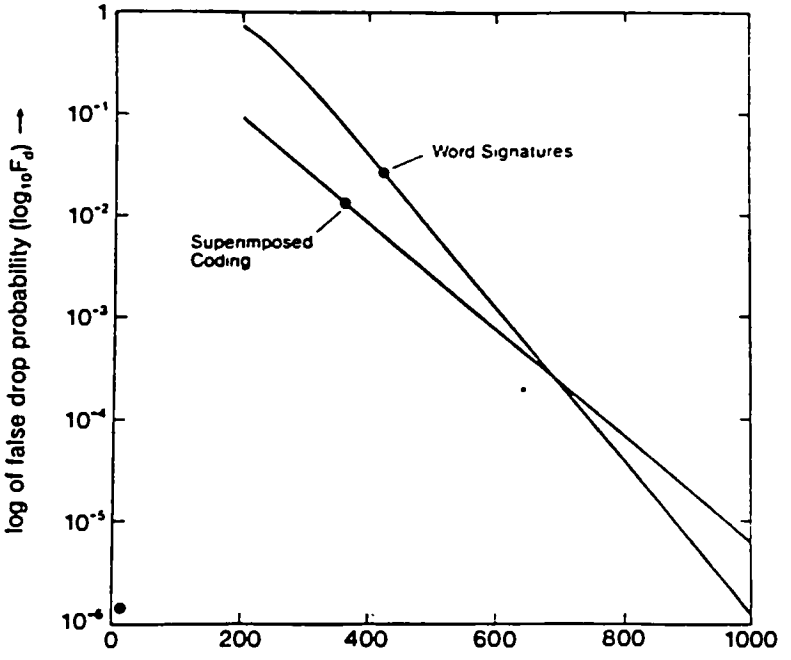
<Eq. 3.3>

In the process of deriving these equations, the authors also assert that the case for the unsuccessful search:

1) does not depend on the vocabulary size;
2) does not depend on the size of the database;
3) does not depend on the occurrence frequencies of words;
4) is not affected by word interdependencies.

3.1.1 Testing

Using these equations, the authors could vary either the signature size or the number of logical blocks to make analytical comparisons of the two systems. The authors test these formulas using a 3.3 mbyte database of bibliographic information, and using hashing based on trigrams. Varying the signature size and holding the logical block (Dbl) to 40 words gives the following chart (Figure 3.2).



<Faloutsos' Superimposed Coding Vs. Word Signatures - Fig. 3.

3.2 Additional Signature Files Methods

In a later paper, Chris Faloutsos [Faloutsos, 1985] extends signature file methodologies. In addition to the two methods already analyzed, he proposes two more methods based on compression of sparse vectors.

In the first of these, which he refers to as "bit compression," a large vector of bits is used. Each word sets n positions in the vector. These vectors are then logically "OR"ed as in usual superimposed coding methods. However, the difference is that n is not chosen explicitly to set 50% of the bits in the signature. And the size of the bit vector is larger than a typical field. Thus, the resultant vector is sparse. For the bit block compression method, each signature is divided into parts:

Part 1 is one bit long and it indicates whether there are any 1's in the bit-block. If there are no bits the signature stops there.

Part 2 indicates the number of 1's in the bit block. The number given is the number of 1's minus 1.

Part 3 gives the offsets of the 1's from the beginning of the bit block. Figure 3.3 shows four "sparse" bit vectors of four bits each:

Sparse Vector	0000	1001	0100	1000
Part I	0	1	1	1
Part II		10	0	0
Part III		00 11	10	00

<Sparse Bit Vector - Fig. 3.3>

These parts can then be represented by either putting all the parts for one block together with the resulting vector as 0 | 1100011 | 1010 | 1000. Or, the final signature

can be assembled from the parts such that all Part 1's are together followed by all Part 2's and all Part 3's. The resulting vector would be 0111 | 1000 | 00111000. In processing these sparse vectors, methods to handle variable length fields would be necessary to determine the differences between the various portions of the vector.

A second method for compressing the bit vector is based on run-length encoding where the number of zeros between two successive 1's is recorded. While this method generally results in better compression, it does result in slower searching since a search for a bit in the nth position requires that all the bits before n be decoded.

3.3 CONCLUSIONS

Faloutsos compares all four signature file methods and concludes:

1) The fast method for searching a signature is superimposed coding. The bit compression method typically requires more comparisons, and also requires additional calculations in order to determine the lengths of the various parts. The run-length encoding method must decode, on average, half of the signature, and therefore provides slower searching. The word signature method requires that the entire block be examined but does not need any decoding or calculations.

2) All the methods do well on conjunctive, multiple-word queries.

3) Superimposed coding is the only method that can handle queries on substrings.

4) Only the word signature method preserves the order of the words in the original document.

Given these differences, he concludes that no one method is clearly preferable. To justify the complexity and extra processing involved with either bit block compression or run length encoding, one would need to demonstrate that the false drop rate can be appreciably lowered by creating sparse vectors. The test results provided in the final chapter do not indicate that there is a hashing algorithm which can actually be optimized by use of a sparse vector. The problems, which are inherent in hashing on English words, are discussed in the next chapter.

CHAPTER 4. TEXT HASHING

In common usage, hashing is the simple transformation of a given key, be it alphabetic, numeric, or alphanumeric, into a table address where the key and its information are stored. While the hashing used in this thesis is intended more as a way to generate a "code" than a table address, the issues involved with hashing remain the same.

There are several considerations which affect the performance of hashing algorithms. For example:

1) the number of table addresses compared to the number of keys to be hashed,

2) a "loading factor," which represents the percentage of the table that can be filled before performance is degraded,

3) "collisions" the number of different keys that generate the same table address,

4) the methods to resolve collisions, or "rehashing"

Most important is the degree of randomness of the keys or phrases which are passed to the hash function for computation into the actual addresses into the hash table. In this respect, English language text is not a likely candidate for good hashing. The frequency of use of English characters is far from even.

The following review presents several approaches to the problems of hashing text. All of the methods attempt to compensate for the uneven, nonrandom distribution of English. The approach generally is to break up English words and text into units such as bigrams, trigram, or n-grams of variable length. In this case "gram" simply means character so that a bigram (sometimes called a digram) means a two character unit.

Fortunately, the simplest methods actually produce the best results. Generally trigrams perform the best, followed by bigrams or other simple hashing means. Variable length n-grams seem fine in theory, but their performance hardly justifies the overhead necessary in processing them.

4.1 CHARACTERISTICS OF ENGLISH TEXT

Several studies of letter distribution of English text have been reported. While the results have strong correlations, the distributions vary with the text under analysis. For example, Bourne and Ford [Bourne, 1961] report a frequency distribution which ranks letters for English language words:

E I R O A T N S L C P M D U H G Y B F V K W X Z J Q

while for names they find:

E A R N L O I S T H D M C B G U W Y J K P F V Z X Q

On the other hand Lynch [Lynch, 1977] cites a ranking based on title words from the INSPEC database (the bibliographic database of the IEEE) as:

E I T O A N R S C L D F H M U P G Y B V W X K Q Z J

There are examples of yet more frequency distributions, some taking into account the position of letters in words (first or second character), while others are based on the differences between normal English text and subject files. Therefore, simple text hashing algorithms based on the integer representations of individual characters, as most computerized algorithms are, will be biased in favor more frequently occuring letters.

Statistical analyses such as the ones mentioned above, were used by Shannon [Shannon, 1950] in formulating a theory on the prediction and entropy of English text. Most of Shannon's work is expressed in mathematical terms. However, a non-mathematical example conveys the general direction of Shannon's work. In one experiment a subject was asked to guess, letter by letter, the characters of a sentence. If the guess was wrong, the subject was then given the correct letter. As might be expected, most of the wrong guesses were initial letters of words. But as might not have been anticipated, there were more correct than incorrect guesses. In fact, the total number of correct guesses was 69%. It would appear that a speaker of English language needs few

clues to unravel the information content of any given message.

This result suggests that a large percentage of English characters, contained in ordinary text, are redundant. That is, many characters convey little if any significant information. The elimination, or compensation for these non-information bearing characters (characters with perhaps the highest frequency) is useful not only in creating smaller codes for words, but also in distributing hashed words more randomly in a hash table or code field.

Various attempts have been made to do this. The motivation has rarely been to provide evenly distributed hash keys, but more often to increase the information density in a coding field, or to reduce the storage requirements for text. Historically, the number of characters allowed on a Hollerith card, or in the limited memories of early generation computers, required the elimination of any superfluous information. Yet the characters or codes which remained needed to be as unique as the original word. They needed to convey all the information of the original without all of the characters.

4.2 CODES FOR ENGLISH WORDS

An example of such attempts can be found in Brace's work [Brace, 1963] where a code is developed for book invoices

based on book titles. The code used the first and third letters of the first word (the second letter of a word has been deemed to have little informational value), and the first letters of the next two significant words if any. The code was designed to be as simple as possible so that punch card operators would easily remember and use it. Codes looked like:

Sacraments	SC
Sacred Wood	SCW
Saga of the Sergeant	SGS
Sailing around the World	SIAW

Even though there were a possible 26**4 (456,976) code words for the estimated half million books in existence, there were still a significant number of code collisions. For the 131 book titles beginning with the letter A there were nine collisions. (Brace resolved these collisions by coding additional information based on the author, publisher, and subtitle of the book.) It should be noted that this system, in ignoring common words, had already been weighted to avoid some of the less significant or non-information bearing characters and words of the English

At about the same time that such coding systems were being sought for use by punch card operators, machines were

beginning to be used to abbreviate words. Bourne and Ford [Bourne, 1961] review and experiment with several such methods. Some of the constraints or objectives of their study were that:

"1) Each word should be coded to require as little storage as possible.

2) Each word should retain the same degree of discrimination and uniqueness that it had in the original sample.

3) If possible, each word should retain some mnemonic similarity to the original word.

4) The procedure should not rely on any prior knowledge of the population of words which must be abbreviated."

The methodologies reviewed were:

1) Simple truncation at the right

2) Simple truncation at the left

3) Elimination of vowels

4) Selective drop-out by letter position. Starting at the left, drop out every nth letter.

5) Selective drop-out by a single ranking of letter usage. Given a ranking of character frequency (independent of position within the word), eliminate the most common letters until the desired code length is achieved.

6) Selective drop out by separate rankings of character usage for each letter position. Using character frequency rankings which are calculated by letter position, eliminate the most common letters until the desired code length is achieved.

7) Selective drop-out by a single ranking of bigrams. Using bigram rankings eliminate the most common bigrams from the word.

8) Truncation after elimination of the second character.

9) The use of a check digit or letter in combination with either simple truncation or selective drop-out of every second letter. For example, while COMPUTERS and COMPUTATION both truncate to COMPUT, if a=1, b=2,..z=26 and a check digit is added for length, then COMPUTERS = COMPUTC and COMPUTATION = COMPUTE.

10) The SOUNDEX code, a phonetically based code which generates a code by the following algorithm:

a) Retain the first letter
b) Drop out A,E,I,O,U,Y,W and H
c) Assign a number to similar sounding sets
    B,F,P,V = 1
    C,G,J,K,Q,S,X,Z = 2
    D,T, = 3
    L = 4
    M,N = 5
    R = 6
    Insufficient consonants = 0

In this way Darington = D645. While this code does not meet at least two of the authors' original objectives (it does not look like the original word, and it does not necessarily have the same discrimination value as the original word) it is a code which has been used successfully in several commercial systems to answer queries where the correct spelling of the query has been in doubt. The phonetic nature of the code causes words to be grouped in phonetic groups. Of the nine methods directly studied by Bourne, et al. (they did not test SOUNDEX), those schemes which produced the best results were those based on the statistical analysis of words. That is, those which dropped out letters based on frequency distributions, or even on statistically insignificant character positions, were deemed best. The one method which performed the best for code fields of size 3-6 characters was to omit every second character while adding a check digit based on the length of the word. Other selective dropout techinques, whether involving frequency distributions or not, were the next best methods, with the size of the coding field affecting the differences between them.

4.3 BIGRAMS AND TRIGRAMS IN CODES AND HASHING

At about the time machine abbreviations of words and stemming algorithms were being implemented, the concept of

using the binary representation of the character data as integer data was emerging. While Bourne and Ford mentioned the possibility of squaring the word as a way to generate a code, they did not pursue the concept. However, hash tables were being widely used for compiler message tables, symbol tables and other internally used tables.

In an attempt to address problems of linear searching through a file, as opposed to using inverted indexes, Willett [Willett, 1979] proposes extending such system hash table concepts to larger vocabularies.

Several hash algorithms were suggested:

1) Treating the word (eight characters, padded if necessary) as a double-length integer and dividing by table size.

2) Using the 8-character word and successively exclusive "OR"ing the four bigrams found in the word. The n least significant bits of the result are then used as the code.

3) Treating each half of the 8-character word as an integer. Multiplying these two integers then taking the 12 middle bits of the double length product, and finally using from these 12 bits the n least significant bits as the code.

Evaluating these algorithms, it appeared that the first algorithm provided somewhat better performance. Furthermore, given the ability to vary the table size by any prime

number, as opposed to being limited to a table size which is a power of 2 (as in cases 2 and 3), Willett recommends the first method, using simple division hashing.

In addition, Willett evaluated the use of redundant and non-redundant coding for both bigram and trigram divisions of words. Redundant coding by bigrams or trigrams involves splitting the word into overlapping segments. For instance, "example" would be ex,xa,am,mp,pl,le or exa,xam,amp,mpl,ple. How one treats the initial and final segments also varies. For example, "example" can be coded with an initial unit of *e (where * marks a blank). However, the bigrams and trigrams used here did not include such units. The following results were obtained from a set of 517 words which had been stemmed.

Willett concluded that the rates for trigrams were superior to those for bigrams. Using trigrams, words tended to have fewer common codes; that is they overlapped less, than for bigrams. The following table (Figure 4.1) shows these results on a set of 517 stemmed words.

OVERLAP FIGURES FOR N-GRAM ENCODING (N = 2 or 3)
OF TITLES USING BINARY DOCUMENT REPRESENTATIVES

Type of coding	Overlap	Type of Coding	Overlap
2RA	0.629	3RA	0.580
2RB	0.630	3RB	0.554
2NA	0.610	3NA	0.572
2NB	0.599	3NB	0.556

R=redundant, N=non-redundant, A=space-starting keys included and B=space-starting keys not included in the key set

<Bigram vs. Trigram Coding - Fig. 4.1>

It is not surprising that bigrams seem to be used in text hashing or compression only when there is a need to have a smaller bound on the number of possible symbols. There are 26**2 (676) bigrams as opposed to 26**3 (17,576) trigrams. Bigrams have been used in hashing associated with data communication [Lynch, 1977] or natural language understanding [Edwards, 1964]. In addition, bigrams form the basis for randomizing squares [Luhn, 1958]

Trigrams were used effectively in a spelling correction program developed by Kohonen [Kohonen, 1978]. One of the basic methods for handling text corrections is to check for words in a dictionary and if they are not found, then choose, based on some measure, the word closest to the misspelled word. The method proposed by Kohonen provided a way to speed up the dictionary look up as well as a distance measure for the isolated word.

Using redundant trigram hashing (and in this case the redundancy also involved cyclical trigrams as opposed to padding the final trigrams with blanks), each dictionary word was hashed into an open addressing table. An open addressing hash scheme allows multiple entries at one address by using a list structure or chain. Each hash address contains a pointer to the actual dictionary entry and a pointer to the rest of next entry, if any. In addition, since open addressing was used, a copy of the

trigram was kept at the table address as well. (If division hashing is used, only the quotient of the trigram hash function needs to be stored, instead of the original trigram.)

At the retrieval stage, each query word is hashed by redundant trigrams, and, if all the appropriate trigrams are found at the hash addresses an exact match is made. If there is no exact match, the trigrams are used to provide a distance measure. For example, if given that Nx trigrams are found in word X and Ny trigrams are in word Y, and that Ne is the number of trigrams common to both X and Y, the distance can be stated:

$$\text{Distance}(X \text{--}> Y) = \max(Nx, Ny) - Ne. \qquad \text{<Eq. 4.1>}$$

Given several possible Y words, the one which is the closest distance to X is chosen as the probable correctly spelled word.

The test dictionaries which were hashed were from the table of most frequently used words in Dewey (Relative Frequency of English Speech Sounds, Harvard Univ. Press, 1923) and 891 surnames from World's Who's Who in Science (Marquis-Who's Who, 1968). Roughly 5000 misspelled words were used as search arguments for each dictionary. The results of these tests were compared with another common distance measure called the Lewenstein distance, a metric

which considers the number of elementary editing operations needed to change one string into another one. The trigram hashing method was almost as accurate as this established method with the significant difference that it was nearly 100 times faster. In its performance it more closely approximates the behavior of dictionary look-up methodologies without the considerable overhead of dictionary searching.

Unfortunately for the purposes of signature file hashing, the authors did not specify the average number of entries in their open addressing table, but merely commented that the collisions did not significantly degrade their average performance.

4.4 FUZZY RETRIEVAL BY TRIGRAMS

Along the same lines, but in a more generalized way and with allusions to larger issues, deHerr, et al. [deHerr,1979] used redundant trigrams to develop what he referred to as "information traces." The traces were like outlines of the original word or as the author called them, "footprints." Rather than providing table addresses for a lookup, the hashed trigrams were used to represent the word. In this case, the trigram-based information traces would be stored on a binary tree which used trigram keys as its

nodes. Consequently, words which share the most tree nodes have the greatest similarity. These traces would then provide similarity measures between documents in response to a query.

Using two programs, one to build an inverted file structure using trees and using the trigrams as database keys, and a second program, "FUZZIE," to accept character strings as queries and to produce answers ordered by descending relevance (or similarity), deHerr generated answers to appropriate queries based on a multi-language database of water resource documents. In fact, queries were able to retrieve relevant answers from documents in a language other than that used in the query. That is, an English query might retrieve Dutch documents on the topic.

Because the retrieval system continued producing answers of decreasing relevance until the user chose to stop the process, and because the relevance of answers is always subjective, deHerr did not attempt to define one answer set to any query nor did he provide a statistical evaluation of accuracy.

Of interest here is that by the use of what might be considered "false drops" to exact match queries, a fuzzy retrieval environment is created. Unfortunately for the purposes of this study, deHerr used only long query phrases, which in a sense would be like conjunctive queries of five

or six words. It would have been interesting to see how well their system performed with very short phrases or single word queries.

4.5 N-GRAMS

Bigrams and trigrams are easy units into which a word can be partitioned by even the simplest of mechanical or computing devices. Attempts to further expand the resolving power or leveling effect of trigrams by developing variable length units, n-grams, have been made by several authors.

In an attempt to reduce the number of terms in an inverted file, Clare, Cook and Lynch [Clare, 1971] decided to see if the Zipfian distribution of alphabetic and other characters in text might offer some solutions to storage issues. If a term dictionary small enough to fit into main memory could be found and used as an index to the larger inverted file, then retrieval could be significantly enhanced. Using a machine-readable tape for one edition of Chemical Titles, the authors analyzed the text for character distribution by individual characters, by digrams (bigrams), trigrams, and so on, up to pentagrams. Some of their pentagrams were: *tion, ation, *and*, s*of*, of*the, hydro, es*of. The rankings of their pentagrams from their 1000 title sample ranged from 73 to 566 occurrences.

While the distribution of pentagrams was more level than

for individual characters, the number of pentagrams or terms generated increased proportionately and thus created a term dictionary which offered only slight reduction over the number of terms in the original inverted file. Thus the authors decided to use a term dictionary based on a variety of length terms, or n-grams (also referred to as x-grams).

Choosing a threshold distribution that a given term should not occur more than 200 times, and beginning with the 2-character terms, they chose any term having a distribution over the threshold for further consideration. A term over the threshold was further divided into at least one more term. For example, if the term GE had a frequency of 249 and 200 was the threshold, then GEN, a trigram with a distribution of 155, was chosen and its distribution was subtracted from that of the term GE*, leaving it with a distribution of 94. If, for example, the threshold had been set at 100 then GEN would need to be further subdivided.

Except for the initial distribution analysis, the authors were not working with computers but rather chose the term dictionary manually. Furthermore, the computer analysis was done very simply by advancing one character at a time. This means that both their trigrams and bigrams spanned word boundaries.

The authors did not actually code their database using their suggested dictionary. Neither did they resolve whether

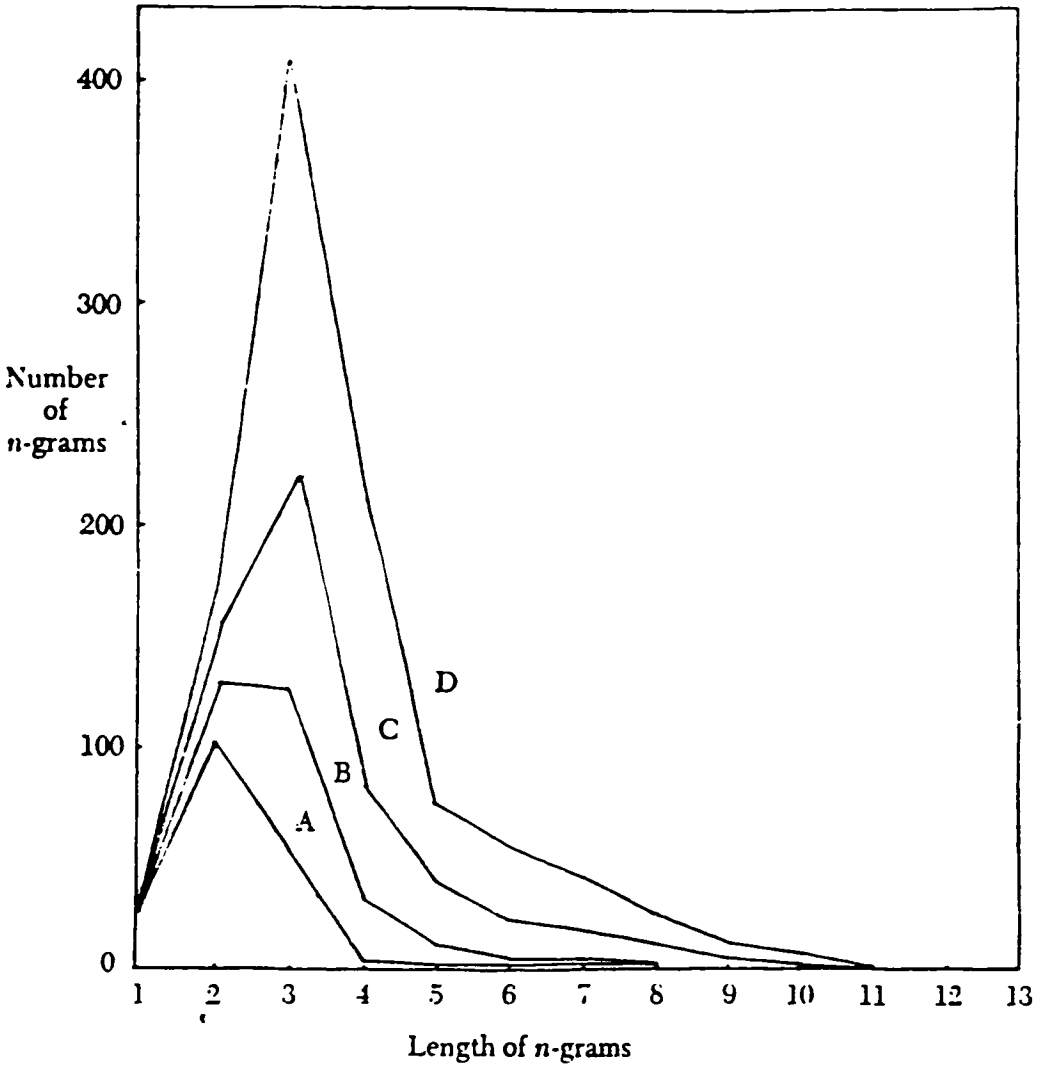
redundancy or non-redundancy was significantly better for storage compression and/or retrieval. That is, should individual terms in the dictionary overlap within individual word boundaries, and should individual terms overlap word boundaries? Since the authors' original analysis of distribution of terms and hence their term dictionary spanned word boundaries, a non-redundant coding would have traded reduced storage requirements for increased processing to match discrete, single word inquiries. However, they noted that most of the terms which spanned word boundaries, like OF*THE and S*IN* were probably of little informational value. It should be noted that the issue of redundancy vs. non-redundancy has not been clearly resolved, although there is a general tendency in creating hash-codes to use redundant codes.

Continuing in this same direction, but using instead a list of 50,000 names taken from the INSPEC database, Fokker and Lynch [Fokker, 1974] set out to find a set of variable length strings which could be used to represent authors' names. Some of the 87 keys produced (and their probabilities) are: A (.023), AL (.007), AN (.006), B (.012), BA(.013), BAR (.006), BE (.017), BO (.014)....

In addition to their previous work on n-grams, Fokkr and Lynch were strongly influenced by information theoretic work on information "infrastructures." To some extent

infrastructures echo linguistic programming work on stemming algorithms (a stem is, in effect, an infrastructure). However, the more theoretical issue resembles what has come to be called "cluster theory," only in this case applied in a micro way to the word itself. Just as a cluster matrix or threshold function of similar documents would identify documents containing significant numbers of similar or related terms, so at the word level, KLIC (Key Letter in Context) methodologies and comparative letter frequencies were seen as a way to map query words to dictionaries.

The persistent theme through this work, and following work done by Lynch, is the relationship between the frequency of the keys and the size of the key set. Lynch found that "the size of the key-set produced from a given data base can be varied arbitrarily by changing the threshold value. There is an approximately hyperbolic relation between the values of the threshold and the number of keys selected. As the size of the key set increases the length of the longest n-gram in the key-set increases, and the distribution of n-grams shifts toward higher values." This relationship can be seen in the following chart (Figure 4.2):



Key-set	size	Threshold probability
A	184	0.0025
B	332	0.0015
C	572	0.0010
D	1034	0.0007

Fig. 2. Distribution characteristics of *n*-grams generated from 10,000 surnames from INSPEC for four different threshold values

<Distribution of N-Grams - Fig. 4.2>

The authors' own particular goals were met by developing n-grams working from both the front and back of surnames.

Ultimately, they included first and second initials, proving that fixed length keys from authors names have insufficient information content in themselves. For the purposes of choosing hashing algorithms, it is interesting to note that the peaks of most of the key-set sizes were at trigrams.

In a slightly more theoretical paper, Lynch [Lynch, 1977] tackles the general concept of variable length strings with an eye to Shannon's theory that the transformation of information is greatest when the symbols of the message are equally frequent and statistically independent of one another. Once again, his concern is with the hyperbolic distribution between the number of keys in a key-set and the length of each key and its threshold of frequency or distribution. That is, that the longer the individual keys, the larger the key-set.

Studying symbol sets across several languages by using a translation of a biblical text into English, Spanish, German and French, Lynch obtained roughly the the same relationship between symbol set size and threshold probability for each language. And the same relationships held for other natural language databases, such as the Brown Corpus (The Standard Sample of Present-Day Edited American English). Thus, the concept of deriving variety-generator symbol sets is likely to work on any particular body of text or in any western language.

In addition, the emphasis of Lynch's work shifts with this paper from the desire to create a "compressed" set of dictionary items, to the creation of a "variety - generation" set of terms -- that is, a set of terms which in varying combinations can be used to generate the entire original document. In order to do this with complete confidence, the variety generator, or symbol set, must include all the original or prime characters (such as letters of the alphabet and digits) as well as derived terms. Furthermore, with an eye to computer applications, Lynch severely restricts the symbol set to 256 symbols with the result that the individual terms have higher thresholds. The following symbol set is derived from INSPEC titles.

SYMBOL SET OF SIZE 256 TAKEN FROM INSPEC TITLES

-0-	-M-	-AN-	-EL-	-IG-	-N -	-PH-	-TA-
-1-	-MF-	-AND-	-ELE-	-IL-	-N A-	-PL-	-TE-
-2-	-N-	-AR-	-EM-	-IM-	-N OF-	-PO-	-TER-
-3-	-O-	-AS-	-EN-	-IN-	-N THE -	-PRO-	-TH-
-4-	-OF-	-AT-	-ENT-	-IN -	-NA-	-Q-	-THE -
-5-	-OF THE-	-ATE-	-ER-	-ING-	-NC-	-R-	-TI-
-6-	-ON-	-ATI-	-ER -	-IO-	-ND-	-R -	-TIO-
-7-	-P-	-ATION-	-FS-	-ION-	-ND -	-RA-	-TION-
-8-	-PR-	-B-	-ES-	-ION OF-	-NE-	-RAT-	-TION
-9-	-R-	-C-	-ET-	-IONS-	-NG-	-RB-	-TIONS
-:-	-RE-	-C -	-F-	-IS-	-NI-	-RE-	-TO-
-;-	-S-	-CAL-	-F -	-IT-	-NO-	-RE -	-TO -
---	-ST-	-CE-	-F THE-	-IV-	-NS-	-RI-	-TUR-
-?-	-T-	-CE -	-FE-	-J-	-NS -	-RM-	-TY-
-.-	-THE-	-CH-	-FI-	-K-	-NT-	-RO-	-U-
-A-	-W-	-CO-	-FOR-	-L-	-O-	-RS-	-UC-
-A -	-	-CON-	-G-	-L -	-OD-	-S-	-UI-
-AND-	-%-	-CT-	-G -	-LA-	-F-	-S -	-UM-
-B-	-(-	-CTI-	-GE-	-LF-	-OF THE-	-S A-	-UM-
-C-	-)-	-CTR-	-H-	-LECT-	-OI-	-S IN-	-UR-
-CO-	-*-	-D-	-H -	-LI-	-OM-	-S OF-	-URI-
-DE-	-+-	-D -	-HA-	-LL-	-ON-	-SE-	-V-
-DI-	-,-	-DE-	-HE-	-LO-	-ON -	-SI-	-VE-
-E-	---	-DI-	-HE -	-LU-	-ON OF-	-SO	-VI-
-F-	-.-	-E-	-HI-	-M-	-ONS-	-SP-	-W-
-FOR-	-/-	-E -	-HI -	-M -	-ONS -	-SP -	-W -
-G-	-A-	-E0-	-IA-	-MA-	-OR-	-ST-	-Y-
-H-	-A -	-ES-	-IC-	-ME-	-OR -	-STA-	-Y -
-I-	-AC-	-EA-	-IC -	-MI-	-OS-	-STR-	-YS -
-IN-	-AD-	-ECT-	-ICAL-	-MO-	-OU-	-SU-	-Z-
-IN -	-AL-	-ECTR-	-IO-	-MP-	-P-	-T-	-[-
-I -	-AL -	-ED-	-IE	-N-	-PER-	-T -	-]-

<Variety Generator Symbol Sets - Fig. 4.3>

In a following paper, Cooper and Lynch [Cooper, 1981] tested the variety generator approach against simple division hashing. Using four files from the INSPEC database; a list of words was extracted, a stop-list of 37 words was generated; and, for the variety generation approach, symbol sets of various sizes (32, 64, 128 and 256 symbols) were created as in earlier papers. Codes were then generated for

the four original files using the symbols sets.

For the division hashing testing, each 16-character word (padded if necessary) was treated as an integer and divided by prime numbers to yield three distinct hash addresses.

In evaluating the comparative performance of the two methods, the authors chose a measure which accounted for the fact that in any information system not all words were likely to be the object of retrieval to the same degree. Instead, they assume that words will be sought with probabilities proportional to the frequencies with which they occur in the file. Deriving, therefore, a formula for calculating a "ratio of averages", they evaluted the "resolving powers", the ratio of the number of distinct codes produced to the number of distinct keys in the file.

For files based on abstracts (files which therefore had the highest ratio between the number of keys and the instances of keys), simple division hashing produced a resolving power average of from .99891 for 15-bit codes to 1.0000 for 24-bit codes. On the other hand, symbol set generation produced resolving power of .99467 for 15 bits and .99997 for 24 bits. While these figures are relatively close, for a file of names (which had a lower ratio between the number of keys and the instances of that key) the differences between the two approaches was greater. For symbol sets for the name file, the lowest average resolving

power was .5714, while for division hashing it was .8707.

The authors conclude that the failure of the variety generator approach to outperform the division hashing methodology is a factor of the statistical dependence between different parts of the same words.

The concept of n-grams was carried further by Schuegraf and Heaps [Schuegraf, 1974] in an attempt to find a set of equi-frequent fragments within words. Using a comparatively small database of 7418 words taken from MARC tapes (bibliographic tapes published by the Library of Congress), the authors expand on the ideas of Clare and Lynch [Lynch, 1971] of extracting equi-frequent strings. However, unlike Clare and Lynch, they are concerned that the strings not cross over word boundaries because of the limits that this might impose on the ability to retrieve single word data.

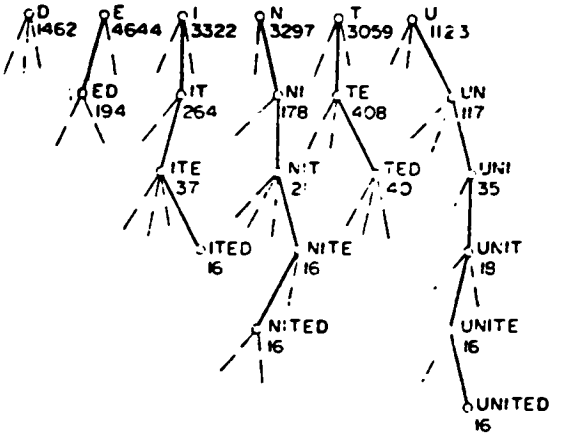
Furthermore, based on their own analysis of both word fragments and text fragments of lengths 1-8, they discovered that allowing fragments to cross word boundaries merely increased the number of fragments in each length group but had little effect on the distribution of fragments by length groups. Thus, they limit their investigations to strings which can be found within word boundaries and are non-overlapping (are not redundant). See Figure 4.4 below:

ALL SELECTED 5-CHARACTER WORD FRAGMENTS AND FREQUENCIES FOR A THRESHOLD FREQUENCY OF 10

ACHIN	11	ANDER	12	ARING	12	ATIVE	15
BIBLI	14	CATIO	10	CHANG	10	CHILD	10
CIVIL	18	COLLE	16	COMMU	10	COUNT	14
DAVID	12	EADER	11	ELECT	10	GRAMM	12
GUIDE	14	HOLOG	12	IENCE	15	INTER	26
IONAL	11	ITIES	17	ITION	12	IVERS	10
JAMES	11	LABOR	10	MATHI	10	MATIC	10
MENTA	14	NAMES	11	NNING	10	NOLOG	11
ONALD	11	POLIC	11	PROCE	12	PROGR	11
RENCE	14	RONIC	10	SCIEN	11	SHING	10
SIONS	12	SPECT	14	STATE	17	STORI	11
STORY	11	STRAT	11	STUDY	23	TEACH	15
THEOR	12	THERN	10	TICAL	19	TIONS	12
TRAIN	10	UTHER	10	UTION	11	WORLD	20

<5-Character Word Frequencies - Fig. 4.4>

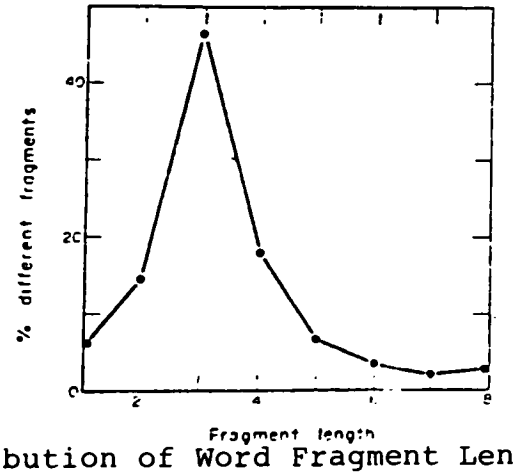
Any word can then be represented by a concatenation of the non-overlapping fragments. Turning their attention to the problem of identifying equi-frequent word fragments, and noting that enumerating all possible non-overlapping fragments and their attendant frequencies by "brute force" is an exponential impossibility, they propose a heuristic. Unlike Lynch's process described earlier, which begins with the individual letter, Schuegraf's relies on progressively selecting the largest string possible with a certain threshold of frequency from a selection forest of nodes which contain frequency information and are chained in a tree-like structure. The following chart shows such a forest.



The selection forest nodes associated with the word fragment UNITED.

<Selection Forest Nodes - Fig. 4.5>

The result of the application of this algorithm is best seen in the following chart which shows the distribution of lengths of the selected fragments. While it is of no apparent interest to the authors, it should be noted that once again trigrams account for over 40% of the character strings.



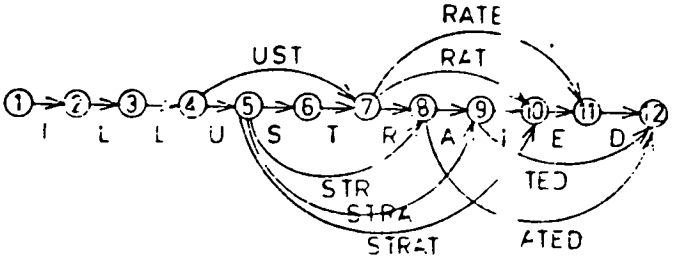
<Distribution of Word Fragment Lengths - Fig. 4.6>

Since the original goal of the authors' investigation was not to find a way to reduce the storage size of the entire database, but to reduce the size of the inverted index storage, they conclude that such an approach might

allow a searchable dictionary to be retained in core if stored in this way.

Continuing in this direction, Schuegraf and Heaps [Schuegraf, 1971] examined various algorithms for actually encoding a given database once a fragment list had been developed. Unlike the straight-forward process of using bigrams or trigrams in the order in which they appear in the text string, encoding a text string with variable length codes involves many choices. The authors devise three separate algorithms:

1) A shortest-path approach, which, given all the possible subfragments in a word, will find a shortest path as shown in the following:



Possible solutions:

I/L/L/UST/RATE/D

I/L/L/U/STRA/TED

I/L/L/U/STR/ATED

<Shortest Path Fragments - Fig. 4.7>

2) A longest-fragment-first algorithm that will accept first the longest fragment it finds in the word and

then the next longest, etc. Both of these approaches require that all subfragments of a word are known.

3) A third approach, called the longest match, begins at the left of the word and accepts the longest match it can find at a given starting position, and then moves to the next character after the match.

While the the shortest-path derivative algorithm achieves optimal database compression, the longest-match algorithm takes the least processing time to encode the database (presumably because it does not parse redundant word fragments first). The authors conclude that the compression differences among all three algorithms is minimal.

The authors also compare text fragment encoding (fragments cross word boundaries) and word fragment encoding and find that much more compression is achieved with text fragments. However, the authors caution that for some environments, such as indexing in information systems, word fragments are more appropriate because of they allow easier retrieval of discrete queries.

Finally, no discussion of n-grams would be complete without noting that many attempts to compress text data, or to store dictionaries rely on stemming words. In fact, a dictionary which includes stems (or roots) and prefixes and suffixes is itself a variety generator from which any word

can be derived. And, an issue in stemming is how to limit such generation to only "acceptable" or real words.

Schuegraf presents a very thorough discussion of such issues, [Schuegraf, 1973] as well as a thorough analysis of word types, bigrams, trigrams, and other stems. However, the use of stemming algorithms is of no particular use here, given that n-grams are generated solely by mechanistic means, while stems involve semantic decisions. It is probably true that the preprocessing necessary to determine stems will almost always increase the information content of the resulting key dictionary, as compared with fragments determined by frequency of use. However, the advantage that remains for fragments is that they do not depend on a given language or even on any particular concepts of what exactly constitutes a word. Names, proper nouns, or even initials, fit equally well in such schemes.

4.6 APPLICATIONS OF TEXT HASHING TO SPELLING DICTIONARIES

Correction of misspelled words and their attendant spelling dictionaries is an area of considerable research and to attempt more than a cursory overview here would be beside the point. However, it is interesting to notice that such dictionaries have been one of the more concrete implementations of text hashing.

At its simplest level, hashing has provided a way to speed up dictionary lookups in a large dictionary. Comer and Shen [Comer, 1982] discuss the merits of using a double hash methodology to resolve collisions versus an open addressing scheme which would put all entries of a single hash table address into an array structure for binary searching for collision resolution. In both cases the authors rely on a simple division hash methodology. Basing their evaluation on a dictionary of 16,949 words, they compared both methodologies with varying table sizes. On the average, the double-hashing performed better for the successful search, while the hash/binary search was significantly better for the unsuccessful search.

On the other hand, Radue [Radue, 1983], designed a far more complex hashing scheme. In order to avoid hash collisions he spread his hash values across a very large hash table to produce a very sparse table. However, the hash address is really a virtual address, and only part of it is used as the real address. A complicated multiply and center hashing function repetitively multipled various parts of a 32-bit word. The resulting 32-bit word was partitioned so that the middle 16 bits designated the real address, and the remaining front and back bits produced a secondary or minor key.

Radue's work closely parallels the work of this thesis

in his use of a hash dictionary. A dictionary of 2716 words was stored using this hash method. Although he is unclear as to how hash collisions were dealt with in the storing of the dictionary itself, the collisions for lookups were tabulated. Since Radue's major concern was with spelling correction, data was generated to represent the most common spelling discrepancies: one letter mistakes consisting of single letter insertion, single letter deletion, reversal of two letters and substitution of one letter. The collision rates ranged from a low of .011% to 0.033%. Radue concludes that it would be unnecessary to store the actual dictionary words in the hash table at all.

4.7 MINIMAL PERFECT HASHING

Before drawing conclusions about the nature of text hashing, it may be useful to note that another direction for text hashing coexists with those mentioned here. Hash tables for frequently used words have been derived which can perfectly match the original word into a table of the same number of entries, or at most 10% more than the original list. Such hashing heuristics are referred to as minimal perfect hashing functions [Cichelli, 1980] and are based on heuristics which seek to find an appropriate numeric value for each letter to be used in the hash function. The ideal weight of each character is determined by a back-tracking

algorithm which reevalutes hash addresses when collisions occur, until it finds values for individual characters which will allow all entries to fit without collision. As might be expected, the execution time for such algorithms is exponential, and therefore they are realistically used for fewer than 50 words. The most frequent use of such functions is for compilers' reserved words lists or system error messages. While such functions are interesting, they unfortunately are unable to offer any assistance in the kind of text hashing involved here with the storage of large databases of text.

4.8 CONCLUSIONS

Having examined the directions in which text hashing has been taken, it appears that there are a multiplicity of ways to split words into hash units and dozens of algorithms for hashing. However, no direction seems dramatically superior to all others. Certainly the concept of variable length strings is an interesting theoretical approach to the problem of unequal distribution of characters. However, the processing involved in choosing such strings as well as the extra steps involved in encoding the strings from the text seems excessive. On the other hand, trigrams do seem to perform better than bigrams with little more in processing complexity.

Complicated hash functions which would seek to compensate for uneven character distribution by multiplication and truncation, or multiply and center, seem to perform no better than those based on simple hashing.

CHAPTER 5: TEST CASES

5.1 PURPOSE OF TESTING

The focus of this thesis has been on signature file methodologies and issues as revealed in the literature. The purpose of testing is to verify that signatures, both superimposed coding signatures and concatenated word signatures, are viable and that signature files perform as presented in the literature. However, the testing presented here does not attempt to replicate either all the approaches presented in the literature or any particular author's application in detail. Nor does the testing attempt to define a new approach, a further refinement to signature files which might be a new "best way." Testing is not intended to be exhaustive.

The variables which could be manipulated in testing include: varying the size of the input vocabulary; varying the size of the signature field (either individual word signatures or superimposed fields); and using any of a half-dozen different text hashing algorithms. To test all of the permutations and combinations of these variables is beyond the intent of the thesis. After some preliminary testing (summarized below), it was decided to hold the

vocabulary constant by using the same file for all tests and to focus primarily on varying signature sizes for superimposed coding, as well as testing three basic hashing algorithms for both superimposed coding and word signatures. The objectives were to determine the relative performance of superimposed coding vs. word signatures; to determine the effect of hashing algorithms on both superimposed coding and word signatures; and to measure the size of the input vocabulary which can be accomodated.

The results of approximately 95 test cases representing averages from 400 individual tests will be presented both in summary tables and in brief discussion. The success of each set of tests is judged by the percentage of false drops obtained. As will be demonstrated, each combination of coding methodolgy, text hashing algorithm and signature size evidenced consistent behavior. And for each test program, there existed an optimum range of minimal false drop rates.

5.2 PROCEDURES AND PROGRAMS

5.2.1 General Goal

One program goal was to gather statistical information about the signature file environment as well as the false drop rates. To achieve this, each input word was hashed, with the results added to the signature as would happen in a true application; and each unique input word, and its length, was also retained in an ordered binary tree. This

allowed generation of a profile of the input text which included word lengths and duplicate words. It also allowed for the calculation of the false drops by word length.

5.2.2 Nonsense Words

In order to calculate false drop rates, it was necessary to search the signature file for words which were guaranteed not to exist in the original text. To do this, "nonsense" words were generated from the original input text. Each unique word in the input text created a nonsense word by the simple algorithm of adding one to the ASCII value of each letter in the word. Thus "a" becomes "b" and the word "the" becomes "uif." It is assumed that the combined probability of: 1) any nonsense word actually becoming another valid word, and 2) the new valid word being present in the text, would be negligible. Thus all nonsense words which were found in the signature file were counted as false drops.

5.2.3 Signature File Profile

Statistics were maintained for the total number of bits set in the signature (which for superimposed coding indicated signature density), the number of bits generated by the hashing algorithm, and in the case of superimposed coding, the number of bit collisions in the superimposition process. Examination of bits set by the hash and/or bit collisions are rough indicators of the ability of the

hashing algorithm to provide adequate random dispersal of the hash codes.

5.2.4 False Drop Counting

Each time a nonsense word was flagged as present in the signature, the false drop flag for that word was set. After searching for all nonsense words, the false drop rates were accumulated in a 1-dimensional array indexed by word length. This array was summed later for the total number of false drops for the test case. However, it was also possible to print out the array and inspect the behavior of the false drop rate as it was influenced by word length.

Well into the testing process, it was decided to monitor the false drop rates for conjunctive queries (two or three word queries). In order to count these, every time two (or three) consecutive words were found to be false drops, the counters for these conjunctive queries were incremented. Boolean flags were maintained to monitor the consecutiveness of the false drops.

5.2.5 Implementation Order

The basic program outlined below was coded first for the creation and searching of superimposed signature files. The basic program was then altered to create and search concatenated word signatures. Finally alternate hashing algorithms were swapped into the two basic programs,

resulting in eight final programs, five for superimposed coding and three for word signatures.

5.3 MAIN PROGRAM PSEUDO CODE

In pseudo code, the major operations of the test programs are:

```
Get input parameters
            (size of signature for superimposed coding;
             no. hash bits per word for fixed length hash;
             no. of words to be used in this test case)
While not EOF
     Initialize all variables
     Read a word
     If Word is not a duplicate word
           Store word on the tree
           Hash the word
           Create nonsense Word and store it with
                   original word
           Add Hash codes to signature
     End if
End While

For each node in the tree (inorder traversal)
     Hash nonsense word
     Search signature for nonsense word
     If word found
        Set false drop counter in tree node
     Check flags and counters for 2 and 3 word queries
End For

Tabulate results
      For each node in the tree (inorder traversal)
          Count and store length of words in summary array
          Count and store false drops by word length
              in summary array
      End For

     Total arrays for false drops
     Total array for unique words

     Print out all totals
     If requested, print out summary arrays and bit
         signature
```

5.4 HASH ALGORITHMS

The five hash algorithms employed are presented below in their general case format. For both trigrams and bigrams initial and final grams must use blanks for non-existent letters.

5.4.1 Simple Hash

The number of values returned by the hashing algorithm is an input parameter. If the input value is four then each of the first four characters is multiplied by a prime and by its position within the word and returned as a value. For example:

```
for i = 1 to 4
    hash[i] = (word[i] * prime * i) mod signsize
```

5.4.2 Bigram Hash

Each redundant bigram, including initial and terminating bigrams with blanks, is multiplied by a prime. Each character is weighted for its position within the bigram. The initial character in the bigram is multipled by 26 so that the bigrams "ab" and "ba" will not evaluate to the same number. The algorithm for the bigram hash is:

```
while not end of word
      hash[i] = (((word[i] * 26) + word[i +1]) *
          prime) mod signsize
     i = i + 1
```

5.4.3 Bigram Hash With Position Weight

The same basic algorithm as shown in above is used with

the exception that the value of i is added as a multiplier to weight the bigram with respect to its position in the word. In this case, the bigram "te" would produce different results in "tender" and in "attention". The algorithm becomes:

```
while not end of word
      hash[i] = (((word[i] * 26) + word[i+1])
                * prime * i)  mod signsize
     i = i + 1
end while
```

5.4.4 Trigram Hash

The principles of bigram hashing as shown above are expanded to trigrams. A simple trigram hash is then:

```
while not end of word
     hash[i] = ((((word[i] * 26 * 26) +(word[i+1] * 26)

     + word[i+2]) * prime) mod signsize
end while
```

5.4.5 Trigram Hash with Position Weight

Like the bigram hash with weight position above, the variable i is added to the equation to indicate the position of the trigram within the word.

```
while not end of word
     hash[i] = ((((word[i] * 26 * 26) + (word[i+1]
                * 26) + word[i+2]) * prime * i)
                 mod signsize
end while
```

5.5 LANGUAGE IMPLEMENTATION

The C language was chosen to implement these programs because of its built-in bit manipulation operators. For example, if a mask is declared, in C notation, as a "#define

MASK 8 0200," the 16-bit representation of that mask is "0000000000010000" in binary. This mask can be logically "OR-ed" with another 16-bit integer using the "|" function.

From this it follows that the simplest way to represent the signature as a sequence of bits is to declare an array of unsigned integers to use with masks. Thus to address bit number 183 in the signature itself, one would address bit number 7 in the 12th integer. (The number 183 divided by 16 yields 11 with a remainder of 7). Using an array indexed from 0-n, the algorithm is simple to "OR" MASK7 with signature[11] in order to set bit 183.

The rest of the programming, with its reliance on trees and arrays, would be as easily accomplished in any high-level language.

5.6 PRELIMINARY TESTS

5.6.1 Single Character Words

A preliminary series of tests were run using two different text files: 1) a copy of the proposal for this thesis and 2) a file composed of several chapters from the online Unix Manual. The file of the Unix chapters produced significantly higher false drop rates than the proposal file. The primary difference between these two test casesis the number of single character words in the Unix file. The Unix file had up to 10% of its vocabulary as single

character "words" compared to 1% for the proposal file. Many of these single character words are command arguments, such as "-r." Furthermore, all of the single character words in the Unix file were generating false drops. This is entirely reasonable when one realizes that nearly the entire alphabet was represented in the Unix file as single character words so that single character "nonsense words" generated by the above algorithm indeed were present.

By not searching for single character nonsense "words" the performance of the two files was brought nearly in line indicating that the problem was not so much storing single character words in the signature as it was searching for them in the retrieval process. Nonetheless, it was decided to use only the proposal file for all tests in order to eliminate variations in input vocabulary from test results.

5.6.2 Hashing Algorithm Optimization

The original test design called for the use of just three hashing algorithms: simple truncation, bigrams, and trigrams. From the literature it was expected that the trigram hash would perform the best, the bigram the next best, and the simple truncation hash perhaps would fail entirely. The rank order of these hashes was as expected. However, the results of the simple truncation hash were better than expected for the concatenated word signatures. This in turn led to the realization that the truncation hash

might have been performing relatively well because it preserved a sense of the ordering of the letters within the word. This suggested the concept of adding a word-position weight to both the trigram and bigram hash to see if their performance would improve. The bigram hash improved. By this point in the testing it was clear that superimposed coding fields were achieving much lower false drop rates than word signatures, so weights were added only to the suprimposed coding testing in order to look for optimum performance.

5.6.3 Conjunctive Queries

After most of the test cases had been run, a decision to investigate the performance of conjunctive queries was made. The principal reason for this testing was to see if the false drop rates were orthogonal probabilities. That is, could the false drop rate for one word be multiplied to find the false drop rates for two words. For example, if the false drop probability were .03 for one word, would the rate for two words be .0009?

The bigram hash was chosen for both superimposed coding and word signatures for this investigation. While the results for conjunctive queries for two words looked positive, limited testing was performed in this area. The primary reason for this limitation lay in the ability of the test cases used for the rest of the testing to capture accurately the entire range of performance of the

conjunctive-queries false drops. The false drop rates for 2-word queries were nearly 1% while the false drop rates for 3-word queries were so nearly zero as to be unreportable.

5.7 FINAL RESULTS

The results for each particular test case are most clearly understood by examining the summary graphs and summary data charts presented in the following pages. The overall generalizations which can be made are that signature files succeed; that superimposed coding for signature files is the overall winner; that trigram hashing produces the best results for either superimposed coding or word signatures; and, most importantly, that each test case had a set of input parameters which produced optimal results. In fact, in many cases this optimal behavior occurred over a relatively broad range of the input parameters. For example, for the superimposed weighted trigram hash, the false drop rate of about 3% could be achieved when the signature was filled anywhere between 3% and 30%. This fact means that the performance for any particular environment can be predicted and values held to optimal ones by careful regulation of input files and signature sizes.

5.7.1 Word Signature Results

The number of test cases for word signatures was limited by the necessity of having each word signature fit into a computer word or portion thereof. Varying signature

size would have been limited to only three cases: a byte, half-word, or full-word. Testing was begun using 16-bit unsigned integers for each word signature, and since the results were not encouraging, no further constriction to eight bits per word would have been logical. On the other hand, to increase the signature to a full 32-bit word would have meant that word signatures achieved insignificant compression over the orginal file. If one assumes an average of five characters per word plus a delimiting blank, storing a word in four bytes is a savings of only two bytes. It should be noted that in the test cases here, duplicate words were not stored in the signature, which would not be the case in a true application.

The variation of hashing algorithms produced some variation in the results for word signatures, but less so than for superimposed coding. The simple truncation hash for concatenated word signatures ranged from 2% to 12% false drops; bigram hashing ranged from 4% to 8%; and trigram hashing from 4% to 7%. The most obvious limit in using word signatures is the size of the total vocabulary to be hashed. As the number of unique words increases so does the number of false drops. The results of word signature false drop rates are easily seen in the following three data charts (figures 5.1 through 5.3) and are summarized in the graphs (figures 5.4 through 5.6).

TRIGRAM HASH

Word Signatures

Words Total	Unique Words	Bits by Hash	Bits set	Bits per Word	False Drop %
100	71.2	579.0	460.8	6.47	.042
200	129.2	1077.4	856.2	6.62	.051
300	173.0	1482.8	1172.4	6.77	.040
400	216.4	1837.4	1455.6	6.72	.046
500	251.8	2120.6	1683.6	6.68	.060
600	294.4	2518.8	1989.4	6.75	.043
700	333.6	2885.0	2276.5	6.82	.064
800	364.0	3153.5	2486.5	6.83	.050
900	399.0	3465.0	2729.8	6.84	.055
1000	435.0	3797.0	2983.8	6.85	.066

<Trigram Hash - Word Signatures - Fig. 5.1>

BIGRAM HASH
Word Signatures

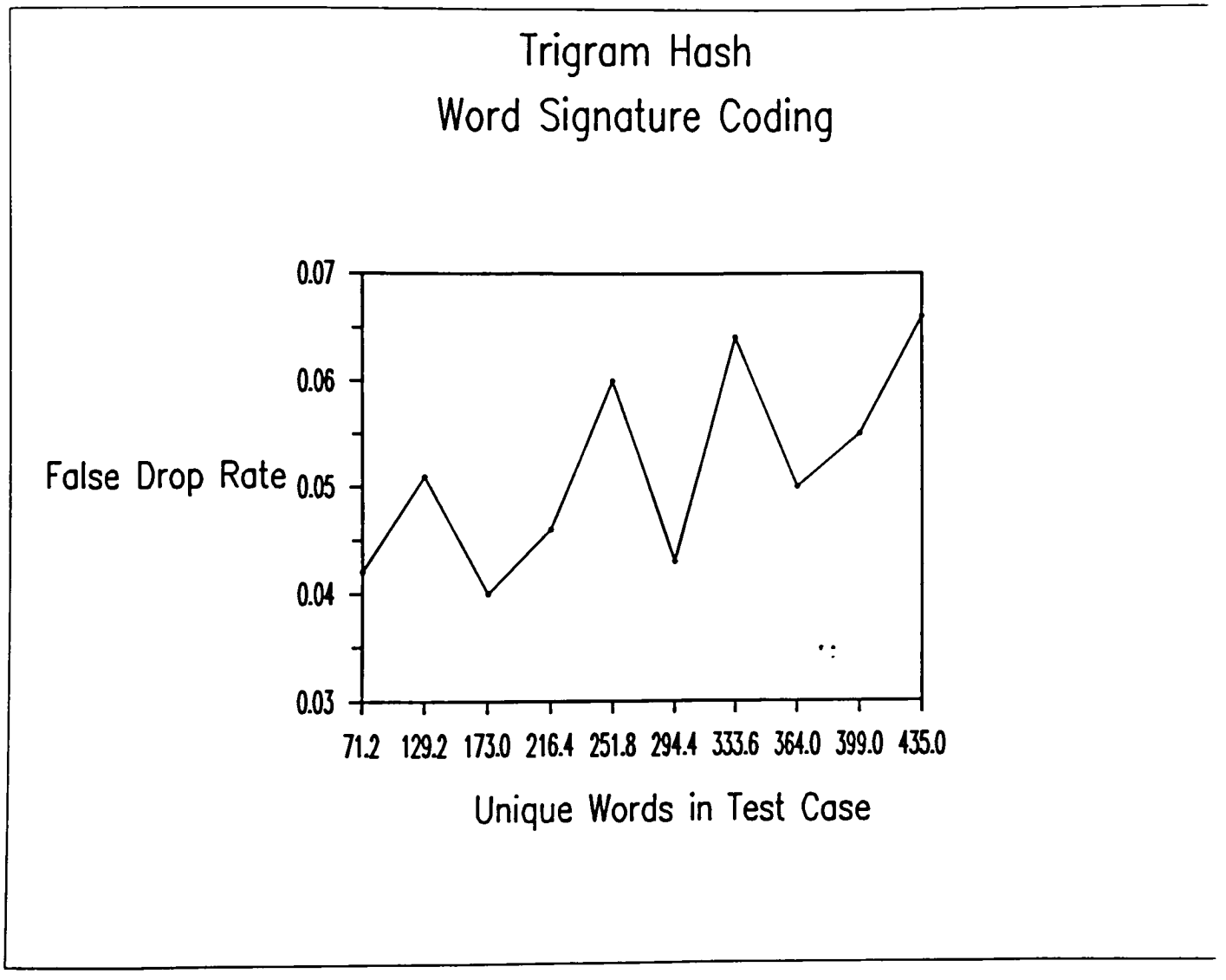
Words Total	Unique Words	Bits by Hash	Bits set	Bits per Word	False Drop %
100	71.2	497.8	404.6	5.68	.045
200	129.2	948.2	772.8	5.98	.063
300	173.0	1309.8	1057.4	6.11	.053
400	216.4	1621.0	1310.2	6.10	.057
500	251.8	1980.3	1593.3	6.32	.043
600	294.4	2224.4	1784.0	6.09	.066
700	333.6	2530.4	2039.4	6.11	.077
800	364.0	2788.7	2246.2	6.17	.072
900	399.0	2965.3	2462.3	6.17	.068
1000	435.0	2698.0	2698.0	6.20	.078

<Bigram Hash - Word Signatures - Fig. 5.2>

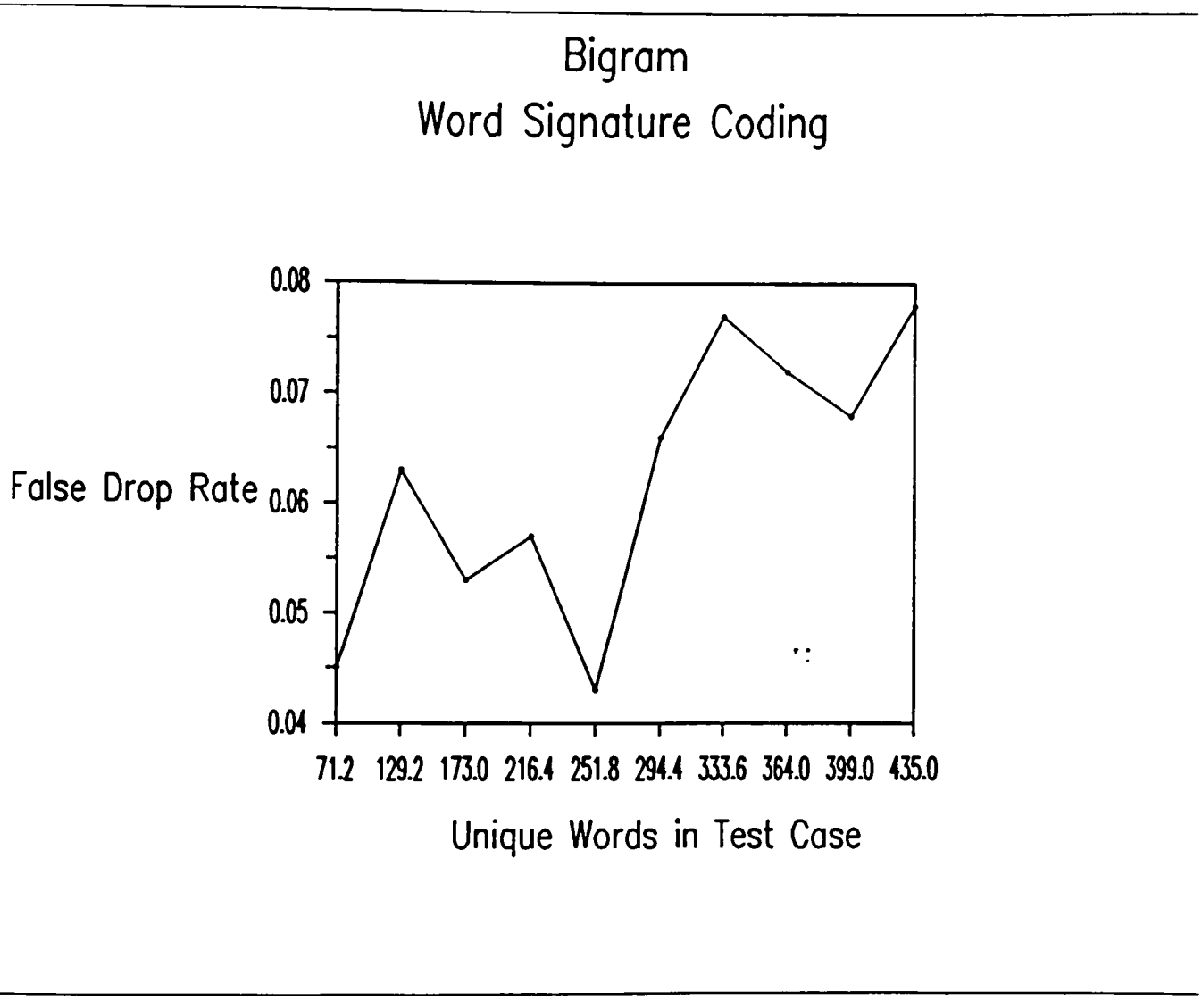
SIMPLE HASH
Word Signatures

Words Total	Unique Words	Bits by Hash	Bits set	Bits per Word	False Drop %
100	71.2	484.7	367.0	5.15	.028
200	129.2	908.2	684.0	5.29	.052
300	173.0	1218.0	945.0	5.46	.059
400	216.4	1529.5	1191.7	5.51	.073
500	251.8	1782.2	1364.8	5.42	.085
600	294.4	2026.5	1567.5	5.38	.081
700	333.6	2359.0	1825.5	5.46	.108
800	364.0	2553.0	1980.5	5.44	.094
900	399.0	2719.5	2115.5	5.30	.108
1000	435.0	3049.0	2366.0	5.43	.127

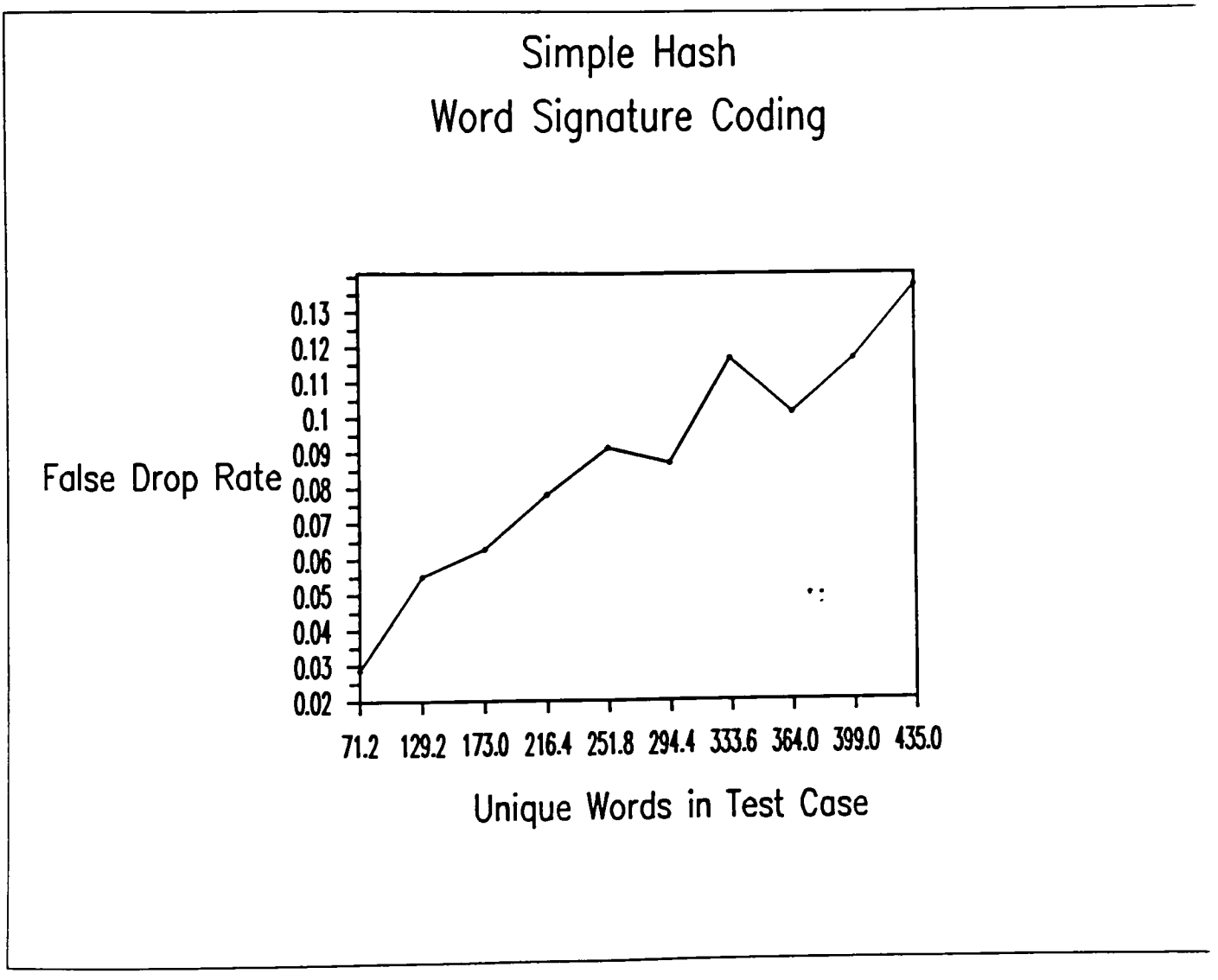
<Simple Hash - Word Signatures - Fig. 5.3>



<False Drops - - Word Signatures - Fig. 5.4>



<False Drops - Bigram Coding, Word Signatures - Fig. 5.5>



<False Drops - Simple Truncation, Word Signatures, Fig. 5.6>

5.7.2 Superimposed Coding Results

Superimposed coding provides a few more variables to be calculated and observed than word signatures. Primarily, with superimposed coding, the size of the signatures is fixed. The density, the percentage of bits set to one, changes as more words are added. With word signatures, the total signature size grows as more words are added, and the number of unique words may be a determinant of false drops. Similarly, the denser the signature, the more likely the occurrence of bit collisions and the lower the number of bits set when a new word is added. There are additional correlates to this fact, such as the denser the signature, the more likely bit collisions are to occur and the lower the number of bits which are set each time a new word is added becomes. Alternatively, the percentage of the bits attempted to be set vs. those actually set decreases. In order to observe the number of collisions, it is necessary to know how many bits are generated by the hash algorithms, both as a total and as an average per word. The following five charts (figures 5.7 through 5.11) present all of these variables for the superimposed test cases. The column headings for the data are defined as:

signature in integers:	the size of the signature array
size in bits:	the number of integers * 16
words total	the number of input words in the test case
unique words	the number of non-duplicate words in the test case
false drops	the percentage of the nonsense words found by the search n.b. .202 = 20.2%
bits by hash	the total number of hashes generated for all unique words
bits set	the number of bits which were not collisions
% of tried	bits set divided by bits generated by hash
% of sign	signature density, bits set divided by signature size
Avg per word	bits by hash divided by unique words (how many hashes per word were generated

TRIGRAM WITH POSITION VARIABLE

Signature in ints	Size in bits	words total	unique	False drop %	Bits by hash	Bits set	% of trigr	% of sign	Avg. per word	Avg set
10	160	100	72.0	.620	586.8	146.0	.24	.81	8.15	2.02
100	1600	100	72.0	.033	586.8	406.6	.69	.25	8.15	5.65
200	3200	100	72.0	.033	586.8	418.0	.71	.13	8.15	5.80
300	4800	100	72.0	.033	586.8	427.7	.73	.09	8.15	5.93
500	8000	100	72.0	.033	586.8	435.8	.74	.05	8.15	6.05
1000	16000	100	72.0	.033	586.8	443.0	.75	.03	8.15	6.15
100	1600	200	129.6	.040	1080.6	579.0	.54	.36	8.33	4.46
100	1600	300	173.8	.047	1488.0	691.6	.46	.43	8.56	3.97
100	1600	400	217.0	.050	1842.0	773.6	.42	.48	8.48	3.56
100	1600	500	250.8	.075	2123.2	835.6	.39	.52	8.46	3.33
100	1600	600	295.0	.072	2527.4	895.2	.35	.55	8.66	3.03
100	1600	700	337.5	.109	2890.2	955.7	.33	.59	8.56	2.83
100	1600	800	365.5	.102	3160.0	977.7	.31	.61	8.64	2.67
100	1600	1000	432.7	.155	3804.0	1057.0	.27	.66	8.79	2.44
200	3200	200	129.6	.037	1080.6	663.8	.61	.21	8.33	5.12
200	3200	300	173.8	.032	1488.0	804.6	.54	.25	8.56	4.63
200	3200	400	217.0	.018	1842.0	927.0	.51	.29	8.46	4.27
200	3200	500	250.8	.035	2123.2	1015.4	.48	.32	8.46	4.04

<Trigram with Position - Superimposed Coding - Fig. 5.7>

TRIGRAM WITHOUT POSITION VARIABLE

Signature in ints	Size in bits	Words total	unique	False drop %	Bits by hash	Bits set	% of tried	% of sig	Avg. per word	Avg set
20	320	100	72.0	.144	586.8	213.2	.76	.66	8.15	2.96
30	480	100	72.0	.072	586.8	253.2	.43	.53	8.15	3.51
40	640	100	72.0	.061	586.8	264.4	.45	.41	8.15	3.67
100	1600	100	72.0	.036	586.8	300.2	.51	.19	8.15	4.17
200	3200	100	72.0	.033	586.8	322.2	.55	.10	8.15	4.48
300	4800	100	72.0	.033	586.8	329.0	.56	.07	8.15	4.56
500	8000	100	72.0	.033	586.8	334.2	.57	.04	8.15	4.64
100	1600	200	129.6	.041	1090.6	446.0	.41	.28	8.33	3.44
100	1600	300	173.8	.033	1488.0	524.6	.36	.33	8.56	3.08
100	1600	400	217.0	.032	1842.0	601.2	.33	.38	8.48	2.77
100	1600	500	250.8	.043	2123.2	649.4	.31	.41	8.46	2.59
100	1600	600	295.0	.037	2527.4	716.0	.28	.44	8.66	2.42
100	1600	700	337.5	.054	2890.2	744.2	.25	.46	8.56	2.20
100	1600	800	365.5	.053	3160.0	797.0	.25	.49	8.64	2.17
100	1600	1000	432.8	.066	3804.0	854.0	.22	.53	8.78	1.97
200	3200	200	129.6	.038	1090.6	478.0	.44	.14	8.33	3.69
200	3200	300	173.8	.031	1488.0	573.2	.37	.18	8.56	3.30
200	3200	400	217.0	.028	1842.0	664.8	.36	.20	8.48	3.06
200	3200	500	250.8	.031	2123.2	721.0	.33	.23	8.46	2.87

<Trigram Without Position Variable - Superimposed Coding - Fig. 5.8>

BIGRAM WITH POSITION VARIABLE

Signature in ints	Size in bits	Words total	unique	False drop %	Bits by hash	Bits set	% of tried	% of sign	Avg. per word	Avg set
10	160	100	72.0	.430	514.8	120.6	.23	.75	7.15	1.60
100	1600	100	72.0	.041	514.8	264.8	.51	.16	7.15	3.67
200	3200	100	72.0	.041	514.8	281.0	.54	.09	7.15	3.90
300	4800	100	72.0	.041	514.8	297.4	.58	.06	7.15	4.13
500	8000	100	72.0	.041	514.8	296.8	.58	.04	7.15	4.12
1000	16000	100	72.0	.041	514.8	300.0	.58	.02	7.15	4.16
100	1600	200	129.6	.060	939.0	396.2	.42	.24	7.24	3.05
100	1600	300	177.8	.059	1314.2	462.6	.35	.29	7.56	2.66
100	1600	400	217.0	.073	1625.0	522.2	.32	.33	7.48	2.40
100	1600	500	250.8	.084	1872.4	557.0	.30	.35	7.46	2.22
100	1600	1000	432.7	.135	3368.0	712.0	.21	.44	7.78	1.64
200	3200	200	129.6	.055	939.0	426.0	.45	.13	7.24	3.30
200	3200	300	177.8	.049	1314.2	426.0	.32	.18	7.56	2.45
200	3200	400	217.0	.059	1625.0	584.6	.36	.18	7.48	2.70
200	3200	500	250.8	.068	1872.4	629.0	.33	.20	7.46	2.51

<Bigram with Position Variable - Superimposed Coding - Fig. 5.9>

BIGRAM WITHOUT POSITION VARIABLE

Signature in ints	Size in bits	Words total	unique	False drop %	Bits by hash	Bits set	% of tried	% of sign	Avg. per word	Avg set
10	160	100	72.0	.202	514.8	111.8	.22	.70	7.15	1.55
20	320	100	72.0	.108	514.8	144.6	.28	.45	7.15	2.08
30	480	100	72.0	.087	514.8	159.8	.31	.33	7.15	2.22
40	640	100	72.0	.077	514.8	167.4	.33	.26	7.15	2.32
50	800	100	72.0	.075	514.8	173.2	.34	.22	7.15	2.40
100	1600	100	72.0	.075	514.8	173.2	.34	.11	7.15	2.40
200	3200	100	72.0	.075	514.8	174.4	.34	.05	7.15	2.42
300	4800	100	72.0	.075	514.8	174.4	.34	.04	7.15	2.42
100	1600	200	129.6	.065	939.0	227.6	.24	.14	7.24	1.76
100	1600	300	173.8	.064	1314.2	256.4	.20	.16	7.56	1.48
100	1600	400	217.0	.070	1625.0	277.8	.17	.17	7.48	1.28
100	1600	500	250.8	.077	1872.4	294.8	.16	.18	7.46	1.18
100	1600	1000	432.7	.088	3368.0	359.3	.11	.22	7.78	0.83
200	3200	200	129.6	.065	939.0	229.0	.24	.07	7.24	1.76
200	3200	300	173.8	.064	1314.2	259.2	.20	.08	7.56	1.49
200	3200	400	217.0	.069	1625.0	281.8	.17	.08	7.48	1.29
200	3200	500	250.8	.077	1872.4	300.4	.16	.09	7.46	1.19

<Bigram without Position Variable - Superimposed Coding - Fig. 5.10>

SIMPLE TRUNCATION WITH POSITION VARIABLE

Signature in ints	Size in bits	Words total	unique	False drop %	Bits by hash	Bits set	% of tried	% of sign	Avg. per word	Avg set
10	160	100	72.0	.256	351.2	66.0	.19	.41	4.87	0.94
100	1600	100	72.0	.086	351.2	83.6	.24	.05	4.87	1.16
200	3200	100	72.0	.086	351.2	83.8	.24	.05	4.87	1.16
300	4800	100	72.0	.086	351.25	83.8	.24	.05	4.87	1.16
500	8000	100	72.0	.086	351.25	83.8	.24	.05	4.87	1.16
100	1600	200	129.6	.216	649.25	100.2	.15	.06	5.00	0.77
100	1600	300	173.8	.310	872.5	107.2	.12	.06	5.02	0.61
100	1600	400	217.0	.355	1093.7	111.7	.10	.07	5.04	0.51
100	1600	500	250.8	.392	1261.7	117.7	.09	.07	5.03	0.46
200	3200	200	129.6	.216	649.25	100.5	.15	.03	5.01	0.77
200	3200	300	173.8	.306	872.5	107.5	.12	.03	5.02	0.61
200	3200	400	217.0	.355	1093.7	111.7	.10	.03	5.03	0.51
200	3200	500	250.8	.390	1261.7	119.5	.09	.03	5.03	0.48

<Simple Truncation - Superimposed Coding - Fig. 5.11>

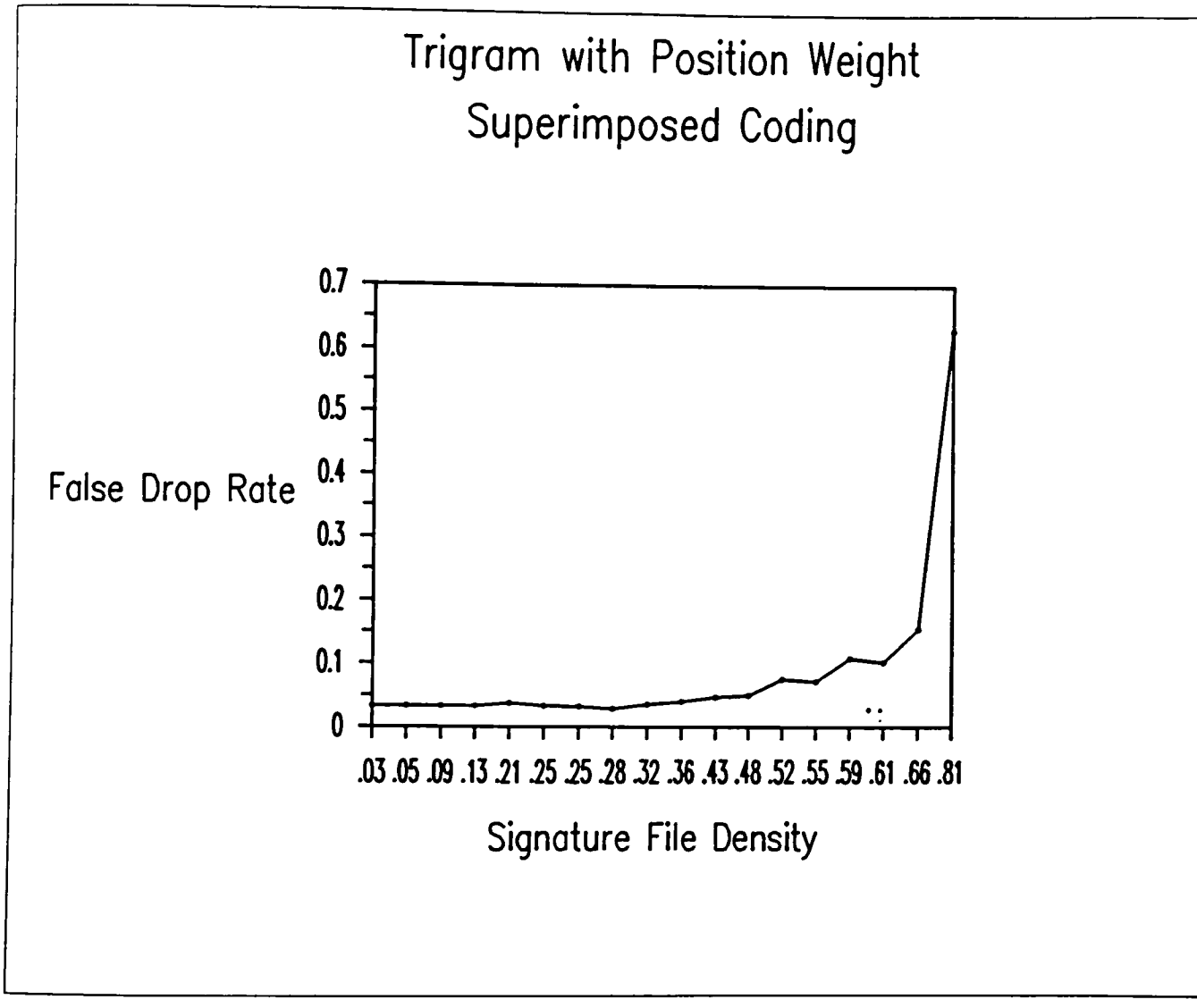
As was mentioned, the columns "% of tried," "average set," and "% of sign" are different measures of roughly the same phenomenon. Thus while one might as easily chart the percentage of false drops against any of these in order to examine the relative performance of each algorithm, it was decided to use the measure which has been suggested in the literature, percentage of signature filled.

Except for the simple truncation algorithm, the results for each algorithm are consistent in that as the density increases beyond a certain point, the false drops increase. This point varies somewhat for each algorithm, suggesting that the notion of 50% density, which was found in the literature as a falling off point is not always achieved. However, it is a good demarcation in that no algorithms succeeded beyond 50% full. In rank order, the algorithms performed as shown in figure 5.12.

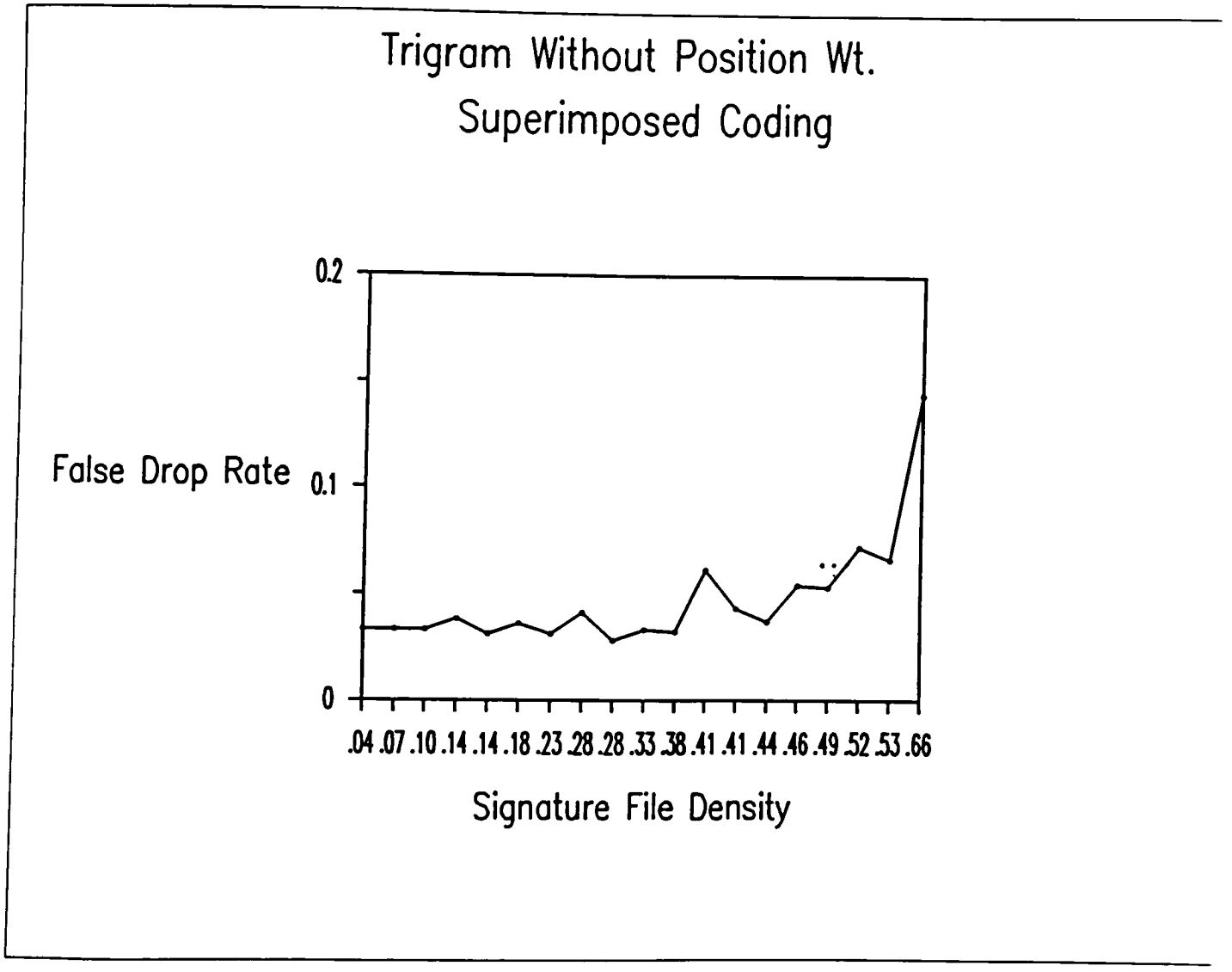
algorithm	Optimum False Drop Rate	Range of Signature Density
Trigram w/ Position	3%-4%	> 40%
Trigram	3%-4%	> 40%
Bigram w/ Position	4%	> 10%
	4%-7%	> 30%
	4%-8%	> 50%
Bigram	6%-8%	> 30%
	6%-9%	> 50%
Simple truncation	only one case less 10% false drops, other data not consistent	

<Superimposed Coding Ranked Order - Fig. 5.12>

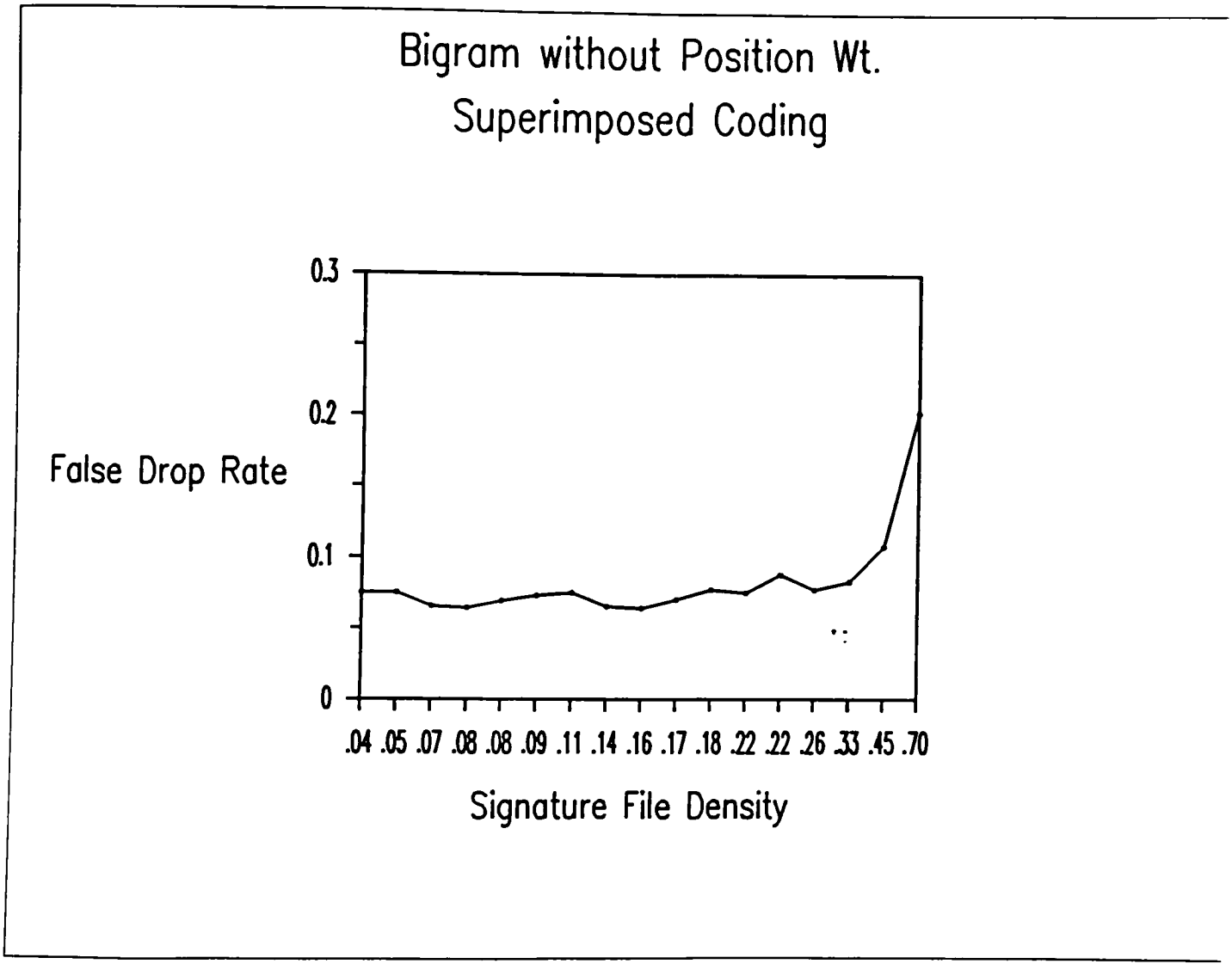
Figures 5.13 through 5.18 graph each individual hashing algorithm on the measure of false drops by density.



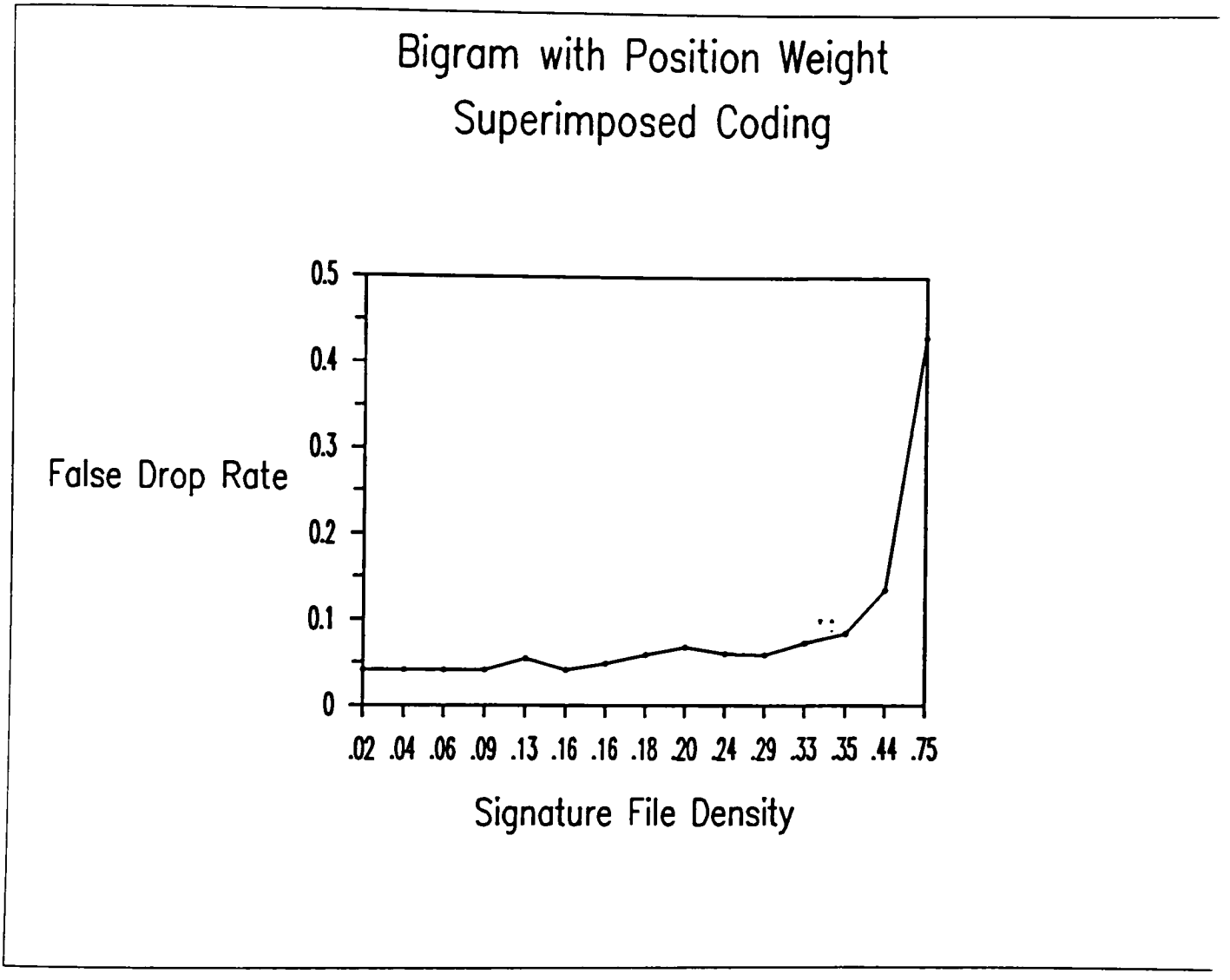
<Trigram with Position Weight - Superimposed Coding - Fig. 5.13>



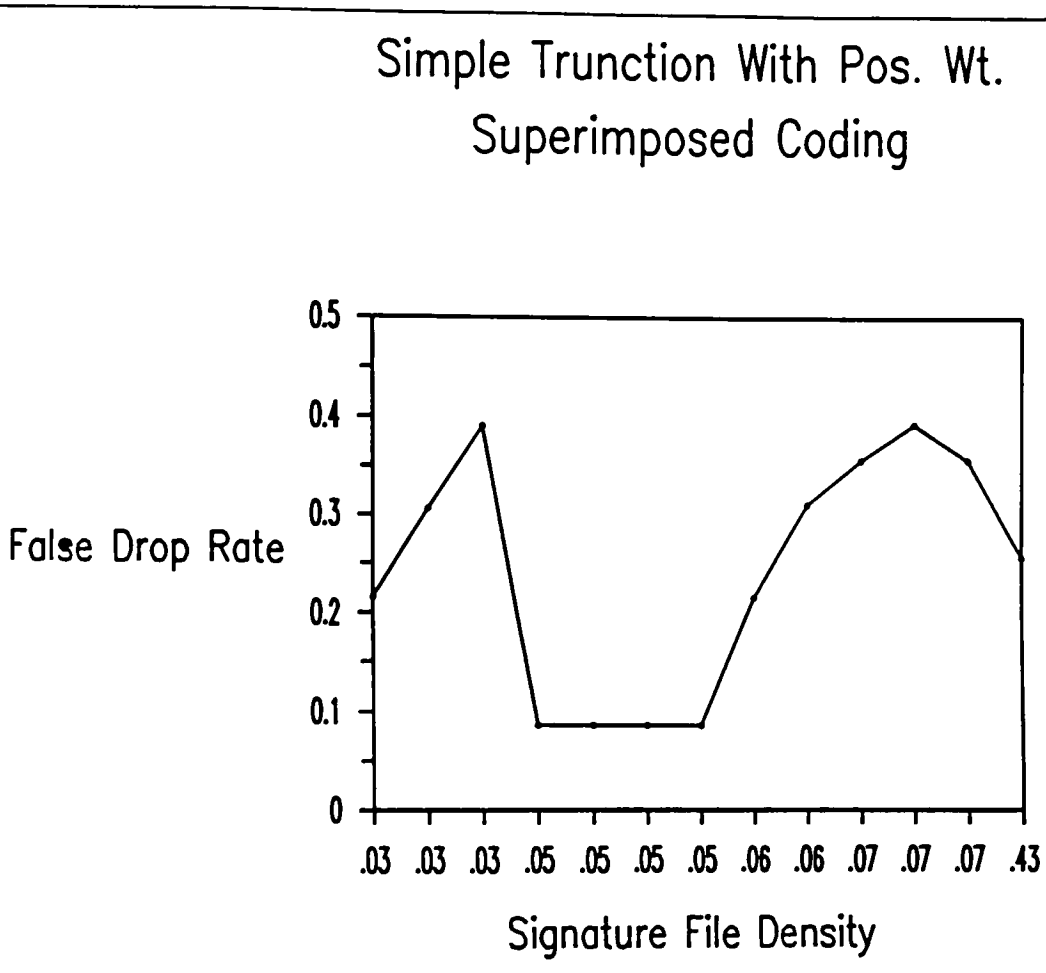
<Trigram Without Position Weight - Superimposed Coding - Fig. 5.14>



<Bigram Without Position Weight - Superimposed Coding - Fig. 5.15>



<Bigram With Position Weight - Superimposed Coding - Fig. 5.16>



<Simple Truncation with Position Weight - Superimposed Coding - Fig. 5.18>

5.7.3 Conjunctive Query Results

Since the testing of conjunctive queries was considerably limited, the results must be viewed as tenative. Of the hash algorithms used for all testing, the medium performer, the bigram hash, was chosen as the test algorithm. For the superimposed signature files, the smallest signature size of 100 integers was chosen to provide a "worst case." As the following figures show, the probability of a two-word false drop is about 1% for either concatenated word signatures or superimposed signatures. However, the probability is not orthogonal, that is, the probability of a false drop multiplied by itself. In fact, the conjunctive false drop rates, while low, are somewhat higher than the orthogonal rate would be. It would be interesting to speculate that Shannon's theory of entropy for the English language, which would predict that the probabilities of two sequential words are not independent probabilities, is at work here. But such speculation would be just that. Significantly broader, larger and more specific test cases would need to be designed to test Shannon's theory or even to accurately monitor conjunctive false drops.

While the rates for 3-word queries were observed at the same time 2-word queries were monitored, they are not reported because so few test cases had any false drops at

all. Without seeing any range or pattern to the 3-word false drops, it was concluded that the test cases were inadequate to monitor 3-word false drops. Two-word false drops are presented in the Figures 5.19 and 5.20.

Conjunctive False Drops for 2-Word Queries

Bigram Hash

Superimposed Coding

Signature Size(Integers)	Total Words	Single Word False Drop %	Two Word False Drop %
100	300	.064	.0045
100	400	.070	.0119
100	500	.070	.0112
100	1000	.088	.0160

<Conjunctive Queries - Superimposed Coding - Fig. 5.19>

Concatenated Word Signatures

Total Words	Single Word False Drop %	Two Word False Drop %
300	.053	.0093
400	.057	.0111
600	.066	.0108
700	.077	.0148
800	.072	.008

<Conjunctive Queries - Word Signatures - Fig. 5.19>

5.8 CONCLUSIONS

Overall the superimposed signature cases performed somewhat better than the concatenated word signatures, and the trigram hash was the best performer in both cases. Figure 5.21 summarizes ranges of optimum behavior.

Hash Algorithm	Superimposed	Word Signatures
Trigram	3%-4%	4%-6%
Bigram	4%-7%	4%-8%
Simple	No optimum behavior	2%-12%

<Optimum Behavior - Fig. 5.20>

Examining then the best case, the trigram hash for superimposed coding, it appears that it succeeded best because it was able to set more unique codes for each word (average set). This fact is clearly related to the ability of the algorithm to generate dispersed random codes. What is

also of significance, however, is the fact that it was impossible to get below a certain optimum behavior by creating a sparse vector. When the signature was only 3%-5% filled the performance was the same as when it was 25% or even 50% filled. Furthermore, the concept of 50% density as a cutoff point was not completely supported. There were non-optimum rates of 5% false drops when the signature was slightly less than 50% full.

Calculating the actual size of the signature for the largest test case still within the optimum performance, one finds that 500 total words or 250.8 unique words were stored in a signature of 200 integers or 3200 bits. Since duplicate words in the superimposed environment simply "reset" previously set bits, they do not increase the density of the signature and do not need to be guarded against. Assuming an average of five characters per word and one blank between words, and assuming one byte per character, the original file would occupy 24,000 bits. This means that the signature of 3200 bits occupied a little more than 10% of the original size.

Performance factors for signature files were not monitored, but it is obvious that the superimposed process, which involves checking only seven bits in the mask after the hash, is significantly more efficient than any search mechanism of concatenated word signatures could be.

The focus of this thesis has not been on how to devise a practical application for signature files, but one can speculate that signature files could in fact provide reasonable alternatives to inverted files with their high storage overhead. However, the problems of false drops may be somewhat higher than posited in the literature, and is significant enough to warrant some final check of the original document to verify the existence of the query word. In fact, one might well imagine constructing layered or blocked signature files which divided the total set of documents into smaller units to limit the search space for such final checking. Each smaller set of files would maintain their own signature which could be searched if a larger document collection signature indicated the presence of the query words.

Finally, recognizing the potential of signature files for information storage, and recognizing the long history of the basic concepts of superimposed coding for text retrieval, one wonders at the absence of developed applicatios in this area. Is it merely that informaion retrieval has not received the same attention as database retrieval? Have the vertical markets for information retrieval, primarily libraries and publishers, been too limited to foster diverse approaches to information storage? Have the performance limitations, and the need to guard

against over-filling a signature file blocked their adoption in favor of simpler procedures? Or more simply, has the decreasing cost of computer memory made the economies of signature file processing unnecessary? One might hope that signature files may yet become a standard information processing technology. Certainly if there is to be a driving force it would be that of maintaining larger amounts of personal data in office workstation environments where the memory capacity is relatively limited.

6.0 BIBLIOGRAPHY

Aho, A. V. and Corasick, M. J. "Efficient String Matching: An Aid to Bibliographic Search", COMMUNICATIONS OF THE ACM 18no6: 333-340, June 1975.

Aho, A. V., Hopcroft, J. E. and Ullman, J. D. THE DESIGN AND ANALYSIS OF COMPUTER ALGORITHMS, Addison-Wesley, 1974.

Arnold, Richard. "An Alternative to Perfect Hashing" IBM Technical Report RC 10332 (#46085). 1/18/84. 8 pages.

Abramson, N. INFORMATION THEORY AND CODING, McGraw-Hill, 1963 chapt. 2.8, pp. 33-8.

Biermann, Alan W. et al. "Natural Language with Discrete Speech as a Mode for Human-to-Machine Communication," COMMUNICATIONS OF THE ACM 28no6:628-636, June, 1985.

Bird, P.R., "Design Analysis of Random Superimposed Coding Methods for Data Storage," in INFORMATION PROCESSING & MANAGEMENT 11:79-88, 1975.

Blair, David C. and M. E. Maron, "An Evaluation of Retrieval Effectiveness for a Full-Text Document Retrieval System," COMMUNICATIONS OF THE ACM 26no3:289-299 March, 1985.

Bloom, B. H. "Space/Time Trade-offs in Hash Coding with Allowable Erros," COMMUNICATIONS OF THE ACM 13no7:422-426 July 1970.

Bourne, Charles P. and Donald F. Ward, "A Study of Methods for Systematically Abbreviating English Words & Names," JOURNAL OF THE ACM 8no4, 1961.

Bourne, Charles P. and Donald F. Ward, "A Study of the Statistics of Letters in English Words," INFORMATION AND CONTROL 4:48-67 1961.

Boyer, R. S. and S. J. Moore, "A Fast String Searching Algorithm" COMMUNICATIONS OF THE ACM 20no10:762-772 Oct, 1977.

Brace, D. A., "Direct Coding of English Language Names," COMPUTER JOURNAL 6:113-117, 1963.

Brzozowski, J.P. "Masquerade: Searching the Full Text of Abstracts Using Automatic Indexing," J. INF.SCI. PRINC. and PRACT (Netherlands) 6no2-3:67-73.

Carter, J. L. "Method of Extending Hash Functions for Long Keys," IBM TECHNICAL DISCLOSURE BULLETIN 19no2:4826.

Case, R.S. and J. W. Perry, eds. Punched Cards: Their Applications to Science and Industry. New York: Reinhold, 1951.

Christodoulakis, S. "Access Files from Batching Queries in Large Information Systems", PROCEEDINGS ACM/IEEE INT CONFERENCE ON DATA BASES (ICOD-2), 1983.

Christodoulakis, Stavros and Chris Faloutsos, "Design Considerations for a Message File Server, " IEEE TRANSACTIONS ON SOFTWARE ENGINEERING 10no2:201-210 March, 1984.

Christodoulakis, S., "Framework for the Development of an Experimental Mixed-Mode Message System," in VanRijsbergen, C.J., RESEARCH AND DEVELOPMENT IN INFORMATION RETRIEVAL (Proceedings of the Third Joint BCS and ACM Symposium, King's College, Cambridge 2-6, July 1984), Cambridge University Press, 1984, pp 1-20.

Christodoulakis, S., "Office Filing," in OFFICE AUTOMATION John Hogg, ed. Springer Verlag, to be published, 1985.

Cichelli, Richard J. "Minimal Perfect Hash Functions Made Simple," COMMUNICATIONS OF THE ACM 23no1:17-19 Jan '80.

Clare, A.C., Elizabeth M. Cook and M. F. Lynch, "The Identification of Variable-length, Equifrequent Character Strings in a Natural Language Database," COMPUTER JOURNAL 15:259-262, 1972.

Comer, Douglas and Vicnet Y. Shen, "Hash-Bucket SearcH; A Fast Technique for Searching an English Spelling Dictionary", SOFTWARE PRACITCE AND EXPER. 12no7:669-682, 1982

Cook, Curtis R. "A Letter Oriented Minimal Perfect Hashing Function," SIGPLAN NOTICES 17no9:18-27 September, 1982.

Cooper, David and Michael F. Lynch and Alice HW McLure, "Generation of Fixed-Length Search Codes from INSPEC words using Techniques based on Variety Generation and on Division-Hashing", PROGRAM (GB) 15no4:226-232 Oct 1981.

Cooper, William S. "On Deriving Design Equations for Information Retrieval Systems," JOURNAL OF THE AMERICAN SOCIETY FOR INFORMATION SCIENCE 385-395 Nov/Dec 1970.

Coulouris, G. F., J.M. Evans and R. W. Mitchell, "Towards Content-addressing in data bases," COMPUTER JOURNAL 15:95-98 1972.

Dattola, R. D. "FIRST: Flexible Information Retrieval For Text", J of ASIS 30:9-14, 1979.

Edwards, A. Wood and Robert L. Chambers, "Can A Priori Probabilities Help in Character Recognition?", JOURNAL OF THE ACM 11not4:465-470, Oct., 1964.

Fagin, Ronald et al. "Extendible Hashing -- A Fast Access Method for Dynamic Files," ACM TRANSACTIONS ON DATABASE SYSTEMS 4no3:315-344, S '1979.

Fairthorne, R. A. "Empirical Hyperbolic Distributions (Bradford-Zipf-Mandelbrot) for Bibliometric Description and Prediction," JOURNAL OF DOCUMENTATION 25:319-343 D '69.

Faloutsos, Chris and Stavros Christodoulakis. "Signature Files: An Access Method for Documents and Its Analytical Performance Evaluation," ACM TRANSACTIONS ON OFFICE INFORMATION SYSTEMS 2no4:267-288, O '84.

Faloutsos, Chris and Stavros Christodoulakis. "Access Methods for Documents" in OFFICE AUTOMATION to be published by Springer-Verlag, 1985.

Faloutsos, Chris. "Signature Files: Design and Performance Comparison of some Signature Extraction Methods," ACM SIGMOD 1985 submitted for publication.

Files, John R. and Harry D. Huskey. "An Information Retrieval System Based on Superimposed Coding," AFIPS FALL JOINT COMPUTER CONFERENCE 35:423-432, 1969.

Fokker, D. W. and M. F. Lynch. "Application of the Variety-Generator..." J OF LIBRARY AUTOMATION 7:105-118 and &;201-203, 1974

Frazer, W. D. "A Proposed System for Multiple Descriptor Data Retrieval," in
SOME PROBLEMS IN INFORMATION SCIENCE (M. Kochen, ed), New York: NY: Scarecrow Press, 1965.

French, James C. IDAM FILE ORGANIZATIONS. UMI Research Press, 1985.

Gabbe, J. D., T.B. London, R.E. Miller and J.D. Beyer. "Applications of Superimposed Coding to Partial-Match Retrieval," in COMPSAC 78, IEEE COMPUTER SOFTWARE AND APPLICATION CONFERENCE 464-469.

Goto, E. M. Sassa and Y. Kanada, "Studies on Hashing II. Algorithms and Programming with CAMS," J. INF. PROCESS (JAPAN) 3no1:13-22 and another article 1-12

Harrison, Malcolm C. "Implementation of the Substring Test by Hashing," COMMUNICATIONS OF THE ACM 14no12:777-779 Dec 1971.

Haskin, Roger "On Extending the Functions of a Relational Database System," ACM SIGMOD 1982, pp. 207-212

Heaps, H.S. INFORMATION RETRIEVAL: COMPUTATIONAL AND THEORETICAL ASPECTS, Academic Press, 1978.

Heaps, H. S. "Storage Analysis for A Compression Coding for Document Data Bases" INF 1972 1047-1061

deHerr, T. "Quasi Comprehension of Natural Language Simulated by Means of Information Traces," INFORMATION PROCESSING & MANAGEMENT 15:89-98 1979.

Hutton, F. C. "Peekabit, Computer Offspring of Punched Card Peekaboo, for Natural Language Searching," COMMUNICATIONS OF THE ACM 11no9:595-598, September, 1968.

Jaeschke, G.. "Reciprocal Hashing: A Method for Generating Minimal Perfect Hashing Functions," COMMUNICATIONS OF THE ACM, 24no12:829-832 Dec 1981.

Jeliaek, "On Variable-Length-to-Block Coding" TRANSACTIONS OF THE IEEE INFORMATION THEORY 1972:IT-18:765-774.

Kautz, W. H. "Nonrandom Binary Superimposed Codes," IEEE TRANSACTIONS ON INFORMATION THEORY pp.363-377 Oct, 1964.

Knuth, Donald E. "Superimposed Coding," in ART OF COMPUTER PROGRAMMING: VOL. 3 SORTING AND SEARCHING, Addison-Wesley, 1973, pp. 559-563.

Knuth, D. E., J. H. Morris and V. B. Pratt, "Fast Pattern Matching in Strings" SIAM JOURNAL OF COMPUTING 6no2:323-350, June, 1977.

Kohonen, T. and E. Reuhkala, "A Very Fast Associative Method for the Recognition and Correction of Misspelt words, Based on Redundant Hash Addressing," PROCEEDINGS OF THE 4TH INTERNATIONAL JOINT CONFERENCE ON PATTERN RECOGNITION. Kyoto Japan. XXII&1166 807-809.

Lee, D. L. and F. H. Lochovsky, "Text Retrieval Machines" in OFFICE AUTOMATION, John Hogg, ed. Springer Verlag, to be published, 1985.

Lewis, M. "Hashed Tables", PRACT. COMPUT (GB) 7no5: 35

Lynch, Michael F. "Compression of Bibliographic Files Using an Adaptation of Run-Length Coding," INFORMATION STORAGE AND RETRIEVAL 9:207-214, 1973.

Lynch, M. F. "Creation of Bibliographic Search Codes for Hash Addressing the Variety-Generator Method," PROGRAM 1975 9:46-55.

Lynch, Michael F. "Variety Generation--A Reinterpretation of Shannon's Mathematical Theory of Communication, and Its Implications for Information Science," JOURNAL OF THE AMERICAN SOCIETY FOR INFORMATION SCIENCE 28:19-25, Jan, 1977.

Mace, Scott "On-line Data at Off-line Prices: Stand-Alone Electronic System Will Index Abstracts of Microcomputer Magazines," INFOWORLD 7no14:36 April 8, 1985.

MacLeod, Ian A. "A Data Base Management System for Document Retrieval Applications," INFORMATION SYSTEMS 6no2:131-137, 1981.

Mandelbrot. "An Informational Theory of the Statistical Structure of Language," in Jackson, W., ed. COMMUNICATION THEORY, London:Butterworth, 1953, pp. 486-501.

McIlroy, M. Douglas, "Development of a Spelling List," IEEE TRANSACTIONS ON COMMUNICATIONS com-30no1:91-99 Jan '1982.

Mor, M. and A. S. Fraenkel, "A Hash Code Method for Detecting and Correcting Spelling Errors," COMMUNICATIONS OF THE ACM 25no12:935-938 Dec 1982.

Nix, R. "Experience with a Space Efficient Way to Store a Dictionary," COMMUNICATIONS OF THE ACM 25no5:297-8.

Ohlman. "Subject Word Letter Frequencies Application to Superimposed Coding," PROCEEDINGS OF THE INTERNATIONAL CONFERENCE OF SCIENTIFIC INFORMATION. vol 2:903-916 Washington, D.C. 1954.

Orosz, G. and L. Tackacs, "Some Probability Problems Concerning the Marking of Codes into the Superimposed Field", JOURNAL OF DOCUMENTATION 12no4:231-234 Dec 1956.

Partridge, D. "A Dynamic Data Base..." COMPUTER JOURNAL 1975 18:43-48

Pfaltz, J. L., W. H. Berman and E. M. Cagey. "Partial Match Retrieval Using Indexed Descriptor Files" COMMUNICATIONS OF THE ACM 23:522-528, September, 1980.

Rabitti, F., "Evaluation of Access Methods to Text Documents in Office Systems," in VanRijsbergen, C.J., RESEARCH AND DEVELOPMENT IN INFORMATION RETRIEVAL (Proceedings of the Third Joint BCS and ACM Symposium, King's College, Cambridge 2-6, July 1984), Cambridge University Press, 1984, pp. 21-33.

Radue, Jon, "On the Design of an Interactive Spelling Dictionary for Personal Computers," ACM SIGPC NOTES 6no1:197-199, (1983 ACM Conference on Personal and Small Computers).

Ramamoorthy, C. V. "Document Compaction by Variable Lenth Encoding", PROCEEDINGS ASIS 1:507-513, 1964.

Roberts, C. S. "Partial Match Retrieval Via Superimposed Codes" PROCEEDINGS IEEE vol. 67:1624-1642, December, 1979

Sacks-Davis, R. "A Two Level Superimposed Coding Scheme for Partial Match Retrieval," INFORM. SYSTEMS 8no4: 273-280, 1983.

Schuegraf, E. J. "A Comparison of Algorithms for Data Base Compression By Use of Fragments as Language Elements," INFORMATION STORAGE AND RETRIEVAL 10:309-319, 1974.

Schuegraf, E. J. and H. S. Heaps "Selection of Equi-frequent Word Fragments for Information Retrieval", INFORMATION STORAGE AND RETRIEVAL 9:697-711 1973.

Schwartz, Eugene S. "A Dictionary for Minimum Redundancy Encoding," JOURNAL OF THE ACM 10:413-439, 1963.

Severance, Dennis. "Identifier Search Mechanisms: A Survey and Generalized Model," COMPUTING SURVEYS 6no3:175-194 Sept '74.

Shannon, "Prediction and Entropy of Printed English" BELL SYSTEM TECH JOURNAL 30no1:50-64 1951. (m396)

Slonim, J., et al. "NDX-100: An Electronic Filing Machine for the Office of the Future," IEEE Computer 14no5:24-36, May, 1981.

Stellhorn, William H. "An Inverted File Processor for Information Retrieval," IEEE TRANSACTIONS ON COMPUTERS 26no12:1258-1267, Dec. 1977.

Stiassny, Simon, "Mathematical Analysis of Various Superimposed Coding Methods," AMERICAN DOCUMENTATION 11:155-169 1960.

Taube, A. Kreithan and L. B. Heilprun "Superimposed Coding for Data Storage with an Appendix of Dropping Fraction Tables" in MECHANIZATION OF DATA RETRIEVAL (STUDIES IN COORDINATE INDEXING SERIES) vol. 4, Washington, D.C.: Documentation, Inc., 1957. chapter 5.

Tharp, A. L., and Kuo-Chung Tai. "The Practicality of Text Signatures for Accelerating String Searching," SOFTWARE PRACTICE AND EXPERIENCE 12no1:35-44

Tsichritzis, D., S. Christodoulakis, A. Lee and J. Vandenbroek, "A Multimedia Filing System," in OFFICE AUTOMATION, John Hogg, ed., Springer-Verlag to be published, 1985.

Tsichritzis, C., "Forms Management" COMMUNICATIONS OF THE ACM 25:453-478, July, 1982

Uhlmann, W. "The Application of Random Superimposed Coding and Chain Spelling to Peek-A-Boo Cards," AMERICAN DOCUMENTATION 15:89-92 1964.

Vallarino, O. "On the Use of Bit Maps for Multiple Key Retrieval," ACM SIGPLAN Notices vol. 11 special issue, 108-114 March, 1976.

vanRijsbergen, C. J. "An Algorithm for Information Structuring and Retrieval," COMPUTER JOURNAL 14no4:407-412.

Vitter, Jeffrey Scott, "Implementations for Coalesced Hashing," COMMUNICATIONS OF THE ACM 25no12:911-926 Dec 1982.

Wagner, R. A. "Common Phrases and Minimum Text Storage" COMMUNICATIONS OF THE ACM 16:148-153, 1973.

Wagner, R. A. "An Algorithm for Extracting Phrases in a Space Optimal Fashion", COMMUNICATIONS OF THE ACM 16:183-185, 1973.

Walker, Verdon R. "Compaction of Names by X-Grams," ASIS PROCEEDINGS Vol. 6:129-135, 1969

Willett, P. "Document Retrieval Experiments Using Indexing Vocabularies of Varying Size II. Hashing, Truncation, Digram and Trigram encoding of index terms," JOURNAL OF DOCUMENTATION 35no4:296-305 December 1979.

Wise, C. S. "Mathematical Analysis of Coding Systems,," in Case, R. S. and J. W. Perry, eds. PUNCHED CARDS: THEIR APPLICATIONS TO SCIENCE AND INDUSTRY, New York:Reinhold, 1951.

Zipf, HUMAN BEHAVIOR AND PRINCIPLE OF LEAST EFFORT, Addison-Wesley, 1949.

Zobrist, A. L. and F. R. Carlson, Jr. "Detection of Combined Occurrences," COMMUNICATIONS OF THE ACM, 20no4:260-261, April, 1977.

```
Basic Program which Creates and Searches Signature File
Example Below is for a Superimposed Signature created with
a Simple Hash
Appendix II contains additional procedures used to alter program
for Word Signature File and for Other Hash Algorithms


/* SIGNATURE FILE CREATOR */
/* May 20, 1986        */

/*****************************************************************************/
/*        Global Declarations                                                */
/*****************************************************************************/

  /* EXTERNAL LIBRARIES */
#include <stdio.h>              /* standard I/O library */
#include <ctype.h>              /* string function library */

   /* STRUCTURE DEFINITION for USE IN TREE */
struct tnode                    /* tree structure definition */
     {
       char *word;              /* input word    */
       char *nonword;           /* scrambled word */
       int  count;              /* frequency of word */
       int leng;                /* length of the word */
       int drops;               /* does nonword match in sign */
       struct tnode *left;      /* left child */
       struct tnode *right;     /* right child */
      };  /* END TNODE */

int cnt,drop2,drop3,nodrop2,nodrop3; /* Flags & Counters for Conjunctive
                                        Queries */

  /* MISC CONSTANTS */
#define TRUE 'true'             /* constant flag */
#define FALSE 'fals'            /* constant flag */
#define PRIME 3039              /* prime for hash function */
#define WHITESPACE '0'          /* constant flag */
#define ALPHANUM 'a'            /* constant flag */
#define NOTHING 'z'             /* constant flag */

   /* SET UP CONSTANTS FOR ARRAY DECLARATIONS */
#define MAXWORD 30              /* maximum size of  word (divisable by 10) */
#define BIGNO 100               /* Maximum size of Maxcount array, e.g. largest
                                   number of occurences of one word */
#define SIGNSIZE 500            /* MAX No. of Ints in SIGN--
                                   SIGNSIZE * 16 = MAX size of SIGNATURE */
#define NOSIGNS 2               /* No. of Signatures per File - Array  */
#define WORDBITS 5              /* No. of Bits hashed by each Word --
                                   hashbits array */
#define VOCAB 1000              /* Maximum number of words in any block */

  /* SET UP MASK CONSTANTS  */
```

```
#define MASK0 00
#define MASK1 01
#define MASK2 02
#define MASK3 04
#define MASK4 010
#define MASK5 020
#define MASK6 040
#define MASK7 0100
#define MASK8 0200
#define MASK9 0400
#define MASK10 01000
#define MASK11 02000
#define MASK12 04000
#define MASK13 010000
#define MASK14 020000
#define MASK15 040000
#define MASK16 0100000

/*****************************************************************************/


main()
{
    /****   VARIABLE DECLARATIONS AND INITLIAZATIONS   ***/

        struct tnode *root, *tree();       /* var & function definition */
        char word[MAXWORD],
            ok[5],
            getparas();
            int maxcount[BIGNO],             /* Word frequency array */
            length2[MAXWORD],                /* Length of nonwords */
            length[MAXWORD],                 /* Lengths of Words */
            hashbits[WORDBITS],              /* Each word's hash values*/
            hash(),
            sign(),
            collisions,                      /* Total bit collisions */
            i, n, j, k, t,
            blocksze,                        /* Number words in each block */
            signsze,                         /* Signsize / 16              */
            wbits,                           /* No. bits hashed each word */
            notdup,                          /* Flag */
            add,                             /* Flag */
            notfound,                        /* Flag */
            prntit;                          /* Flag to print sign or not */
        unsigned int signature[NOSIGNS][SIGNSIZE];

        j = 1;
        collisions = 0;                 /* init counters */
        counter = 0;
        notdup = 1;                     /* init flags */
        notfound = 1;
        add = 1;
        for (i = 0; i < NOSIGNS; i++)  /* Initialize Signature Array */
           for (j = 0;  j < SIGNSIZE; j++)
               signature[i][j] = MASK0;
```

```
/*  Program written by: Karen Caviglia
    Superimposed Coding Program  */

/******************** PROGRAM OUTLINE ***************************************/

/*   Each row in the array SIGNATURE[i][j] represents an  individual     */
/*   signature. The size of each individual signature is determined by   */
/*   input parameters:                                                   */
/*   1) Blocksize; the number of total words (not unique words) per      */
/*      signature and                                                    */
/*   2) Signsize, the number of integers (columns) in the row, or        */
/*      individual siganture. Signsize * 16 bits is the total signature  */
/*      size in bits.                                                    */
/*   The other input parameter, wbits, governs the number of hashed bits */
/*      generated for each word in a fixed length hash                   */

/*   Algorithm

     While more rows and not EOF
        Set up Variables

        While not Blocksize and not EOF
            Read word and store it (also generate a nonsense word and
              store it)
            If the word is a unique word
               Hash word
               Put it in superimposed Signature
        End Inner While

        Using inorder traversal or storage tree
          Search for nonsense words in the signature
        Collect statistics and print them

    End Outer While   */
/*
```

```
/******************    MAINLINE    ******************************************/

        i = 1;
        while (i <= NOSIGNS && t != EOF)
           {
            /* INITIALIZATIONS FOR EACH INDIVIDUAL SIGNATURE */
           root = NULL;
           for (k = 0; k <= MAXWORD; k++)
               {
               length[k] = 0;                    /* counting arrays */
               length2[k] = 0;
               }
           for (k = 0; k <= BIGNO; k++)
              maxcount[k] = 0;

           for (k=0; k <= VOCAB; k++)
              drops[k] = 9;

              /* DO ONE ROW OR ONE SIGNATURE */
           j = 1;
           while (j <= blocksze && t != EOF)
              {
               notdup = 1;                       /* set flag */
               t = getword(word,MAXWORD);
               if (t == ALPHANUM)
                  {
                  root = tree(root,word,&notdup,add); /* Add to tree if not a
                                                         duplicate word */
                  if  (notdup)                      /* If not a duplicate
                                                       add to signature */
                     {
                     hash(word,wbits,hashbits,signsze);
                     sign(hashbits,wbits,i,&collisions,notfound,add,signature);
                     }
                  j = j + 1;
                  } /* END IF ALPHANUM LOOP */
              } /* END WHILE < BLOCKS */

           add = 0;                              /* set flag for sign_match */

           printf("The number of Bit Collisions is %3d \n", collisions);
           sign_match(root,hashbits,wbits,signsze,add,
              collisions,drops,&counter,signature);
           count_tree(root,maxcount,length,length2);  /* get statistics  */
           printout(length,length2,maxcount,signature,signsze,i,prntit);
           i = i + 1;
           } /* END WHILE < SIGNS */
        if (t != EOF)                            /* If file was finished last  */
           printf("File not finished \n");       /* signature may not be full  */
                                                 /* to blocksize               */



}       /* END OF MAINLINE */
/*
```

```
/*****************************************************************************/
/*   HASH FUNCTION  Gets word to be hashed, number of bits for that word     */
/*                  size of signature to use, and returns hashed values in   */
/*  SIMPLE HASH     hashbits array                                           */
/*****************************************************************************/

hash(w,wbits,hashbits,signsze)
char *w;
int wbits,hashbits[],signsze;

{  /* Begin HASH function */

   int i, j, t;
   j = 1;
   for (i = 1; i <= wbits; i++)
       hashbits[i] = 0;


   i = 1;
   t = *w++;
             /* Simple hash on value of letters only */
   while ((i <= wbits) & (t != '\n'))
      {
       t = *w++;
       hashbits[i] = ((i * PRIME + t) % (signsze * 16)) + 1;
       i++;
       }  /* END while */

   if (i < wbits)   /* if word too short do a rehash of sorts */
      {
      for (i; i <= wbits; i++)      /* pick up old value of i and finish */
          {
          hashbits[i] = (hashbits[j] * PRIME) % (signsze * 16);
          j = j + 1;
          }  /* END for */
      }  /* END if */

} /* END HASH FUNCTION */

/*
```

```
/*****************************************************************************/
/*    PRINTOUT    Print signatures, maxcount array, length array, total then */
/*                and print collision totals, word count, etc.               */
/* If Prntit flag is 0 then print just totals and not entire arrays & sign   */
/*****************************************************************************/

printout(length,length2,maxcount,signature,signsze,signno,prntit)

int length[], length2[], maxcount[], signature[NOSIGNS][SIGNSIZE],
    signno, prntit;

{  /* BEGIN PRINTOUT */

   int bits, nobits, mod, i,j, unitotal, total, dtotal;

   /*  Print out the Length Frequency array */
   j = 0;

   printf("THE NUMBER OF UNIQUE WORDS BY LENGTH IS:\n\n");
   while (j < MAXWORD)
      {
      for (i =1; i <=10; i++)
         printf("%6d", i + j);
      printf("\n");
      for (i = 1; i <= 10; i++)
         printf("%6d", length[i+j]);
       printf("\n\n");
      j = j + 10;
      }  /* END LENGTH PRINTOUT */

   j = 0;
   dtotal = 0;
   printf("THE NUMBER OF FALSE DROPS BY WORD LENGTH IS: \n\n");
   While (j < MAXWORD)
      {
      for (i = 1; i <= 10; i++)
         printf("%6d", i + j);
      printf("\n");
      for (i = 1; i <= 10; i++)
         {
         printf("%6d", length2[i+j]);
         dtotal = dtotal + length2[i+j]);
         }
      printf("\n\n");
      j = j + 10;
      }  /* END LENGTH2 PRINTOUT */
    printf("The Total Number of False Drops is %d \n", dtotal);

     /* PRINTOUT THE NUMBER OF WORDS BY FREQUENCY */
    printf("THE NUMBER OF WORDS BY FREQUENCY IS:\n\n");
    j = 0;
    total = 0;
    unitotal = 0;
    while (j < BIGNO)
      {
      for (i = 1; i <=10; i++)
```

```
            printf("%6d", i + j);
         printf("\n");
         for (i = 1; i <= 10; i++)
            {
            unitotal = unitotal + maxcount[i+j];
            printf("%6d", maxcount[i+j]);
            total = total + (maxcount[i+j] * (i + j));
            }
            j = j + 10;
            printf("\n\n");
        } /* END WHILE */

        printf("THE TOTAL NUMBER OF UNIQUE WORDS IS %5d \n", unitotal);
        printf("THE GRAND TOTAL NUMBER OF WORDS IS: %5d \n\n", total);


      /* PRINT THE SIGNATURE or AT LEAST COUNT IT */
        if (prntit)
           printf("THE SIGNATURE LOOKS LIKE THIS \n\n");
           j = 1;
           nobits = 0;
        while (j <= signsze)
          {
           bits = signature[signno][j];
           for (i = 1; i <= 16; i++)
              {
              mod = bits%2;
              bits = bits/2;
              if (prntit)
                 printf("%d",mod);
              nobits = nobits + mod;
              }
          if (prntit)
             {
             printf("  ");    /* Put blank between each sixteen digits */
             if (j%4 == 0)
                 {
                printf(" %d ",j);
                printf("\n");
                }
             }
          j++;
          } /* END WHILE */
        printf("THE TOTAL NUMBER OF BITS SET IS: %6d \n", nobits);
        printf(" The number of 2 word false drops is %d \n", drop2);
        printf(" The number of 3 word false drops is %d \n", drop3);

      } /* END PRINTOUT */



      /*
```

```
/*****************************************************************************/
/*    SIGN FUNCTION  Using the values in hashbits array, 1) if creating sign */
/*                   (add - 1) Masks value to integer or 2) if searching     */
/*                   signature (add = 0) returns notfound flag = 0 for any   */
/*                   integer notfound in masking                             */
/*****************************************************************************/

sign(hashbits,wbits,i,collisions,notfound,add,signature)

int hashbits[], wbits, i, *collisions, *notfound, add;
unsigned int signature[NOSIGNS][SIGNSIZE];

{  /* BEGIN SIGN FUNCTION */

    int j, divno, rem;
    unsigned int temp;

    for (j = 1; j <= wbits; j++)
      {
      divno = hashbits[j]/16;   /* Choose the Signature integer */
      rem = (hashbits[j]%16) +1;  /* Chose the bit to use as mask */

    /* General logic:
       if bit already set then
          collisions is incremented;
       else
          if creating signature set appropriate bit or
          if searching signature set notfound flag to false */

      switch(rem)
         {
         case 1:
         if ((temp = signature[i][divno] & MASK1) == MASK1)
            *collisions = *collisions + 1;
         else
            if (add)
              signature[i][divno] = signature[i][divno] | MASK1;
            else
              *notfound = 0;
            break;

         case 2:
         if ((temp = signature[i][divno] & MASK2) == MASK2)
            *collisions = *collisions + 1;
         else
            if (add)
               signature[i][divno] = signature[i][divno] | MASK2;
            else
               *notfound = 0;
         break;

         case 3:
         if ((temp = signature[i][divno] & MASK3) == MASK3)
            *collisions = *collisions + 1;
         else
            if (add)
```

```
      signature[i][divno] = signature[i][divno] | MASK3;
   else
      *notfound = 0;
break;

case 4:
if ((temp = signature[i][divno] & MASK4) == MASK4)
   *collisions = *collisions + 1;
else
   if (add)
      signature[i][divno] = signature[i][divno] | MASK4;
   else
      *notfound = 0;
break;

case 5:
if ((temp = signature[i][divno] & MASK5) == MASK5)
   *collisions = *collisions + 1;
else
   if (add)
      signature[i][divno] = signature[i][divno] | MASK5;
   else
      *notfound = 0;
   break;

case 6:
if ((temp = signature[i][divno] & MASK6) == MASK6)
   *collisions = *collisions + 1;
else
   if (add)
      signature[i][divno] = signature[i][divno] | MASK6;
   else
      *notfound = 0;
break;

case 7:
if ((temp = signature[i][divno] & MASK7) == MASK7)
   *collisions = *collisions + 1;
else
   if (add)
      signature[i][divno] = signature[i][divno] | MASK7;
   else
      *notfound = 0;
break;

case 8:
if ((temp = signature[i][divno] & MASK8) == MASK8)
   *collisions = *collisions + 1;
else
   if (add)
      signature[i][divno] = signature[i][divno] | MASK8;
   else
      *notfound = 0;
break;

case 9:
```

```
if ((temp = signature[i][divno] & MASK9) == MASK9)
   *collisions = *collisions + 1;
else
   if (add)
      signature[i][divno] = signature[i][divno] | MASK9;
   else
      *notfound = 0;
break;

case 10:
if ((temp = signature[i][divno] & MASK10) == MASK10)
   *collisions = *collisions + 1;
else
   if (add)
      signature[i][divno] = signature[i][divno] | MASK10;
   else
      *notfound = 0;
break;

case 11:
if ((temp = signature[i][divno] & MASK11) == MASK11)
   *collisions = *collisions + 1;
else
   if (add)
      signature[i][divno] - signature[i][divno] | MASK11;
   else
      *notfound = 0;
break;

case 12:
if ((temp = signature[i][divno] & MASK12) == MASK12)
   *collisions = *collisions + 1;
else
   if (add)
      signature[i][divno] = signature[i][divno] | MASK12;
   else
      *notfound = 0;
break;

case 13:
if ((temp = signature[i][divno] & MASK13) == MASK13)
   *collisions = *collisions + 1;
else
   if (add)
      signature[i][divno] = signature[i][divno] | MASK13;
   else
      *notfound = 0;
break;

case 14:
if ((temp = signature[i][divno] & MASK14) == MASK14)
   *collisions = *collisions + 1;
else
   if (add)
      signature[i][divno] - signature[i][divno] | MASK14;
   else
```

```
                *notfound = 0;
             break;

          case 15:
          if ((temp = signature[i][divno] & MASK15) == MASK15)
             *collisions = *collisions + 1;
          else
             if (add)
                signature[i][divno] = signature[i][divno] | MASK15;
             else
                *notfound = 0;
          break;

          case 16:
          if ((temp = signature[i][divno] & MASK16) == MASK16)
             *collisions = *collisions + 1;
          else
             if (add)
                signature[i][divno] = signature[i][divno] | MASK16;
             else
                *notfound = 0;
          break;

          default:
             printf("CASE statement default error %d\n", rem);
          } /* END SWITCH */
       } /* END for j */

} /* END SIGN FUNCTION */

/*
```

```
/*****************************************************************************/
/*     GETPARMS PROCEDURE   READS INPUT FILE FOR PARAMETERS  WHICH WILL BE   */
/*                          USED TO TRY DIFFERENT SIGNATURES SIZES           */
/*****************************************************************************/

char getparms(wbits,blocksze,signsze,prntit)

int *wbits, *blocksze, *signsze, *prntit;
{
FILE *parms,*fopen();
float t;

    parms = fopen("parameters", "r");
    if (parms == NULL)
        return(FALSE);
    else
       {
      fscanf(parms, "%d%d%d%d", &*wbits, &*blocksze, &*signsze, &*prntit);
      t = *signsze/*blocksze;
      printf("Wordbits Blocksize Signsize Print flag Signsize in Bits \n");
      printf("%d %10d %10d %10d %10d %10d", *wbits,*blocksze,
             *signsze,*prntit,t);
       return(TRUE);
      }
  } /* END OF GETPARMS */
/*
```

```
/*************************************************************************/
/*  TREE FUNCTION - BUILDS TREE INORDER AND RETURNS THE TNODE STRUCTURE */
/*                  Tree is used to store information about the length  */
/*                  as well as to store nonsense word for false drop    */
/*                  searching                                           */
/*************************************************************************/

struct tnode *tree(p, w, notdup,add)

struct tnode *p;
char *w;
int *notdup,add;

{ /* BEGIN TREE FUNCITON */
        char *strsave();
        int cond;

        if (p == NULL)          /* IF CURRENT NODE IS NULL INSERT NEW */
             {
              if (add)
                 p = (struct tnode *) malloc (sizeof(*p)); /* get new node */
                 p->word = strsave(w);
                 p->count = 1;
                 p->nonword = strsave(w);
                 p->leng = strlen(p->word);
                 notword(p->nonword);           /* turn word into nonsense */
                 p->drops = 0;
                 p->left = NULL;
                 p->right = NULL;
               }
             } /* end if */

        else if ((cond = strcmp(w, p->word)) == 0)    ..
              {                         /* word already there */
              p->count = p->count + 1;
              *notdup = 0;
              }
        else if (cond < 0)
              p->left = tree(p->left, w, &*notdup, add);
        else
              p->right = tree(p->right, w, &*notdup,add);
        return(p);
} /* END TREE FUNCTION */
/*
```

```
/*****************************************************************************/
/*  NOTWORD FUNCTION     Translates word into nonsense by adding one to each */
/*                       letter                                              */
/*****************************************************************************/

char *notword(s)
char *s;

     {
     while (*s != '\0')
        {
         if (*s == 'z')
             ;
         else
            *s = *s + 1;
         s++;
         }   /* END WHILE */

     } /* END NOTWORD */

/*
```

```
/*****************************************************************************/
/* SIGN_MATCH   If the nonword is more than 1 character looks for it in the  */
/*              signature by calling the sign function and checking if       */
/*              notfound is 0 otherwise it stores a count in p->nonword      */
/*              An inorder traversal of the tree is used                     */
/*****************************************************************************/


sign_match(p, hashbits, wbits, signsze, add, collisions,
   drops, counter, signature)
struct tnode *p;
int hashbits[], wbits, signsze, *notfound, add, collisions, drops[], *counter;
unsigned int signature[NOSIGNS][];

/* Using an inorder traversal, hash nonword (scrambled word) and look for it in
   in signature. If it is found set appropriate counters */


{ /* BEGIN SIGN MATCH */

     if (p != NULL)
        { /* Begin Traversal */
            sign_match(p->left, hashbits, wbits, signsze
               add,collisions,drops, &*counter,signature);
            hash(p->word, wbits, hashbits, signsze);
            if (strlen(p->nonword) > 1)
              {
               int notfound;
               notfound = 1;
               sign(hashbits, wbits,collisions,&notfound,add,signature);
               if (notfound)
                 p->drops = 1;
              }
            sign_match(p->right, hashbits,wbits,signsze,
               add,collisions,drops, &*counter,signature);
         } /* END TRAVERSAL */
 } /* END SIGN_MATCH  */
/*
```

```
/*****************************************************************************/
/* COUNT_TREE FUNCTION   Count the various fields which have been set in the */
/*                       tree into summary arrays                            */
/*****************************************************************************/

count_tree(p, maxcount,length,length2)
struct tnode *p;
int maxcount[],length[],length2[];

/* Using an inorder traversal, count the various fields in the tree node
   structure */

{ /* BEGIN COUNT_TREE */

     int i;

     if (p != NULL)
        { /* Begin Traversal */
         count_tree(p->left, maxcount,length,length2);
         length[p->leng] = length[p->leng] + 1;
         length2[p->leng] = length2[p-leng] + p->drops;
         maxcount[p->count] = maxcount[p->count] + 1;
         /* look for conjunctive queries */
         if (p->drops != 1)
            {
             nodrop2 = 0;
             nodrop3 = 0;
             }
         if ((cnt % 2) == 0)
            {
             if (nodrop2)    /* if is still 1 then 2 word false drop */
                drop2++;
             nodrop2 = 1;   /* reset boolean */
             }
         if ((cnt % 3) == 0)
            {
            if (nodrop3)
               drop3++;
            nodrop3 = 1;
            }
         count_tree(p->right, maxcount,length,length2);
         } /* END TRAVERSAL */
 } /*END COUNT_TREE  */
/*
```

```
/*****************************************************************************/
/* GETWORD FUNCTION */
getword(w, lim)        /* get next word from input file */
char *w;
int lim;
{
        int c, t;
                /*look for first nonwhitespace
                return to main program when non alpha found */
        if (type(c = *w++ = getchar()) != ALPHANUM)
            {
                *w = '\0';
                return(c);
             }
       if isupper(c)
          { *w--;
            *w++ = c= tolower(c);
          } /* END IF */

                /* fill rest of word and echo print */
        while (--lim >0)
              {
                t = type(c = *w++ = getchar());
                if (t == ALPHANUM)   /* IF is ALPHA THEN ECHO PRINT */
                   {
                    if (isupper(c))
                      {
                      *w--;
                      *w++ = c = tolower(c);
                      }  /*END IF ISUPPER */
                   } /* END IF */
                if (t == WHITESPACE)
                      {
                        ungetc(c, stdin);
                        break;
                      } /* END IF */
                } /* END WHILE */
                if (t == NOTHING)
                     *w--;
        *(w-1) = '\0';
        return(ALPHANUM);
     } /* END GETWORD */
/*
```

```
/*************************************************************************/
/* TYPE FUNCTION */
type (c)                /* return type of character */
int c;
    {
       /* if end of file, end of line, blank, or tab, return whitespace;
          else if alpha return alpha
          else return nothing */
       if (c == EOF || c == '\n' || c == ' ' || c == '\t')
              return(WHITESPACE);
       else if isalnum(c)
              return(ALPHANUM);
       else
              return(NOTHING);
     }

/*************************************************************************/
/* STRSAVE FUNCTION */
char *strsave(s)        /*save string s somewhwere */
char *s;
      {
        char *p;
        if (( p = (char*) malloc(strlen(s) + 1)) != NULL)
               strcpy(p, s);
        return(p);
        }
```

APPENDIX II

The following are examples of alternate hashes which are used in the basic program given in appendix I.

```
/* TRIGRAM HASH without position    ************************************/

hash(w,wbits,hashbits,signsze)
char w[MAXWORD];
int *wbits,hashbits[],signsze;
{
    int count, i, j, blank, tablesize;
    tablesize = signsze * 16;
    count = strlen(w);
    blank = 1;
    i = 1;
    j = 1;
    /* DO INITIAL TRIGRAMS */
    hashbits[j] = (26 * ((26 * blank) + blank) + w[i]) % tablesize;

    j++;
    if (count == 1)
       hashbits[j] = (26 * ((26 * blank) + w[i]) + blank) % tablesize;
    else
       hashbits[j] = (26 * ((26 * blank) + w[i]) + w[i+1]) % tablesize;
    j++;

    /* DO BODY */
    for(i = 1; i <= (count - 2); i++)
       {
       hashbits[j] = (26 * ((26 * w[i]) + w[i+1]) + w[i+2]) % tablesize;
                j++;
       }

    /* DO END TRIGRAMS */
    if (count == 1)
       ;
    else
      {
       hashbits[j] = (26 * ((26 * w[i]) + w[i+1]) + blank) % tablesize;
    j++;
     }
    hashbits[j] = (26 * ((26 * w[i]) + blank) + blank) % tablesize;

    *wbits = j;

}
```

```
/* TRIGRAM HASH with Position   ************************************/


hash(w,wbits,hashbits,signsze)
char w[MAXWORD];
int *wbits,hashbits[],signsze;
{
    int count, i, j, blank, tablesize;
    tablesize = signsze * 16;
    count = strlen(w);
    blank = 1;
    i = 1;                /* multiplying times i weights by word position*/
    j = 1;
    /* DO INITIAL TRIGRAMS */
    hashbits[j] = ((26 * ((26 * blank) + blank) + w[i])   * i) % tablesize;

    j++;
    if (count == 1)
      hashbits[j] = ((26 * ((26 * blank) + w[i]) + blank)   * i) % tablesize;
    else
      hashbits[j] = ((26 * ((26 * blank) + w[i]) + w[i+1])   * i) % tablesize;
    j++;

    /* DO BODY */
    for(i = 1; i <= (count - 2); i++)
      {
      hashbits[j] = ((26 * ((26 * w[i]) + w[i+1]) + w[i+2])    * i) %
tablesize;
                j++;
      }

    /* DO END TRIGRAMS */
    if (count == 1)
      ;
    else
     {
      hashbits[j] = ((26 * ((26 * w[i]) + w[i+1]) + blank)    * i) % tablesize;
    j++;
     }
    hashbits[j] = ((26 * ((26 * w[i]) + blank) + blank)   * i) % tablesize;

    *wbits = j;

}
```

```
/* BIGRAM HASH  with Position weight   ************************************/


hash(w,wbits,hashbits,signsze)
char w[MAXWORD];
int *wbits,hashbits[],signsze;
{
    int count, i, j, blank, tablesize;
    tablesize = signsze * 16;
    count = strlen(w);
    blank = 1;
    i = 1;        /* multiplying times i in body weights hash by position */
    j = 1;
    hashbits[j] = ((26 * blank) + w[i]) % tablesize;
    j++;

    /* DO BODY */
    for(i = 1; i <= (count - 1); i++)
       {
       hashbits[j] = (((26 * w[i]) + w[i+1])  * PRIME * i) % tablesize;
                j++;
       }


     hashbits[j] = (( 26 * w[i]) + blank) % tablesize;
    *wbits = j;

}
```

```
/* BIGRAM HASH without Position weight    ************************************/


hash(w,wbits,hashbits,signsze)
char w[MAXWORD];
int *wbits,hashbits[],signsze;
{
    int count, i, j, blank, tablesize;
    tablesize = signsze * 16;
    count = strlen(w);
    blank = 1;
    i = 1;
    j = 1;
    /* DO INITIAL BIGRAM */
    hashbits[j] = ((26 * blank) + w[i]) % tablesize;

    j++;

    /* DO BODY */
    for(i = 1; i <= (count - 1); i++)
       {
       hashbits[j] = ((26 * w[i]) + w[i+1]) % tablesize;
                j++;
       }


     hashbits[j] = (( 26 * w[i]) + blank) % tablesize;
    *wbits = j;

}
```

The following procedure replaces the sign_match procedure given in the basic program for the word signature test cases. For word signatures, bit mask matching need not be done once the query mask is set. Instead the integer value of that particular 16 bit (unsigned integer) code is searched sequentially in the entire signature.

```
/* SIGN MATCHING PROCEDURE FOR WORD SIGNATURES  ####################*/

sign_match(p, i, hashbits,wbits,signsze,add,collisions,signature)


struct tnode *p;
int i, hashbits[], *wbits, signsze,add, collisions;
unsigned int signature[NOSIGNS][SIGNSIZE];

{ /* BEGIN SIGN MATCH */

     if (p != NULL)
       {
        sign_match(p->left,i,hashbits,&*wbits,signsze,add,collisions,signature);
        hash(p->nonword,&*wbits,hashbits,signsze);
        if (strlen(p->nonword) > 1)
          {
          int notfound, j, n, bits;
          notfound = 1;
          bits = *wbits;
          j = 0;
          n = 0;
          signature[0][0] = 0;

          sign(hashbits,bits,n,j,&collisions,&notfound,add,signature);
          for (j = 1; j <= signsze; j++)
              if (signature[i][j] == signature[0][0])
                 {
                 p->drops = 1;
                 break;
                 }
      sign_match(p->right,i,hashbits,&*wbits,signsze,add,collisions,signature);

       }  /* End if */

) /* End SIGN */
```

APPENDIX IV

The following is a sample test case for superimposed coding using a trigram hash.

The Following Data is from Super.Trigram with 2 Trigram 3 word drops

wordbits	blocksize	signsize	doprint	total signature
7	100	100	1	1600

THE NUMBER OF UNIQUE WORDS BY LENGTH IS:

1	2	3	4	5	6	7	8	9	10
1	8	8	12	8	7	6	4	5	2

11	12	13	14	15	16	17	18	19	20
4	1	1	3	0	1	0	0	0	0

21	22	23	24	25	26	27	28	29	30
0	0	0	0	0	0	0	0	0	0

THE NUMBER OF FALSE DROPS BY WORD LENGTH IS:

1	2	3	4	5	6	7	8	9	10
0	2	1	0	0	0	0	0	0	0

11	12	13	14	15	16	17	18	19	20
0	0	0	0	0	0	0	0	0	0

21	22	23	24	25	26	27	28	29	30
0	0	0	0	0	0	0	0	0	0

The Total Number of False Drops is 3

THE NUMBER OF WORDS BY FREQUENCY IS:

1	2	3	4	5	6	7	8	9	10
52	15	1	1	1	1	0	0	0	0

11	12	13	14	15	16	17	18	19	20
0	0	0	0	0	0	0	0	0	0

21	22	23	24	25	26	27	28	29	30
0	0	0	0	0	0	0	0	0	0

31	32	33	34	35	36	37	38	39	40
0	0	0	0	0	0	0	0	0	0

41	42	43	44	45	46	47	48	49	50
0	0	0	0	0	0	0	0	0	0

51	52	53	54	55	56	57	58	59	60
0	0	0	0	0	0	0	0	0	0

```
 61   62   63   64   65   66   67   68   69   70
  0    0    0    0    0    0    0    0    0    0

 71   72   73   74   75   76   77   78   79   80
  0    0    0    0    0    0    0    0    0    0

 81   82   83   84   85   86   87   88   89   90
  0    0    0    0    0    0    0    0    0    0

 91   92   93   94   95   96   97   98   99  100
  0    0    0    0    0    0    0    0    0    0

THE TOTAL NUMBER OF WORDS IS    71
The Grand total of Words is 100
THE SIGNATURE LOOKS LIKE THIS
0000000000100000  0100000000010000  0000000000010000  0011000000100010  4
0000000000000000  0000010000000000  0000001000000001  0110000111001100  8
1000100000010000  0000010001000000  1000000010000000  0001001000100010  12
0010011111000100  0010001111101011  0000100000110010  1000000100000010  16
0000100001000000  0000010001000100  0101001000000000  0001010001100000  20
0110000010000000  1100010100000000  0001000000000010  0010000011000100  24
0000000000000010  0100000000001100  0000000000010000  0010110010000110  28
0000100001010111  0110001110101100  1010001000000000  0000101000001000  32
1000000000000000  0000000010000101  0000011110101100  0100000101000000  36
1000000101000000  0000100000011000  0100000000000010  0000000000000000  40
0000000000000100  0000000000100000  0100010001001001  0010000001000010  44
0010000000000000  0001000000101000  0100000000000000  0010100001000000  48
0000000100000100  0001000000010111  1101101101101101  1110000100001100  52
0001000010000000  1000100111000110  0110001000000000  0000000000000000  56
0000001010000000  0000000000000001  0011010011101110  0101010001100010  60
0000000100010000  0000011000000000  0010100000000010  0100000001010000  64
0000000000011000  0100010000000010  0000001000000000  0000010000000100  68
0000000000000001  0000001010001001  1000000000010100  0000001010000001  72
0001100010000000  0001000000001010  1000100001010000  0000000000010000  76
0100100011101001  0000000001010100  0000000011010000  0000000010000000  80
0100010001100000  0100111000000000  0000000000010100  0000000001000010  84
0001000100000000  0010000001000000  0000000000011011  0100000000000100  88
0000000010010000  0000000001010101  1000100010000000  0000000100000000  92
0100010000010101  0001010101100000  0100010001000010  0000000000010010  96
0000010000000000  1010000000000001  0000110000000000  0000000001000001  100

The total number of bits set is 312
The number of 2 word false drops is 0
The number of 3 word false drops is 0
The number of8it Collisions is 267
```

The following data is from a concatenated word signature program using a trigram hash. Note: the blank integers represent duplicate words which were not processed.

The following output is from trigram in WORDSIG

wordbits	blocksize	signsize	doprint	total signature
7	100	100	1	1600

THE NUMBER OF UNIQUE WORDS BY LENGTH IS:

1	2	3	4	5	6	7	8	9	10
1	8	8	12	8	7	6	4	5	2

11	12	13	14	15	16	17	18	19	20
3	1	1	3	0	1	0	0	0	0

21	22	23	24	25	26	27	28	29	30
0	0	0	0	0	0	0	0	0	0

THE NUMBER OF FALSE DROPS BY WORD LENGTH IS:

1	2	3	4	5	6	7	8	9	10
0	2	0	0	0	0	0	0	0	0

11	12	13	14	15	16	17	18	19	20
0	0	0	0	0	0	0	0	0	0

21	22	23	24	25	26	27	28	29	30
0	0	0	0	0	0	0	0	0	0

The Total Number of False Drops is 2

THE NUMBER OF WORDS BY FREQUENCY IS:

1	2	3	4	5	6	7	8	9	10
51	15	1	1	1	1	0	0	0	0

11	12	13	14	15	16	17	18	19	20
0	0	0	0	0	0	0	0	0	0

21	22	23	24	25	26	27	28	29	30
0	0	0	0	0	0	0	0	0	0

31	32	33	34	35	36	37	38	39	40
0	0	0	0	0	0	0	0	0	0

41	42	43	44	45	46	47	48	49	50
0	0	0	0	0	0	0	0	0	0

51	52	53	54	55	56	57	58	59	60
0	0	0	0	0	0	0	0	0	0

61	62	63	64	65	66	67	68	69	70
0	0	0	0	0	0	0	0	0	0

71	72	73	74	75	76	77	78	79	80
0	0	0	0	0	0	0	0	0	0

```
 81   82   83   84   85   86   87   88   89   90
  0    0    0    0    0    0    0    0    0    0

 91   92   93   94   95   96   97   98   99  100
  0    0    0    0    0    0    0    0    0    0

THE TOTAL NUMBER OF WORDS IS    70
The Grand total of Words is 99
THE SIGNATURE LOOKS LIKE THIS
0010011101010100  1110000111010010  0100111111100101  0110010100100001   4
0100010100100101  0111100001110111  1111111011011100  0010010101001111   8
1101000011000000  0101000010010010  0101000110111110  1011100111000000  12
0010011001000001  1000010000010111  1001100001000000  0101101111000110  16
0110010011010011  0000000000000000  0110001000011101  0100100000010110  20
1101000101011011  1100100000011000  1101111101011000  0110000100000101  24
0110000100011101  1001000001001000  0000000000000000  0100000010010110  28
0000000000000000  0000000000000000  0000010000010010  0001010010100001  32
0100001000111100  0000000000000000  0101011001000001  1001110101100001  36
0110010100001101  1110100100010000  0000011001000001  0111010001111100  40
0000000000000000  1001000011000111  1111100111010100  0110010010010110  44
0000000000000000  0011010011101110  0000000000000000  0101011110100110  48
0110111101001100  0000000000000000  0010010000100101  0110010010011010  52
1001000001001000  0010010001001001  0000000000000000  0000011101000011  56
0000000000000000  0000000101000111  0110000100000100  0100011000011010  60
0110011000000101  0000000000000000  0000000000000000  0000010100000011  64
0001011000000001  0000000000000000  0000000000000000  0000000000000000  68
0101100111110000  0100000000111110  0000000000000000  0110001101011001  72
1101000111000000  0000000000000000  0000000000000000  0000010100000011  76
0001001100001100  0000000000000000  1001000001001000  1100011010110000  80
0111000001011100  0100000001011110  0000000000000000  1010100101001101  84
0000000000000000  0000000000000000  0000101100001101  0000010000101001  88
1001000011000000  0000010000101101  0000000000000000  0001000100100100  92
0000000000000000  0000000000000000  0101111111011110  0000000000000000  96
0000000000000000  0000000000000000  1000010000100101  0000000000000000  100

The total number of bits set is 452
The total number of 2 word false drops is 0
The total number of 3 word false drops is 0
The number of Bit Collisions is 114
```