Efficient Generation of Lexically Proper Palindromes

† Richard Rankin, † Hal Berghel, ‡ Xu Tielin
† Dept. of Computer Science, University of Arkansas
‡ Taiyuan University of Technology

Abstract:

Palindromes are strings of symbols which are symmetrical about the center. This paper outlines a method for generating certain types of palindromes, called lexical palindromes, which consist of legitimate English words. The method reported provides substantial pruning of a Prolog search tree by calculating the number of success nodes along certain search paths instead of visiting them, indexing words to improve database performance, and continuous analysis of current states to eliminate non-productive search paths. These efficiency measures allow lexical palindromes to be generated using a microcomputer.

Our ultimate objective is to be able to estimate the ratio of sentential palindromes for any given length to the set of syntactically well-formed English sentences of that length with respect to a given lexicon and grammatical model. We shall take up the problem of determining the ratio of sentential palindromes to well-formed sentences in a sequel to this paper.

Introduction:

Given the finite alphabet, V, a sentence over V is any finite string composed of symbols from V. The set of sentences made from symbols in V, we refer to as V* (V+ if we exclude the empty sentence.) A language L, over V is some subset of V*. Let a 'palindromic language' be one in which all sentences are symmetrical about the center, i.e. the first half of the sentence is the reverse of the second half of the sentence. Let L be the subset of V+ which consists of palindromes. Let L' be a subset of L such that for each palindrome in L', there is at least one way of partitioning the palindrome where each part of the partition is a correctly spelled word of the English language. We will refer to L as the set of orthographical palindromes, and to L' as the set of lexical palindromes. In addition, we define a sentential palindrome to be a lexical palindrome which is also a syntactically well-formed sentence of English.

Palindromes have been treated as both mystical [2] and trivial word games [1,3,4]. The term has been extended to cover symmetries unrelated to the alphabetic units with which they are normally associated [5]. In this paper, we are concerned with the generation of lexical palindromes over the Roman alphabet. We present here a technique to generate the lexical palindromes in an efficient manner.

There are an infinite number of palindromes which may be generated using any alphabet. Obviously, any palindrome can be expanded into another palindrome by adding the same character onto both ends of a string [3]. Just as clearly, there are an infinite number of lexical palindromes in English. Adding a palindromic word to opposite ends of any lexical palindrome will generate a new lexical palindrome. There are only a finite number of lexical palindromes, however, when palindromes are constrained by length and the size of the lexicon. It is this environment with which we are concerned.

General Discussion of The Problem:

The goal of this paper is to present an effective means for the generation of lexical palindromes, ignoring case and punctuation. The simplest strategy is to generate palindromes from an alphabet, and test to see if they are, in fact, lexical palindromes. Processing with the generate and test approach would begin with the random generation of a string, then proceed to a parsing/lookup system which would test to determine if the string is a lexical palindrome. For example, the system may generate '$s_1s_2s_3s_4$' where all s_i are elements of some symbol system. The parser might then attempt to partition the string so that each block contains a correctly spelled word from the lexicon. In this example, the partitions { <s_1,s_2,s_3,s_4>, <s_1s_2,s_3,s_4>, <$s_1s_2s_3$,s_4>, etc.} would each be checked to see if any of the possible partitions consists of legitimate words. If at least one of the partitions forms legitimate words, and the string is a palindrome, then the string is a lexical palindrome.

One of the problems of the character-oriented approach above is that there will be an enormous amount of time spent by the system partitioning and checking unacceptable output. One obvious improvement in the processing algorithm would be to check the generated string for palindromic properties before partitioning. This would eliminate many untenable search paths. The character approach basically applies a factorial process to an exponential output, resulting in a combinatorial explosion.

To ameliorate the problem of checking irrelevant output, one also could attempt a word-oriented approach. This, at the least, assures that substrings inserted are lexically correct since they are obtained from the lexicon. In this method, the palindrome begins as a string of variables of the length desired. A word is inserted beginning in the first available position, normally proceeding from left to right. The reverse of the word is then inserted at the corresponding mirror image position of the string. One knows, that at least half of the string, when read from either direction, consists of legitimate words, and that the string is symmetrical. For example, 'seekkees' is a palindrome which resulted from the insertion and reversal of an actual word. However, the string 'seekkees' is not a lexical palindrome since it cannot be entirely partitioned into legitimate words.

The basic problem, using a word-oriented approach, is to determine whether the reverse of words inserted conform to an acceptable lexical partition. In the example above, 'seek' should not have been inserted, because the resulting string, 'seekkees' cannot be partitioned into lexically correct segments. The string may initially be partitioned into {<seek,kees>}. 'Seek' is obviously in the dictionary, so an attempt would be made to partition 'kees' into words. This is not possible, so the system would backtrack and a new word will be attempted, but not until all possible partitions of 'kees' had been attempted.

Processing is generally from left to right within a string. Problems regarding the partitioning of the string in the reversed part of the list may not, therefore, be discovered until the process is much deeper into the search tree than the level at which the word was originally inserted. The reason for the late discovery of failure with left to right processing is that the system cannot attempt to analyze the second half of the string until after the first half of the string has been filled. The situation causing the failure may easily be the last character of the string, which was placed upon the insertion of the first word. If a problem has arisen at the end of the string which can never be resolved, backtracking may require an exponential number of steps before finally returning to the first word inserted, and changing it.

Therefore, what is needed is a system which takes advantage of the concept of working only with lexically correct insertions, but eliminates the problem of failure occurring deep in the search tree. The earlier a failure may be discovered in the search, the more unfruitful paths may be pruned without being explored.

Specific Discussion of the Problem:

Given an alphabet of n symbols, there are n^k distinct strings possible of length k. If n>1, not all strings will be palindromes. Therefore, the number of palindromes will be less than the n^k strings generated for testing under a simple method. When one considers all strings less than or equal to length k, then the total number of strings generated is:

$$\sum_{r=1}^{k} n^r$$

If one assumes an alphabet of 26 symbols, and a maximum string length of 7, then there are approximately $8*10^9$ strings which may be generated. The number of these strings which can be lexically correct can be calculated, if the distribution of the lexicon is known. Obviously, this makes the assumption that any word not present in the lexicon is not a correctly spelled word in the English language.

The combinatorial formula to calculate the number of possible lexical strings (LexicalStrings, below) of length 7, and the values of the variables used for the example provided in the text, is shown below.

$$\begin{aligned}\text{LexicalStrings} = {} & n_7 + \\ & (2!*n_6*n_1)+(2!*n_5*n_2)+((3!/2!)*n_5*n_1^2)+(2!*n_4*n_3)+ \\ & (3!*n_4*n_2*n_1)+(((4!/3!)*n_4*n_1^3)+((5!/4!)*n_3*n_1^4)+ \\ & ((3!/2!)*n_3^2*n_1)+((3!/2!)*n_3*n_2^2)+((4!/2!)*n_3*n_2*n_1^2)+ \\ & ((4!/3!)*n_2^3*n_1)+(((5!/(2!*3!))*n_2^2*n_1^3)+ \\ & ((6!/5!)*n_2*n_1^5)+ \\ & n_1^7\end{aligned}$$

Sample Lexicon Used:

n_x (x=length)	frequency
1	2
2	5
3	19
4	119
5	208
6	330
7	460

Using our lexicon of 4517 words, 1143 words are of length 7 or less, so that the upper bound for lexically correct strings with length 7 is 35917. Figures 1 and 2 show the length distribution of our test lexicons. The enormous discrepancy between strings which could be generated, and the upper bound of the number of these strings which can be lexically correct shows the advantage of a word oriented approach. If only insertions known in advance to be lexically correct are made, only $35917/8*10^9$ or .00043% of the possible strings need actually be checked - those composed of legitimate words. By dealing only with lexical entries as candidates for insertion, we effectively eliminate 99.99957% of the search paths.

Since lexical palindromes are a subset of lexically correct strings, the number of lexical palindromes, in this case, must be less than or equal to 35917. We have empirical evidence that there are 117 lexical palindromes of length 7 or less which can be constructed from the lexicon used. Therefore, in this case, lexical palindromes constitute .32% of the lexical strings possible with a word-oriented approach, but only .0000014% of the total strings which could be generated under a character approach. Although the success nodes remain proportionally few in number, the search space has been diminished by several orders of magnitude.

Palindrome Generation:

A description of the method chosen to generate palindromes for this paper follows. It involves both the analysis of palindrome generation, and the inclusion of efficiency measures. The analysis portion drives the processing methodology so that only potentially successful paths are pursued. Paths guaranteed to fail are quickly eliminated. Efficiency measures involve the storage and retrieval system developed to minimize the search space.

One of the design goals of the project was to efficiently generate palindromes on a microcomputer. The program was implemented using Arity Prolog version 5.0, and timed on an IBM AT clone, running at 10mHz, with 512k of primary memory, PC-DOS 3.2, and a 20 megabyte fixed disk.

Analytic Generation of Lexical Palindromes:

In our processing algorithm, two lists are maintained. The concatenation of these lists (even length palindromes), or their concatenation with a single character overlap (odd length palindromes) holds a palindrome. After each word insertion, the resulting string is

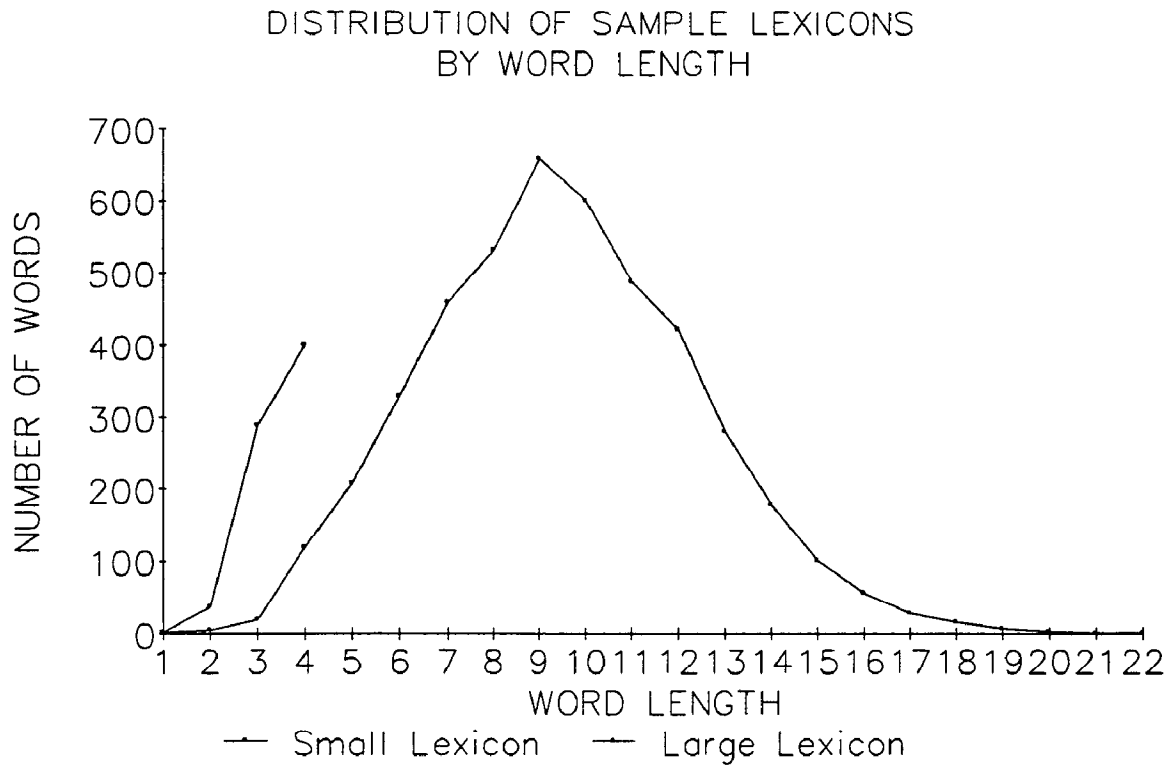

Figure 1

analyzed to provide an intelligent choice for the selection of the subsequent word to be inserted.

To begin, a word is selected from the lexicon and inserted into ForwardList. The word is reversed and placed in ReverseList. ReverseList is analyzed to see which of three conditions holds:

1) ReverseList can be partitioned so that all parts are words in the lexicon (EVEN LENGTH PALINDROMES);
2) ReverseList can be partitioned so that all parts are words in the lexicon, except that the last character in ForwardList is considered an intersection between ForwardList and ReverseList (ODD LENGTH PALINDROMES);
3) ReverseList can be partitioned so that all parts, except the leftmost partition, are words in the lexicon. (The empty word is considered to exist in the lexicon.) The leftmost partition is used as an index to locate the next word to be inserted. If no word ending in the leftmost partition can be found to insert, backtracking is invoked.

Example of case 1, even length palindrome:

1: ForwardList = [], ReverseList = [], looking for palindrome of length 6
2: insert 'top' into ForwardList and 'pot' into ReverseList
3: The concatenation of [top],[pot] results in a lexical palindrome

Example of case 2, odd length palindrome:

1: ForwardList = [], ReverseList = [], looking for a palindrome of length 5
2: insert 'tax' into ForwardList and 'xat' into ReverseList
3: The concatenation of [tax],[xat], allowing for the single character overlap, results in a lexical palindrome [tax,at]

Example of Case 3:

1: ForwardList = [], ReverseList = [], looking for a palindrome of length 10
2: insert: 'as'
3: ForwardList = [a,s], ReverseList = [s,a]
4: Partition ReverseList:
   4a: {<sa>} -> 'sa' is not in lexicon
   4b: {<s>,<a>} -> 'a' is in lexicon, but 's' (the leftmost partition) isn't
5: Use 's' as an index to find a word ending in 's' to be inserted
6: insert 'pots' in ReverseList, and its reverse into ForwardList (Since 's' already occurs in both lists, it is the excess characters of the word located which get inserted.)
7: ForwardList = [a,s,t,o,p], ReverseList = [p,o,t,s,a]
8: At this point, we have a lexical palindrome with the partitions: {<as>,<top>,<pots>,<a>}

In cases 1) and 2), the concatenation of ForwardList and ReverseList forms a valid palindrome. In case 3) the concatenation of the lists may result in a lexical palindrome only if the additional letters can be used to obtain another word from the lexicon which, when inserted, will lead to a palindrome. The additional letters are used to select the next word inserted, and processing continues. In case 3), a leftmost substring of a valid entry already inserted is used to obtain another word from the lexicon as the next candidate for insertion, and processing continues.

Once the above conditions are applied to ReverseList, either a palindrome is found of the requested length, or processing must continue. If processing continues, a word is selected and inserted for ReverseList, then its reverse is inserted into ForwardList. The conditions are then applied to ForwardList except that case 3) is concerned with the rightmost partition of ForwardList.

As can be noted from this description of the processing methodology, palindrome generation is accomplished from the outside of the string inwards. Since all characters inserted are derived from valid lexical entries, a palindrome, if found, must be lexically correct. The terminating conditions of this process is determined by a user-entered maximum length for palindromes to be generated. For example, an argument of 10 means that only palindromes of length less than or equal to 10 will be generated.

The importance of the outside-in processing is that insertions leading to failure are discovered as high in the search tree as possible, eliminating most of the backtracking. If a word has been inserted which must later lead to failure regardless of future insertions, this fact is discovered immediately due to the lack of an acceptable word as a candidate for the next insertion. This does not mean that every path undertaken will lead to success, but it does mean that paths guaranteed to fail will not be pursued.

As an example, assume we have a limited lexicon consisting of {seek, weed, ...} and we wish to form a palindrome of length 14. Under our system, the processing would insert 'seek' and analyze the situation as follows. Note that case 2 is never applied, because we are looking for a palindrome of even length.

1: ForwardList = [], ReverseList = []
2: Insert 'seek' into ForwardList and 'kees' into ReverseList
3: Analyze current situation:
   a) 'kees' is not in the lexicon (Case 1 fails)
   b) partition <k>, <ees>: 'ees' is not in the lexicon (partition attempt fails)
   c) partition <ke>,<es>: 'es' is not in the lexicon (partition attempt fails)
   d) partition <kee>,<s>: 's' is not in the lexicon (partition attempt fails)
   e) The rightmost partition consists of the empty word. Since the empty word is considered to be in the lexicon, this means that, to proceed towards a palindrome, 'kees' would have to be the end of a legitimate word, and is used as an index to locate such a word. A version of the lexicon indexed by the last character of words is maintained to improve the performance of these searches. A search of the lexicon results in the determination that no word ends in 'kees', therefore this path cannot result in a palindrome. Backtracking is invoked.
4: ForwardList = [], ReverseList = []
5: Insert next word ('weed') and try again

Under this method, since the insertion of 'seek' is guaranteed to lead to failure further down in the search tree, the path is pruned immediately. Without this heuristic, backtracking would be invoked the same number of times as the number of length 3 words in the lexicon (attempting to fill the string to length 14), before reaching the point where 'seek' is discarded.

Improving the Efficiency of Palindromic Generation:

Improving the efficiency of palindromic generation for this project fell into two general categories, eliminating valid yet unnecessary palindromes, and search method improvement. The first technique prunes sections of the search tree by calculating the number of palindromes eliminated during pruning, rather than visiting those nodes. The second is concerned with decreasing the size of the search space.

Eliminating The Exhaustive Generation of Palindromes:

When case 1) or Case 2), above, holds, a palindrome has been found and that palindrome is saved into a file. The palindromes generated from Case 1), though, have additional properties of interest. Suppose that a palindrome is of even length, and word partitions divide along the midpoint into lexical entries. For example, [no,on], [a,a], and [may,me,it,tie,my,am], when separated into ForwardList and ReverseList, do not overlap. Both lists contain only complete words, and their concatenation results in a palindrome. Palindromes of this type, which are generated by Case 1), above, are formed from even length sublists [3].

Anytime we have an even length palindrome of length n, a palindrome of length k, and a maximum length constraint w, with (n+k)<w, then the k length palindrome may be inserted into the middle of the even length palindrome to make another palindrome of length n+k. If one can determine the number of even length palindromes and their lengths, it is possible to skip the actual generation of some palindromes, yet still know how many there should have been in the portion of the search tree eliminated.

Suppose our regular palindrome file (file 1) contains [in,i] and [am,a]. We then discover the palindrome [no,on]. We would add this new palindrome to the file, so that we now have 3 entries. We would also write the new palindrome to a second file (file 2), since it meets our criteria of even in length. This is done in lieu of continuing the traversal of the search tree to obtain [no,am,a,on], and [no,in,i,on]. Assuming that these are all the palindromes found using our program, with a length constraint of 7, how many total palindromes are there? Obviously, we have the three palindromes in file 1. Within a length constraint of 7, however, we could also have made [no,am,a,on], and [no,in,i,on] for a total of 5 palindromes. (Constructing [no,no,on,on] fails because of the length constraint.)

The total number of palindromes can be determined simply by taking the number of palindromes in file 1, and adding the constructions which could have been made by combining entries from file 2 with entries from file 1, and weren't. Obviously, the total palindromes of length n or less is :

$$\sum_{i=1}^{n} T(i)$$

where T(i) is the total number of palindromes of length i. What may not be as obvious is how we determine T(i) when we haven't generated all the palindromes. The method of calculating the results is:

R(n) The number of palindromes with n characters in file 2 (even length)
N(n) The number of palindromes with n characters in file 1
T(n) The total number of palindromes with n characters

$T(1) = N(1)$
$T(2) = N(2)$
$T(3) = N(3) + R(2) * T(1)$
$T(4) = N(4) + R(2) * T(2)$
$T(5) = N(5) + R(2) * T(3) + R(4) * T(1)$
$T(6) = N(6) + R(2) * T(4) + R(4) * T(2)$
$T(7) = N(7) + R(2) * T(5) + R(4) * T(3) + R(6) * T(1)$
$T(8) = N(8) + R(2) * T(6) + R(4) * T(4) + R(6) * T(2)$
.....

When n is odd:

$$T(n) = N(n) + R(2) * T(n-2) + R(4) * T(n-4) + \ldots + R(n-1) * T(1)$$

When n is even:

$$T(n) = N(n) + R(2) * T(n-2) + R(4) * T(n-4) + \ldots + R(n-2) * T(2)$$

This method eliminates slow search tree processing with a substitution of simple calculation. Thus, when counting is the primary goal of palindrome generation, we may accomplish this goal without traversing the complete search tree. If the palindromes themselves are of interest, the two files could be combined after program termination, or one may slightly alter our program to remove this efficiency measure, and perform complete search tree traversals. The number of search tree paths actually eliminated is a function of the size of the lexicon and the length constraint.

Search Method Improvements:

In general, the highest cost of generating palindromes derives from the characteristics of Prolog and its search tree mechanism. At the highest level, the palindrome generation process may be considered as a tree traversal. If we assume that there are 1000 words in the lexicon, for example, the problem may be represented as a tree, with 1000 arcs leaving each node.

Step one is to select a word from the lexicon. This initial root of the tree has 1000 branches, one for each word. Once a particular word is selected, a second word, out of 1000 possibilities, must be selected from the lexicon. The second word is tested to determine if it may provide a path to a success node containing a palindrome. There are, however, an additional 1000 branches from this node so that the selection of a third word involves the selection of one out of 1000 more branches, etc. If, for example, five words need to be selected from the lexicon to compose a palindrome, the worst case search involves 1000^5 nodes to derive all possible palindromes from the lexicon. This is untenable if one desires realistically finite runtimes.

Our solution involved the elimination of as many useless paths as possible, as early as possible in the search process. Every word eliminated as a candidate for the n^{th} word in a k-word palindrome eliminates 1000^{k-n} nodes from the search tree.

The method used to eliminate unsuccessful paths relies on indexing the lexicon and a constant analysis of the situations in case 3) above. First, the lexicon was divided into sub-groups indexed by the first

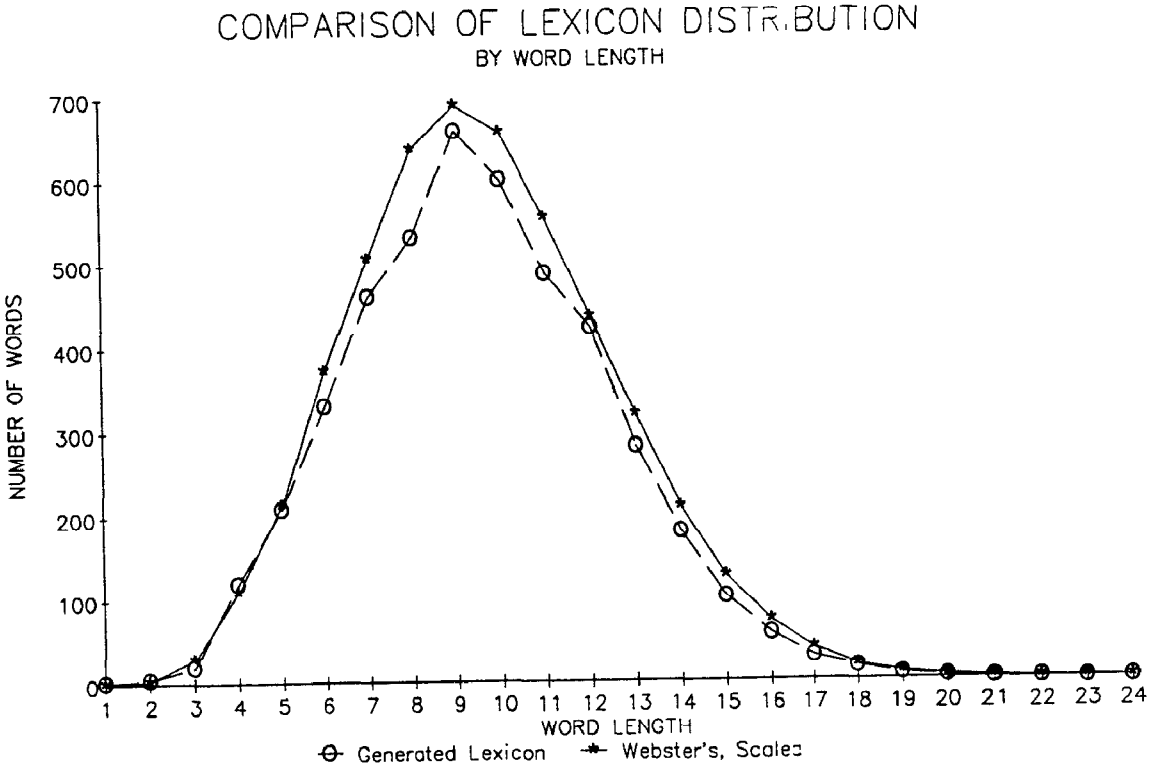

Figure 2

letter of the word. A second index was constructed and grouped by the first letter of the reversed words. This resulted in two lexical indices, one indexed by the first letter of each word, and a second indexed by the last letter of each word.

This optimization of the lexicon aids in eliminating potentially wasteful paths in the search tree. In case 3), above, some word has been inserted, part of which contributes to a palindrome, and part of which consists of 'leftover' letters. Any further attempt to achieve a complete palindrome must account for these extra letters which have been introduced. Merely attempting to insert another word from front to back, and checking to see if it accounts for the extra letters is a wasteful, brute-force approach which requires a full test of the lexicon. We use the extra letters and the reverse index to select a word which must fit properly. This identifies only potentially valid word paths from the analysis of the reversed insertion.

Conclusion:

We have shown a procedure for generating English lexical palindromes of various lengths. This procedure offers several improvements over a brute-force generate and test system with significant search tree pruning. These improvements utilize a method of calculation to eliminate some paths in the search tree, indexing to improve database performance, and stepwise analysis of a current state to eliminate obviously impossible search tree branching.

Future research in this area is expected to pursue a determination of the frequency of lexical palindromes which also are sentential palindromes.

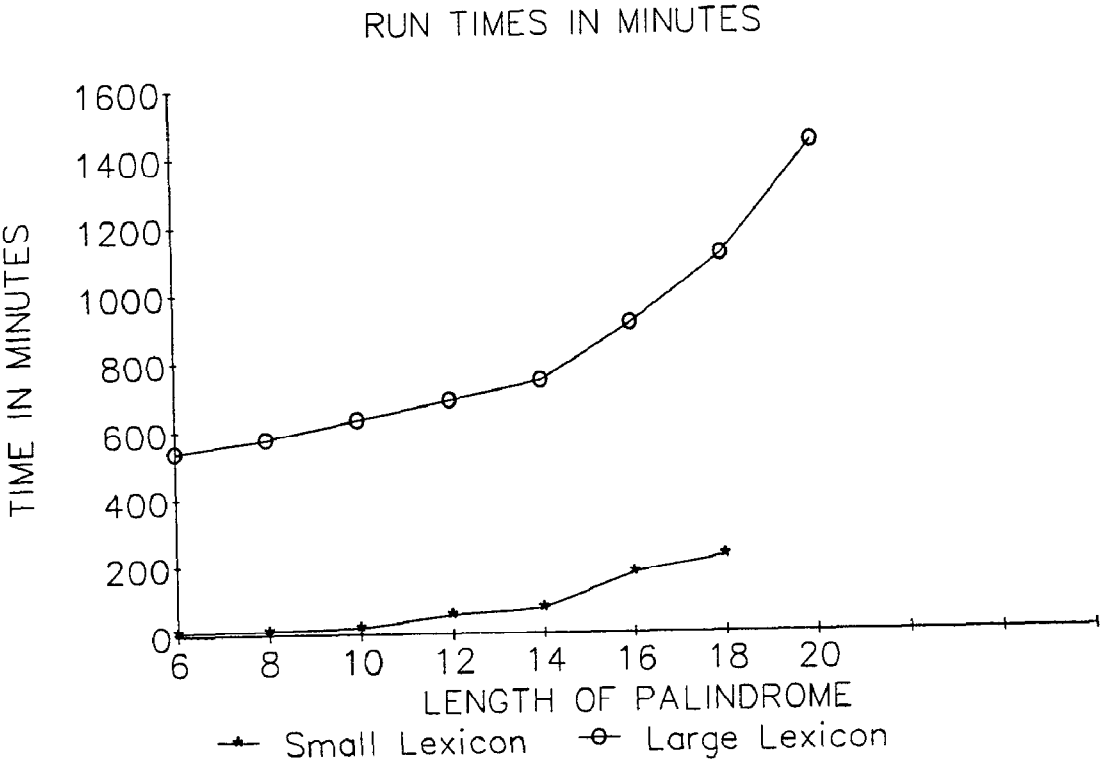

Figure 3

The use of the indices insures that only words beginning with the proper letter will ever be attempted. By checking for congruence in the index of the last letters, one quickly determines which subset of available words could possibly be candidates for the next insertion from the front of the string. If none are available, failure occurs and backtracking is invoked immediately to change the last word inserted from the front of the string.

If a candidate is available, it is chosen for insertion. The insertion, in turn, invokes a new analysis of the string built so far. Through this means, although a completely successful path is not guaranteed, paths guaranteed to fail by not accounting for letters already present in the string, are quickly eliminated.

In a lexicon evenly distributed by first letter of each word and containing 1000 words, the potential worst case for a particular palindrome requiring five words is now approximately 40^5 $(1000/26)^5$ instead of 1000^5. This improvement is provided through indexing the words by first and last letters, and using this resulting subset to locate potentially fruitful candidates for insertion.

Figure 3 shows the run times of our program, using different lexicons, and different palindrome length constraints.

REFERENCES

[1] Borgmann, D., "The Majestic Palindrome", Word Ways, Vol. 18, No. 1, Feb., 1985, pp6-15.

[2] Borgmann, D., "Palindromes: The Ascending Tradition", Word Ways, Vol. 13, No. 2, May, 1980, pp91-101.

[3] Funt, R., "Notes on Palindromes", Word Ways, Vol. 10, No. 4, Nov., 1977, pp. 232-234.

[4] Irvine, W., and S. Guarnaccia, Madam, I'm Adam and Other Palindromes, Charles Scribner's Sons, NY, 1987.

[5] Ranta, J., "Palindromes, Poems, and Geometric Form", College English, Vol. 36, No. 2, Oct., 1974, pp.161-172.