August 1983

Report No. STAN-CS-83-977

Word Hy-phen-a-tion by Com-put-er

by

Franklin Mark Liang

Department of Computer Science

Stanford University
Stanford, CA 94305

WORD HY-PHEN-A-TION BY COM-PUT-ER

Franklin Mark Liang
Department of Computer Science
Stanford University
Stanford, California 94305

Abstract

This thesis describes research leading to an improved word hyphenation algorithm for the TEX82 typesetting system. Hyphenation is viewed primarily as a data compression problem, where we are given a dictionary of words with allowable division points, and try to devise methods that take advantage of the large amount of redundancy present.

The new hyphenation algorithm is based on the idea of hyphenating and inhibiting patterns. These are simply strings of letters that, when they match in a word, give us information about hyphenation at some point in the pattern. For example, '`-tion`' and '`c-c`' are good hyphenating patterns. An important feature of this method is that a suitable set of patterns can be extracted automatically from the dictionary.

In order to represent the set of patterns in a compact form that is also reasonably efficient for searching, the author has developed a new data structure called a packed trie. This data structure allows the very fast search times characteristic of indexed tries, but in many cases it entirely eliminates the wasted space for null links usually present in such tries. We demonstrate the versatility and practical advantages of this data structure by using a variant of it as the critical component of the program that generates the patterns from the dictionary.

The resulting hyphenation algorithm uses about 4500 patterns that compile into a packed trie occupying 25K bytes of storage. These patterns find 89% of the hyphens in a pocket dictionary word list, with essentially no error. By comparison, the uncompressed dictionary occupies over 500K bytes.

This research was supported in part by the National Science Foundation under grants IST-82-01926 and MSC-83-00984, and by the System Development Foundation. 'TEX' is a trademark of the American Mathematical Society.

WORD HY-PHEN-A-TION
BY COM-PUT-ER

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE

AND THE COMMITTEE ON GRADUATE STUDIES

OF STANFORD UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

by

Franklin Mark Liang

June 1983

Acknowledgments

I am greatly indebted to my adviser, Donald Knuth, for creating the research environment that made this work possible. When I began work on the TEX project as a summer job, I would not have predicted that computer typesetting would become such an active area of computer science research. Prof. Knuth's foresight was to recognize that there were a number of fascinating problems in the field waiting to be explored, and his pioneering efforts have stimulated many others to think about these problems.

I am also grateful to the Stanford Computer Science Department for providing the facilities and the community that have formed the major part of my life for the past several years.

I thank my readers, Luis Trabb Pardo and John Gill, as well as Leo Guibas who served on my orals committee on short notice.

In addition, thanks to David Fuchs and Tom Pressburger for helpful advice and encouragement.

Finally, this thesis is dedicated to my parents, for whom the experience of pursuing a graduate degree has been perhaps even more traumatic than it was for myself.

Table of contents

Chapter 1

Introduction

The work described in this thesis was inspired by the need for a word hyphenation routine as part of Don Knuth's TEX typesetting system [1]. This system was initially designed in order to typeset Prof. Knuth's seven-volume series of books, *The Art of Computer Programming*, when he became dissatisfied with the quality of computer typesetting done by his publisher. Since Prof. Knuth's books were to be a definitive treatise on computer science, he could not bear to see his scholarly work presented in an inferior manner, when the degradation was entirely due to the fact that the material had been typeset by a computer!

Since then, TEX (also known as Tau Epsilon Chi, a system for technical text) has gained wide popularity, and it is being adopted by the American Mathematical Society, the world's largest publisher of mathematical literature, for use in its journals. TEX is distinctive among other systems for word processing/document preparation in its emphasis on the highest quality output, especially for technical material.

One necessary component of the system is a computer-based algorithm for hyphenating English words. This is part of the paragraph justification routine, and it is intended to eliminate the need for the user to specify word division points explicitly when they are necessary for good paragraph layout. Hyphenation occurs relatively infrequently in most book-format printing, but it becomes rather critical in narrow-column formats such as newspaper printing. Insufficient attention paid to this aspect of layout results in large expanses of unsightly white space, or (even worse) in words split at inappropriate points, e.g. `new-spaper`.

Hyphenation algorithms for existing typesetting systems are usually either rule-based or dictionary-based. Rule-based algorithms rely on a set of division rules such as given for English in the preface of Webster's Unabridged Dictionary [2]. These include recognition of common prefixes and suffixes, splitting between double consonants, and other more specialized rules. Some of the "rules" are not particularly

amenable to computer implementation; e.g. "split between the elements of a compound word". Rule-based schemes are inevitably subject to error, and they rarely cover all possible cases. In addition, the task of finding a suitable set of rules in the first place can be a difficult and lengthy project.

Dictionary-based routines simply store an entire word list along with the allowable division points. The obvious disadvantage of this method is the excessive storage required, as well as the slowing down of the justification process when the hyphenation routine needs to access a part of the dictionary on secondary store.

Examples

To demonstrate the importance of hyphenation, consider Figure 1, which shows a paragraph set in three different ways by TEX. The first example uses TEX's normal paragraph justification parameters, but with the hyphenation routine turned off. Because the line width in this example is rather narrow, TEX is unable to find an acceptable way of justifying the paragraph, resulting in the phenomenon known as an "overfull box".

One way to fix this problem is to increase the "stretchability" of the spaces between words, as shown in the second example. (TEX users: This was done by increasing the stretch component of spaceskip to .5em.) The right margin is now straight, as desired, but the overall spacing is somewhat loose.

In the third example, the hyphenation routine is turned on, and everything is beautiful.

In olden times when wishing still helped one, there lived a king whose daughters were all beautiful, but the youngest was so beautiful that the sun itself, which has seen so much, was astonished whenever it shone in her face. Close by the king's castle lay a great dark forest, and under an old lime-tree in the forest was a well, and when the day was very warm, the king's child went out into the forest and sat down by the side of the cool fountain, and when she was bored she took a golden ball, and threw it up on high and caught it, and this ball was her favorite plaything.

In olden times when wishing still helped one, there lived a king whose daughters were all beautiful, but the youngest was so beautiful that the sun itself, which has seen so much, was astonished whenever it shone in her face. Close by the king's castle lay a great dark forest, and under an old lime-tree in the forest was a well, and when the day was very warm, the king's child went out into the forest and sat down by the side of the cool fountain, and when she was bored she took a golden ball, and threw it up on high and caught it, and this ball was her favorite plaything.

In olden times when wish-ing still helped one, there lived a king whose daughters were all beautiful, but the youngest was so beautiful that the sun itself, which has seen so much, was as-tonished whenever it shone in her face. Close by the king's castle lay a great dark forest, and un-der an old lime-tree in the forest was a well, and when the day was very warm, the king's child went out into the forest and sat down by the side of the cool fountain, and when she was bored she took a golden ball, and threw it up on high and caught it, and this ball was her favorite plaything.

Figure 1. A typical paragraph with and without hyphenation.

sel-fadjoint	as-so-ciate	as-so-ci-ate
Pit-tsburgh	prog-ress	pro-gress
clearin-ghouse	rec-ord	re-cord
fun-draising	a-rith-me-tic	ar-ith-met-ic
ho-meowners	eve-ning	even-ing
playw-right	pe-ri-od-ic	per-i-o-dic
algori-thm		
walkth-rough	in-de-pen-dent	in-de-pend-ent
Re-agan	tri-bune	trib-une

Figure 2. Difficult hyphenations.

However, life is not always so simple. Figure 2 shows that hyphenation can be difficult. The first column shows erroneous hyphenations made by various typesetting systems (which shall remain nameless). The next group of examples are words that hyphenate differently depending on how they are used. This happens most commonly with words that can serve as both nouns and verbs. The last two examples show that different dictionaries do not always agree on hyphenation (in this case Webster's vs. American Heritage).

TEX and hyphenation

The original TEX hyphenation algorithm was designed by Prof. Knuth and the author in the summer of 1977. It is essentially a rule-based algorithm, with three main types of rules: (1) suffix removal, (2) prefix removal, and (3) vowel-consonant-consonant-vowel (vccv) breaking. The latter rule states that when the pattern 'vowel-consonant-consonant-vowel' appears in a word, we can in most cases split between the consonants. There are also many special case rules; for example, "break vowel-q" or "break after ck". Finally a small exception dictionary (about 300 words) is used to handle particularly objectionable errors made by the above rules, and to hyphenate certain common words (e.g. `pro-gram`) that are not split by the rules. The complete algorithm is described in Appendix H of the old TEX manual.

In practice, the above algorithm has served quite well. Although it does not find all possible division points in a word, it very rarely makes an error. Tests on a pocket dictionary word list indicate that about 40% of the allowable hyphen points are found, with 1% error (relative to the total number of hyphen points). The algorithm requires 4K 36-bit words of code, including the exception dictionary.

The goal of the present research was to develop a better hyphenation algorithm. By "better" we mean finding more hyphens, with little or no error, and using as little additional space as possible. Recall that one way to perform hyphenation is to simply store the entire dictionary. Thus we can view our task as a data compression problem. Since there is a good deal of redundancy in English, we can hope for substantial improvement over the straightforward representation.

Another goal was to automate the design of the algorithm as much as possible. The original TEX algorithm was developed mostly by hand, with a good deal of trial and error. Extending such a rule-based scheme to find the remaining hyphens seems very difficult. Furthermore such an effort must be repeated for each new language. The former approach can be a problem even for English, because pronunciation (and thus hyphenation) tends to change over time, and because different types of publication may call for different sets of admissible hyphens.

Time magazine algorithm

A number of approaches were considered, including methods that have been discussed in the literature or implemented in existing typesetting systems. One of the methods studied was the so-called Time magazine algorithm, which is table-based rather than rule-based.

The idea is to look at four letters surrounding each possible breakpoint, namely two letters preceding and two letters following the given point. However we do not want to store a table of $26^4 = 456{,}976$ entries representing all possible four-letter combinations. (In practice only about 15% of these four-letter combinations actually occur in English words, but it is not immediately obvious how to take advantage of this.)

Instead, the method uses three tables of size 26^2, corresponding to the two letters preceding, surrounding, and following a potential hyphen point. That is, if the letter pattern `wx-yz` occurs in a word, we look up three values corresponding to the letter pairs `wx`, `xy`, and `yz`, and use these values to determine if we can split the pattern.

What should the three tables contain? In the Time algorithm the table values were the probabilities that a hyphen could occur after, between, or before two given letters, respectively. The probability that the pattern `wx-yz` can be split is then estimated as the product of these three values (as if the probabilities were independent, which they aren't). Finally the estimated value is compared against a threshold to determine hyphenation. Figure 3 shows an example of hyphenation probabilities computed by this method.

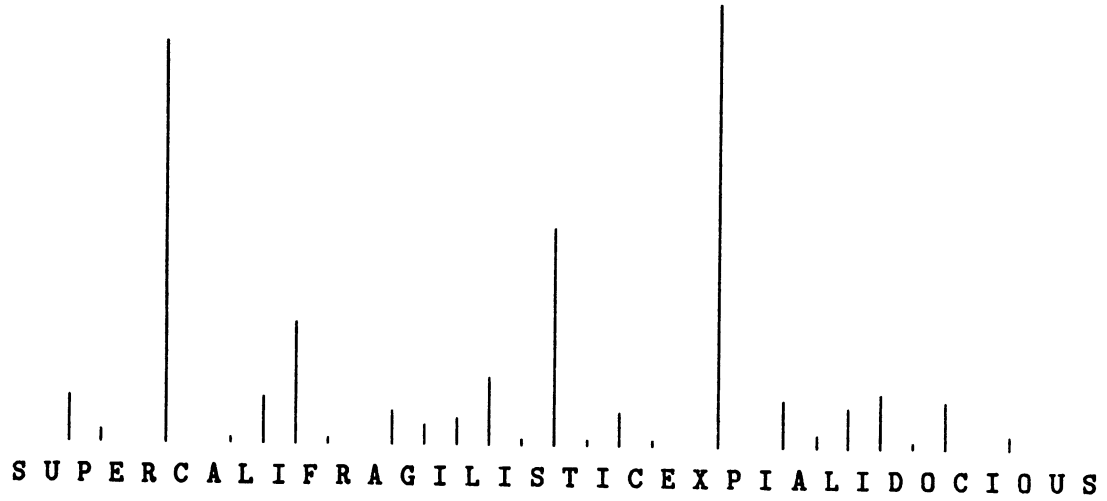


Figure 3. Hyphenation probabilities.

The advantage of this table-based approach is that the tables can be generated automatically from the dictionary. However, some experiments with the method yielded discouraging results. One estimate is 40% of the hyphens found, with 8% error. Thus a large exception dictionary would be required for good performance.

The reason for the limited performance of the above scheme is that just four letters of context surrounding the potential break point are not enough in many cases. In an extreme example, we might have to look as many as 10 letters ahead in order to determine hyphenation, e.g. `dem-on-stra-tion` vs. `de-mon-stra-tive`.

So a more powerful method is needed.

Patterns

A good deal of experimentation led the author to a more powerful method based on the idea of hyphenation *patterns*. These are simply strings of letters that, when they match in a word, will tell us how to hyphenate at some point in the pattern. For example, the pattern 'tion' might tell us that we can hyphenate before the 't'. Or when the pattern 'cc' appears in a word, we can usually hyphenate between the c's. Here are some more examples of good hyphenating patterns:

```
.in-d .in-s .in-t .un-d b-s -cia con-s con-t e-ly er-l er-m
ex- -ful it-t i-ty -less l-ly -ment n-co -ness n-f n-l n-si
n-v om-m -sion s-ly s-nes ti-ca x-p
```

(The character '.' matches the beginning or end of a word.)

Patterns have many advantages. They are a general form of "hyphenation rule" that can include prefix, suffix, and other rules as special cases. Patterns can even describe an exception dictionary, namely by using entire words as patterns. (Actually, patterns are often more concise than an exception dictionary because a single pattern can handle several variant forms of a word; e.g. `pro-gram`, `pro-grams`, and `pro-grammed`.)

More importantly, the pattern matching approach has proven very effective. An appropriate set of patterns captures very concisely the information needed to perform hyphenation. Yet the pattern rules are of simple enough form that they can be generated automatically from the dictionary.

When looking for good hyphenating patterns, we soon discover that almost all of them have some exceptions. Although `-tion` is a very "safe" pattern, it fails on the word `cat-ion`. Most other cases are less clear-cut; for example, the common pattern `n-t` can be hyphenated about 80 percent of the time. It definitely seems worthwhile to use such patterns, provided that we can deal with the exceptions in some manner.

After choosing a set of hyphenating patterns, we may end up with thousands of exceptions. These could be listed in an exception dictionary, but we soon notice there are many similarities among the exceptions. For example, in the original TEX algorithm we found that the vowel-consonant-consonant-vowel rule resulted in hundreds of errors of the form `X-Yer` or `X-Yers`, for certain consonant pairs `XY`, so we put in a new rule to prevent those errors.

Thus, there may be "rules" that can handle large classes of exceptions. To take advantage of this, patterns come to the rescue again; but this time they are *inhibiting* patterns, because they show where hyphens should *not* be placed. Some good examples of inhibiting patterns are: `b=ly` (don't break between `b` and `ly`), `bs=`, `=cing`, `io=n`, `i=tin`, `=ls`, `nn=`, `ns=t`, `n=ted`, `=pt`, `ti=al`, `=tly`, `=ts`, and `tt=`.

As it turns out, this approach is worth pursuing further. That is, after applying hyphenating and inhibiting patterns as discussed above, we might have *another* set of hyphenating patterns, then another set of inhibiting patterns, and so on. We can think of each level of patterns as being "exceptions to the exceptions" of the previous level. The current TEX82 algorithm uses five alternating levels of hyphenating and inhibiting patterns. The reasons for this will be explained in Chapter 4.

The idea of patterns is the basis of the new TEX hyphenation algorithm, and it was the inspiration for much of the intermediate investigation, that will be described.

Overview of thesis

In developing the pattern scheme, two main questions arose: (1) How can we represent the set of hyphenation patterns in a compact form that is also reasonably efficient for searching? (2) Given a hyphenated word list, how can we generate a suitable set of patterns?

To solve these problems, the author has developed a new data structure called a *packed trie*. This data structure allows the very fast search times characteristic of indexed tries, but in many cases it entirely eliminates the wasted space for null links usually present in such tries.

We will demonstrate the versatility and practical advantages of this data structure by using it not only to represent the hyphenation patterns in the final algorithm, but also as the critical component of the program that generates the patterns from the dictionary. Packed tries have many other potential applications, including identifier lookup, spelling checking, and lexicographic sorting.

Chapter 2 considers the simpler problem of recognizing, rather than hyphenating, a set of words such as a dictionary, and uses this problem to motivate and explain the advantages of the packed trie data structure. We also point out the close relationship between tries and finite-state machines.

Chapter 3 discusses ways of applying these ideas to hyphenation. After considering various approaches, including minimization with don't cares, we return to the idea of patterns.

Chapter 4 discusses the heuristic method used to select patterns, introduces dynamic packed tries, and describes some experiments with the pattern generation program.

Chapter 5 gives a brief history, and mentions ideas for future research.

Finally, the appendix contains the WEB [3] listing of the portable pattern generation program PATGEN, as well as the set of patterns currently used by TEX82.

Note: The present chapter has been typeset by giving unusual instructions to TEX so that it hyphenates words much more often than usual; therefore the reader can see numerous examples of word breaks that were discovered by the new algorithm.

Chapter 2

The dictionary problem

In this chapter we consider the problem of recognizing a set of words over an alphabet. To be more precise, an *alphabet* is a set of characters or symbols, for example the letters A through Z, or the ASCII character set. A *word* is a sequence of characters from the alphabet. Given a set of words, our problem is to design a data structure that will allow us to determine efficiently whether or not some word is in the set.

In particular, we will use spelling checking as an example throughout this chapter. This is a topic of interest in its own right, but we discuss it here because the pattern matching techniques we propose will turn out to be very useful in our hyphenation algorithm.

Our problem is a special case of the general set recognition problem, because the elements of our set have the additional structure of being variable-length sequences of symbols from a finite alphabet. This naturally suggests methods based on a character-by-character examination of the key, rather than methods that operate on the entire key at once. Also, the redundancy present in natural languages such as English suggests additional opportunities for compression of the set representation.

We will be especially interested in space minimization. Most data structures for set representation, including the one we propose, are reasonably fast for searching. That is, a search for a key doesn't take much more time than is needed to examine the key itself. However, most of these algorithms assume that everything is "in core", that is, in the primary memory of the computer. In many situations, such as our spelling checking example, this is not feasible. Since secondary memory access times are typically much longer, it is worthwhile to try compressing the data structure as much as possible.

In addition to determining whether a given word is in the set, there are other operations we might wish to perform on the set representation. The most basic are insertion and deletion of words from the set. More complicated operations include performing the union of two sets, partitioning a set according to some criterion,

determining which of several sets an element is a member of, or operations based on an ordering or other auxiliary information associated with the keys in the set. For the data structures we consider, we will pay some attention to methods for insertion and deletion, but we shall not discuss the more complicated operations.

We first survey some known methods for set representation, and then propose a new data structure called a "packed trie".

Data structures

Methods for set representation include the following: sequential lists, sorted lists, binary search trees, balanced trees, hashing, superimposed coding, bit vectors, and digital search trees (also known as tries). Good discussions of these data structures can be found in a number of texts, including Knuth [4], Standish [5], and AHU [6]. Below we make a few remarks about each of these representations.

A sequential list is the most straightforward representation. It requires both space and search time proportional to the number of characters in the dictionary.

A sorted list assumes an ordering on the keys, such as alphabetical order. Binary search allows the search time to be reduced to the logarithm of the size of the dictionary, but space is not reduced.

A binary search tree also allows search in logarithmic time. This can be thought of as a more flexible version of a sorted list that can be optimized in various ways. For example if the probabilities of searching for different keys in the tree are known, then the tree can be adapted to improve the expected search time. Search trees can also handle insertions and deletions easily, although an unfavorable sequence of such operations may degrade the performance of the tree.

Balanced tree schemes (including AVL trees, 2-3 trees, and B-trees) correct the above-mentioned problem, so that insertions, deletions, and searches can all be performed in logarithmic time in the worst case. Variants of trees have other nice properties, too; they allow merging and splitting of sets, and priority queue operations. B-trees are well-suited to large applications, because they are designed to minimize the number of secondary memory accesses required to perform a search. However, space utilization is not improved by any of these tree schemes, and in fact it is usually increased because of the need for extra pointers.

Hashing is an essentially different approach to the problem. Here a suitable randomizing function is used to compute the location at which a key is stored. Hashing methods are very fast on the average, although the worst case is linear; fortunately this worst case almost never happens.

An interesting variant of hashing, called superimposed coding, was proposed by Bloom [7] (see also [4, §6.5], [8]), and at last provides for reduction in space,

although at the expense of allowing some error. Since this method is perhaps less well known we give a description of it here.

Superimposed coding

The idea is as follows. We use a single large bit array, initialized to zeros, plus a suitable set of d different hash functions. To represent a word, we use the hash functions to compute d bit positions in the large array of bits, and set these bits to ones. We do this for each word in the set. Note that some bits may be set by more than one word.

To test if a word is in the set, we compute the d bit positions associated with the word as above, and check to see if they are all ones in the array. If any of them are zero, the word cannot be in the set, so we reject it. Otherwise if all of the bits are ones, we accept the word. However, some words not in the set might be erroneously accepted, if they happen to hash into bits that are all "covered" by words in the set.

It can be shown [7] that the above scheme makes the best use of space when the density of bits in the array, after all the words have been inserted, is approximately one-half. In this case the probability that a word not in the set is erroneously accepted is 2^{-d}. For example if each word is hashed into 4 bit positions, the error probability is 1/16. The required size of the bit array is approximately $nd \lg e$, where n is the number of items in the set, and $\lg e \approx 1.44$.

In fact Bloom specifically discusses automatic hyphenation as an application for his scheme! The scenario is as follows. Suppose we have a relatively compact routine for hyphenation that works correctly for 90 percent of the words in a large dictionary, but it is in error or fails to hyphenate the other 10 percent. We would then like some way to test if a word belongs to the 10 percent, but we do not have room to store all of these words in main memory. If we instead use the superimposed coding scheme to test for these words, the space required can be much reduced. For example with $d = 4$ we only need about 6 bits per word. The penalty is that some words will be erroneously identified as being in the 10 percent. However, this is acceptable because usually the test word will be rejected and we can then be sure that it is not one of the exceptions. (Either it is in the other 90 percent or it is not in the dictionary at all.) In the comparatively rare case that the word is accepted, we can go to secondary store, to check explicitly if the word is one of the exceptions.

The above technique is actually used in some commercial hyphenation routines. For now, however, TEX will not have an external dictionary. Instead we will require that our hyphenation routine be essentially free of error (although it may not achieve complete hyphenation).

An extreme case of superimposed coding should also be mentioned, namely the bit-vector representation of a set. (Imagine that each word is associated with a single bit position, and one bit is allocated for each possible word.) This representation is often very convenient, because it allows set intersection and union to be performed by simple logical operations. But it also requires space proportional to the size of the universe of the set, which is impractical for words longer than three or four characters.

Tries

The final class of data structures we will consider are the digital search trees, first described by de la Briandais [9] and Fredkin [10]. Fredkin also introduced the term "trie" for this class of trees. (The term was derived from the word re*trie*val, although it is now pronounced "try".)

Tries are distinct from the other data structures discussed so far because they explicitly assume that the keys are a *sequence* of values over some (finite) alphabet, rather than a single indivisible entity. Thus tries are particularly well-suited for handling variable-length keys. Also, when appropriately implemented, tries can provide compression of the set represented, because common prefixes of words are combined together; words with the same prefix follow the same search path in the trie.

A trie can be thought of as an m-ary tree, where m is the number of characters in the alphabet. A search is performed by examining the key one character at a time and using an m-way branch to follow the appropriate path in the trie, starting at the root.

We will use the set of 31 most common English words, shown below, to illustrate different ways of implementing a trie.

A	FOR	IN	THE
AND	FROM	IS	THIS
ARE	HAD	IT	TO
AS	HAVE	NOT	WAS
AT	HE	OF	WHICH
BE	HER	ON	WITH
BUT	HIS	OR	YOU
BY	I	THAT	

Figure 4. The 31 most common English words.

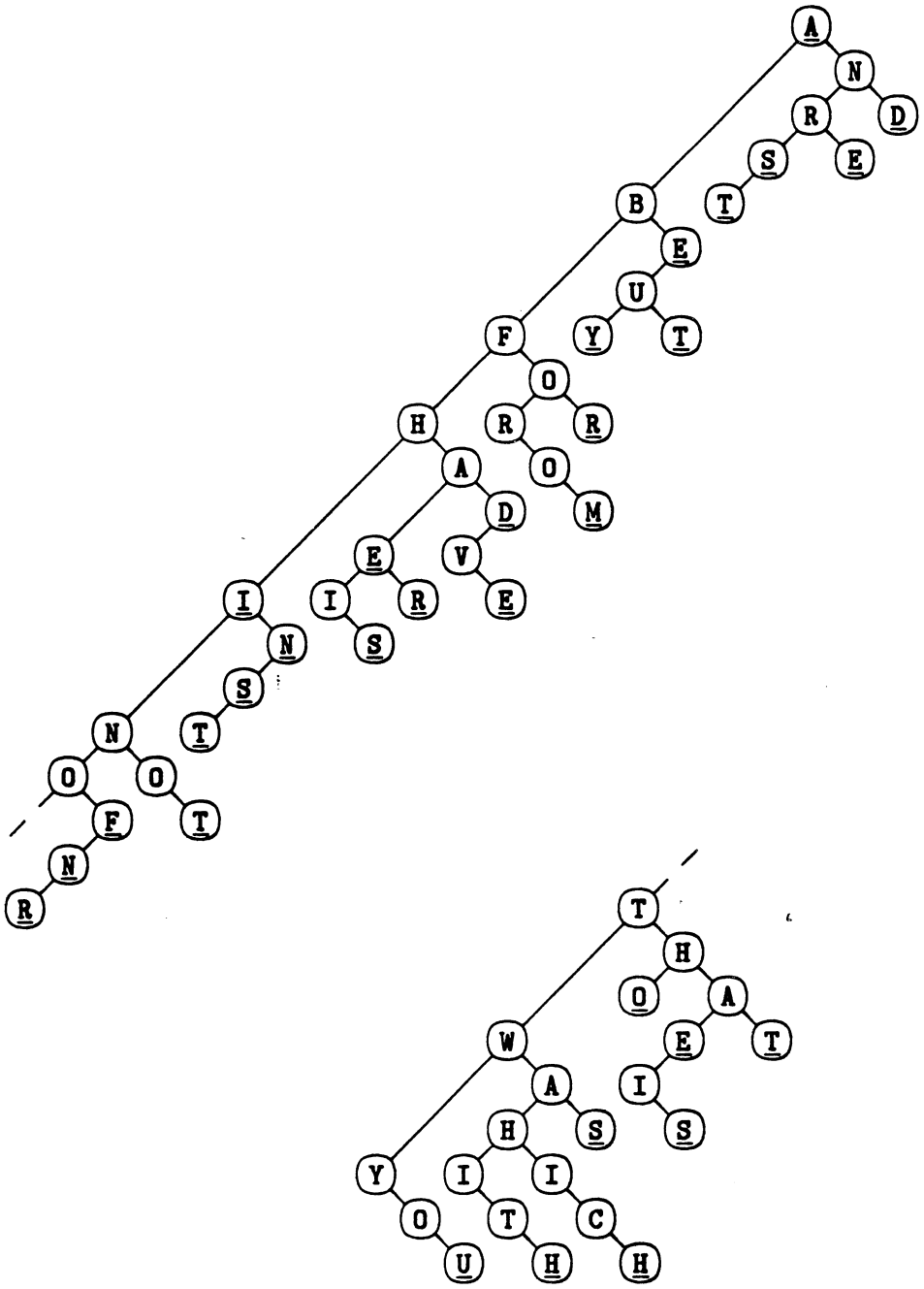


Figure 5. L ed trie for the 31 most common English words.

Figure 5 shows a *linked trie* representing this set of words. In a linked trie, the m-way branch is performed using a sequential series of comparisons. Thus in Figure 5 each node represents a yes-no test against a particular character. There are two link fields indicating the next node to take depending on the outcome of the test. On a 'yes' answer, we also move to the next character of the key. The underlined characters are terminal nodes, indicated by an extra bit in the node. If the word ends when we are at a terminal node, then the word is in the set.

Note that we do not have to actually store the keys in the trie, because each node implicitly represents a prefix of a word, namely the sequence of characters leading to that node.

A linked trie is somewhat slow because of the sequential testing required for each character of the key. The number of comparisons per character can be as large as m, the size of the alphabet. In addition, the two link fields per node are somewhat wasteful of space. (Under certain circumstances, it is possible to eliminate one of these two links. We will explain this later.)

In an *indexed trie*, the m-way branch is performed using an array of size m. The elements of the array are pointers indicating the next family of the trie to go to when the given character is scanned, where a "family" corresponds to the group of nodes in a linked trie for testing a particular character of the key. When performing a search in an indexed trie, the appropriate pointer can be accessed by simply indexing from the base of the array. Thus search will be quite fast.

But indexed tries typically waste a lot of space, because most of the arrays have only a few "valid" pointers (for words in the trie), with the rest of the links being null. This is especially common near the bottom of the trie. Figure 6 shows an indexed trie for the set of 31 common words. This representation requires $26 \times 32 = 832$ array locations, compared to 59 nodes for the linked trie.

Various methods have been proposed to remedy the disadvantages of linked and indexed tries. Trabb Pardo [11] describes and analyzes the space requirements of some simple variants of binary tries. Knuth [4, ex. 6.3–20] analyzes a composite method where an indexed trie is used for the first few levels of the trie, switching to sequential search when only a few keys remain in a subtrie. Mehlhorn [12] suggests using a binary search tree to represent each family of a trie. This requires storage proportional to the number of "valid" links, as in a linked trie, but allows each character of the key to be processed in at most $\log m$ comparisons. Maly [13] has proposed a "compressed trie" that uses an implicit representation to eliminate links entirely. Each level of the trie is represented by a bit array, where the bits indicate whether or not some word in the set passes through the node corresponding to

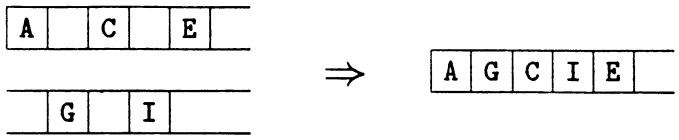
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
1	2	5				7		11	16					17	19					20			24		31	
2														3				4	0	0						
3				0																						
4					0																					
5					0																6				0	
6																				0						
7															8			9								
8																		0								
9															0											
10													0													
11	12				14				15																	
12				0																		13				
13					0																					
14																		0								
15																			0							
16														0					0	0						
17															18											
18																				0						
19						0								0				0								
20								21							0											
21	22				0				23																	
22																				0						
23																			0							
24	25							26	29																	
25																			0							
26									27																	
27			28																							
28								0																		
29																				30						
30								0																		
31															32											
32																					0					

Figure 6. Indexed trie for the 31 most common English words.

that bit. In addition each family contains a field indicating the number of nonzero bits in the array for all nodes to the left of the current family, so that we can find the desired family on the next level. The storage required for each family is thus reduced to $m + \log n$ bits, where n is the total number of keys. However, compressed tries cannot handle insertions and deletions easily, nor do they retain the speed of indexed tries.

Packed tries

Our idea is to use an indexed trie, but to save the space for null links by packing the different families of the trie into a single large array, so that links from one family may occupy space normally reserved for links for other families that happen to be null. An example of this is illustrated below.



(In the following, we will sometimes refer to families of the indexed trie as *states*, and pointers as *transitions*. This is by analogy with the terminology for finite-state machines.)

When performing a search in the trie, we need a way to check if an indexed pointer actually corresponds to the current family, or if it belongs to some other family that just happens to be packed in the same location. This is done by additionally storing the character indexing a transition along with that transition. Thus a transition belongs to a state only if its character matches the character we are indexing on. This test always works if one additional requirement is satisfied, namely that different states may not be packed at the same base location.

The trie can be packed using a first-fit method. That is, we pack the states one at a time, putting each state into the lowest-indexed location in which it will fit (not overlapping any previously packed transitions, nor at an already occupied base location). On numerous examples based on typical word lists, this heuristic works extremely well. In fact, nearly all of the holes in the trie are often filled by transitions from other states.

Figure 7 shows the result when the indexed trie of Figure 6 is packed into a single array using the first-fit method. (Actually we have used an additional compression technique called suffix compression before packing the trie; this will be explained in the next section.) The resulting trie fits into just 60 locations. Note

	0	1	2	3	4	5	6	7	8	9
00		A 8	B11			D 0	F 3	E 0	H30	I23
10	C 5			H 0	N25	O32	E 0		O12	M 0
20	T33	R14	N 1	W46	T 0	Y37	R 2	S 0	T 0	0 6
30	R 0	A29	U 4	D 0	S 0	E12	Y 0	N 0	F 0	I15
40	0 4	H44	S 0	T 0	I 7	A 4	N 0	A15	0 0	E 0
50	R 0	V 2	O38	I15	H35	I36	T 5			U 0

Figure 7. Packed trie for the 31 most common English words.

that the packed trie is a single large array; the rows in the figure should be viewed as one long row.

As an example, here's what happens when we search for the word HAVE in the packed trie. We associate the values 1 through 26 with the letters A through Z. The root of the trie is packed at location 0, so we begin by looking at location 8 corresponding to the letter H. Since 'H30' is stored there, this is a valid transition and we then go to location 30. Indexing by the letter A, we look in location 31, which tells us to go to 29. Now indexing by V gets location 51, which points to 2. Finally indexing by E gets location 7, which is underlined, indicating that the word HAVE is indeed in the set.

Suffix compression

A big advantage of the trie data structure is that common prefixes of words are combined automatically into common paths in the trie. This provides a good deal of compression. To save more space, we can try to take advantage of common suffixes.

One way of doing this is to construct a trie in the usual manner, and then merge common subtries together, starting from the leaves (lieves) and working upward. We call this process *suffix compression.*

For example, in the linked trie of Figure 5 the terminal nodes for the words HIS and THIS, both of which test for the letter S and have no successors, can be combined into a single node. That is, we can let their parent nodes both point to the same node; this does not change the set of words accepted by the trie. It turns out that we can then combine the parent nodes, since both of them test for I and go to the S node if successful, otherwise stop (no left successor). However, the grandparent nodes (which are actually siblings of the I nodes) cannot be combined even though they both test for E, because one of them goes to a terminal R node upon success, while the other has no right successor.

With a larger set of words, a great deal of merging can be possible. Clearly all leaf nodes (nodes with no successors) that test the same character can be combined together. This alone saves a number of nodes equal to the number of words in the dictionary, minus the number of words that are prefixes of other words, plus at most 26. In addition, as we might expect, longer suffixes such as -ly, -ing, or -tion can frequently be combined.

The suffix compression process may sound complicated, but actually it can be described by a simple recursive algorithm. For each node of the trie, we first compress each of its subtries, then determine if the node can be merged with some other node. In effect, we traverse the trie in depth-first order, checking each node to see if it is equivalent to any previously seen node. A hash table can be used to identify equivalent nodes, based on their (merged) transitions.

The identification of nodes is somewhat easier using a binary tree representation of the trie, rather than an m-ary representation, because each node will then have just two link fields in addition to the character and output bit. Thus it will be convenient to use a linked trie when performing suffix compression. The linked representation is also more convenient for constructing the trie in the first place, because of the ease of performing insertions.

After applying suffix compression, the trie can be converted to an indexed trie and packed as described previously. (We should remark that performing suffix compression on a linked trie can yield some additional compression, because trie families can be partially merged. However such compression is lost when the trie is converted to indexed form.)

The author has performed numerous experiments with the above ideas. The results for some representative word lists are shown in Table 1 below. The last three

columns show the number of nodes in the linked, suffix-compressed, and packed tries, respectively. Each transition of the packed trie consists of a pointer, a character, and a bit indicating if this is an accepting transition.

word list	words	characters	linked	compressed	packed
pascal	35	145	125	104	120
murray	2726	19,144	8039	4272	4285
pocket	31,036	247,612	92,339	38,619	38,638
unabrd	235,545	2,256,805	759,045	—	—

Table 1. Suffix-compressed packed tries.

The algorithms for building a linked trie, suffix compression, and first-fit packing are used in TEX82 to preprocess the set of hyphenation patterns into a packed trie used by the hyphenation routine. A WEB description of these algorithms can be found in [14].

Derived forms

Most dictionaries do not list the most common derived forms of words, namely regular plurals of nouns and verbs (`-s` forms), participles and gerunds of verbs (`-ed` and `-ing` forms), and comparatives and superlatives of adjectives (`-er` and `-est`). This makes sense, because a user of the dictionary can easily determine when a word possesses one of these regular forms. However, if we use the word list from a typical dictionary for spelling checking, we will be faced with the problem of determining when a word is one of these derived forms.

Some spelling checkers deal with this problem by attempting to recognize affixes. This is done not only for the derived forms mentioned above but other common variant forms as well, with the purpose of reducing the number of words that have to be stored in the dictionary. A set of logical rules is used to determine when certain prefixes and suffixes can be stripped from the word under consideration.

However such rules can be quite complicated, and they inevitably make errors. The situation is not unlike that of finding rules for hyphenation, which should not be surprising, since affix recognition is an important part of any rule-based hyphenation algorithm. This problem has been studied in some detail in a series of papers by Resnikoff and Dolby [15].

Since affix recognition is difficult, it is preferable to base a spelling checker on a complete word list, including all derived forms. However, a lot of additional space will be required to store all of these forms, even though much of the added data is

redundant. We might hope that some appropriate method could provide substantial compression of the expanded word list. It turns out that suffix-compressed tries handle this quite well. When derived forms were added to our pocket dictionary word list, it increased in size to 49,858 words and 404,946 characters, but the resulting packed trie only increased to 46,553 transitions (compare the pocket dictionary statistics in Table 1).

Hyphenation programs also need to deal with the problem of derived forms. In our pattern-matching approach, we intend to extract the hyphenation rules automatically from the dictionary. Thus it is again preferable for our word list to include all derived forms.

The creation of such an expanded word list required a good deal of work. The author had access to a computer-readable copy of Webster's Pocket Dictionary [16], including parts of speech and definitions. This made it feasible to identify nouns, verbs, etc., and to generate the appropriate derived forms mechanically. Unfortunately the resulting word lists required extensive editing to eliminate many never-used or somewhat nonsensical derived forms, e.g. '`informations`'.

Spelling checkers

Computer-based word processing systems have recently come into widespread use. As a result there has been a surge of interest in programs for automatic spelling checking and correction. Here we will consider the dictionary representations used by some existing spelling checkers.

One of the earliest programs, designed for a large timesharing computer, was the DEC-10 SPELL program written by Ralph Gorin [17]. It uses a 12,000 word dictionary stored in main memory. A simple hash function assigns a unique 'bucket' to each word depending on its length and the first two characters. Words in the same bucket are listed sequentially. The number of words in each bucket is relatively small (typically 5 to 50 words), so this representation is fairly efficient for searching. In addition, the buckets provide convenient access to groups of similar words; this is useful when the program tries to correct spelling errors.

The dictionary used by SPELL does not contain derived forms. Instead some simple affix stripping rules are normally used; the author of the program notes that these are "error-prone".

Another spelling checker is described by James L. Peterson [18]. His program uses three separate dictionaries: (1) a small list of 258 common English words, (2) a dynamic 'cache' of about 1000 document-specific words, and (3) a large, comprehensive dictionary, stored on disk. The list of common words (which is static) is represented using a suffix-compressed linked trie. The dynamic cache is maintained

using a hash table. Both of these dictionaries are kept in main memory for speed. The disk dictionary uses an in-core index, so that at most one disk access is required per search.

Robert Nix [19] describes a spelling checker based on the superimposed coding method. He reports that this method allows the dictionary from the SPELL program to be compressed to just 20 percent of its original size, while allowing 0.1% chance of error.

A considerably different approach to spelling checking was taken by the TYPO program developed at Bell Labs [20]. This program uses digram and trigram frequencies to identify "improbable" words. After processing a document, the words are listed in order of decreasing improbability for the user to peruse. (Words appearing in a list of 2726 common technical words are not shown.) The authors report that this format is "psychologically rewarding", because many errors are found at the beginning, inducing the user to continue scanning the list until errors become rare.

In addition to the above, there have recently been a number of spelling checkers developed for the "personal computer" market. Because these programs run on small microprocessor-based systems, it is especially important to reduce the size of the dictionary. Standard techniques include hash coding (allowing some error), in-core caches of common words, and special codes for common prefixes and suffixes. One program first constructs a sorted list of all words in the document, and then compares this list with the dictionary in a single sequential pass. The dictionary can then be stored in a compact form suited for sequential scanning, where each word is represented by its difference from the previous word.

Besides simply detecting when words are not in a dictionary, the design of a practical spelling checker involves a number of other issues. For example many spelling checkers also try to perform spelling *correction*. This is usually done by searching the dictionary for words similar to the misspelled word. Errors and suggested replacements can be presented in an interactive fashion, allowing the user to see the context from the document and make the necessary changes. The contents of the dictionary are of course very important, and each user may want to modify the word list to match his or her own vocabulary. Finally, a plain spelling checker cannot detect problems such as incorrect word usage or mistakes in grammar; a more sophisticated program performing syntactic and perhaps semantic analysis of the text would be necessary.

Conclusion and related ideas

The dictionary problem is a fundamental problem of computer science, and it has many applications besides spelling checking. Most data structures for this problem consider the elements of the set as atomic entities, fitting into a single computer word. However in many applications, particularly word processing, the keys are actually variable-length strings of characters. Most of the standard techniques are somewhat awkward when dealing with variable length keys. Only the trie data structure is well-suited for this situation.

We have proposed a variant of tries that we call a packed trie. Search in a packed trie is performed by indexing, and it is therefore very fast. The first-fit packing technique usually produces a fairly compact representation as well.

We have not discussed how to perform dynamic insertions and deletions with a packed trie. In Chapter 4 we discuss a way to handle this problem, when no suffix compression is used, by repacking states when necessary.

The idea of suffix compression is not new. As mentioned, Peterson's spelling checker uses this idea also. But in fact, if we view our trie as a finite-state machine, suffix compression is equivalent to the well-known idea of state minimization. In our case the machine is acyclic, that is, it has no loops.

Suffix compression is also closely related to the common subexpression problem from compiler theory. In particular, it can be considered a special case of a problem called acyclic congruence closure, which has been studied by Downey, Sethi, and Tarjan [21]. They give a linear-time algorithm for suffix compression that does not use hashing, but it is somewhat complicated to implement and requires additional data structures.

The idea for the first-fit packing method was inspired by the paper "Storing a sparse table" by Tarjan and Yao [22]. The technique has been used for compressing parsing tables, as discussed by Zeigler [23] (see also [24]). However, our packed trie implementation differs somewhat from the applications discussed in the above references, because of our emphasis on space minimization. In particular, the idea of storing the character that indexes a transition, along with that transition, seems to be new. This has an advantage over other techniques for distinguishing states, such as the use of back pointers, because the character requires fewer bits.

The paper by Tarjan and Yao also contains an interesting theorem characterizing the performance of the first-fit packing method. They consider a modification suggested by Zeigler, where the states are first sorted into decreasing order based on the number of non-null transitions in each state. The idea is that small states, which can be packed more easily, will be saved to the end. They prove that if the

distribution of transitions among states satisfies a "harmonic decay" condition, then essentially all of the holes in the first-fit packing will be filled.

More precisely, let $n(l)$ be the total number of non-null transitions in states with more than l transitions, for $l \geq 0$. If the harmonic decay property $n(l) \leq n/(l+1)$ is satisfied, then the first-fit-decreasing packing satisfies $0 \leq b(i) \leq n$ for all i, where $n = n(0)$ is the total number of transitions and $b(i)$ is the base location at which the ith state is packed.

The above theorem does not take into account our additional restriction that no two states may be packed at the same base location. When the proof is modified to include this restriction, the bound goes up by a factor of two. However in practice we seem to be able to do much better.

The main reason for the good performance of the first-fit packing scheme is the fact that there are usually enough single-transition states to fill in the holes created by larger states. It is not really necessary to sort the states by number of transitions; any packing order that distributes large and small states fairly evenly will work well. We have found it convenient simply to use the order obtained by traversing the linked trie.

Improvements on the algorithms discussed in this chapter are possible in certain cases. If we store a linked trie in a specific traversal order, we can eliminate one of the link fields. For example, if we list the nodes of the trie in preorder, the left successor of a node will always appear immediately after that node. An extra bit is used to indicate that a node has no left successor. Of course this technique works for other types of trees as well.

If the word list is already sorted, linked trie insertion can be performed with only a small portion of the trie in memory at any time, namely the portion along the current insertion path. This can be a great advantage if we are are processing a large dictionary and cannot store the entire linked trie in memory.

Chapter 3

Hyphenation

Let us now try to apply the ideas of the previous chapter to the problem of hyphenation. TEX82 will use the pattern matching method described in Chapter 1, but we shall first discuss some related approaches that were considered.

Finite-state machines with output

We can modify our trie-based dictionary representation to perform hyphenation by changing the output of the trie (or finite-state machine) to a multiple-valued output indicating how the word can be hyphenated, instead of just a binary yes-no output indicating whether or not the word is in the dictionary. That is, instead of associating a single bit with each trie transition, we would have a larger "output" field indicating the hyphenation "action" to be taken on this transition. Thus on recognizing the word `hy-phen-a-tion`, the output would say "you can hyphenate this word after the second, sixth, or seventh letters".

To represent the hyphenation output, we could simply list the hyphen positions, or we could use a bit vector indicating the allowable hyphen points. Since there are only a few hundred different outputs and most of them occur many times, we can save some space by assigning each output a unique code and storing the actual hyphen positions in a separate table.

To conveniently handle the variable number of hyphen positions in outputs, we will use a linked representation that allows different outputs to share common portions of their output lists. This is implemented using a hash table containing pairs of the form (*output*, *next*), where *output* is a hyphenation position and *next* is a (possibly null) pointer to another entry in the table. To add a new output list to the table, we hash each of its outputs in turn, making each output point to the previous one. Interestingly, this process is quite similar to suffix compression.

The trie with hyphenation output can be suffix-compressed and packed in the same manner as discussed in Chapter 2. Because of the greater variety of outputs more of the subtries will be distinct, and there is somewhat less compression.

From our pocket dictionary (with hyphens), for example, we obtained a packed trie occupying 51,699 locations.

We can improve things slightly by "pushing outputs forward". That is, we can output partial hyphenations as soon as possible instead of waiting until the end of the word. This allows some additional suffix compression.

For example, upon scanning the letters hyph at the beginning of a word, we can already say "hyphenate after the second letter" because this is allowed for all words beginning with those letters. Note we could not say this after scanning just hyp, because of words like hyp-not-ic. Upon further scanning ena, we can say "hyphenate after the sixth letter".

When implementing this idea, we run into a small problem. There are quite a few words that are prefixes of other words, but hyphenate differently on the letters they have in common, e.g. ca-ret and care-tak-er, or as-pi-rin and as-pir-ing. To avoid losing hyphenation output, we could have a separate output whenever an end-of-word bit appears, but a simpler method is to append an end-of-word character to each word before inserting it into the trie. This increases the size of the linked trie considerably, but suffix compression merges most of these nodes together.

With the above modifications, the packed trie for the pocket dictionary was reduced to 44,128 transitions.

Although we have obtained substantial compression of the dictionary, the result is still too large for our purposes. The problem is that as long as we insist that only words in the dictionary be hyphenated, we cannot hope to reduce the space required to below that needed for spelling checking alone. So we must give up this restriction.

For example, we could eliminate the end-of-word bit. Then after pushing outputs forward, we can prune branches of the trie for which there is no further output. This would reduce the pocket dictionary trie to 35,429 transitions.

Minimization with don't cares

In this section we describe a more drastic approach to compression that takes advantage of situations where we "don't care" what the algorithm does.

As previously noted, most of the states in an indexed trie are quite sparse; that is, only a few of the characters have explicit transitions. Since the missing transitions are never accessed by words in our dictionary, we can allow them to be filled by arbitrary transitions.

This should not be confused with the overlapping of states that may occur in the trie-packing process. Instead, we mean that the added transitions will actually become part of the state.

There are two ways in which this might allow us to save more space in the minimization process. First, states no longer have to be identical in order to be merged; they only have to agree on those characters where both (or all) have explicit transitions. Second, the merging of non-equivalent states may allow further merging that was not previously possible, because some transitions have now become equivalent.

For example, consider again the trie of Figure 5. When discussing suffix compression, we noted that the terminal S nodes for the words HIS and THIS could be merged together, but that the parent chains, each containing transitions for A, E, and I, could not be completely merged. However, in minimization with don't cares these two states can be merged. Note that such a merge will require that the DV state below the first A be merged with the T below the second A; this can be done because those states have no overlapping transitions.

As another example, notice that if the word AN were added to our vocabulary, then the NRST chain succeeding the root A node could be merged with the NST chain below the initial I node. (Actually, it doesn't make much sense to do minimization with don't cares on a trie used to recognize words in a dictionary, but we will ignore that objection for the purposes of this example.)

Unfortunately, trie minimization with don't cares seems more complicated than the suffix-compression process of Chapter 2. The problem is that states can be merged in more than one way. That is, the collection of mergeable states no longer forms an equivalence relation, as in regular finite-state minimization. In fact, we can sometimes obtain additional compression by allowing the same state to appear more than once. Another complication is that don't care merges can introduce loops into our trie.

Thus it seems that finding the minimum size trie will be difficult. Pfleeger [25] has shown this problem to be NP-complete, by transformation from graph coloring; however, his construction requires the number of transitions per state to be unbounded. It may be possible to remove this requirement, but we have not proved this.

So in order to experiment with trie minimization with don't cares, we have made some simplifications. We start by performing suffix compression in the usual manner. We then go through the states in a bottom-up order, checking each to see if it can be merged with any previous state by taking advantage of don't cares. Note that such merges may require further merges among states already seen.

We only try merges that actually save space, that is, where explicit transitions are merged. Otherwise, states with only a few transitions are very likely to be mergeable, but such merges may constrain us unnecessarily at a later stage of the minimization. In addition, we will not consider having multiple copies of states.

Even this simplified algorithm can be quite time consuming, so we did not try it on our pocket dictionary. On a list of 2726 technical words, don't care minimization reduced the number of states in the suffix-compressed, output-pruned trie from 1685 to just 283, while the number of transitions was reduced from 3627 to 2427. However, because the resulting states were larger, the first-fit packing performed rather poorly, producing a packed trie with 3408 transitions. So in this case don't care minimization yielded an additional compression of less than 10 percent.

Also, the behavior of the resulting hyphenation algorithm on words not in the dictionary became rather unpredictable. Once a word leaves the "known" paths of the packed trie, strange things might happen!

We can get even wilder effects by carrying the don't care assumption one step further, and eliminating the character field from the packed trie altogether (leaving just the output and trie link). Words in the dictionary will always index the correct transitions, but on other words we now have no way of telling when we have reached an invalid trie transition.

It turns out that the problem of state minimization with don't cares was studied in the 1960s by electrical engineers, who called it "minimization of incompletely specified sequential machines" (see e.g. [26]). However, typical instances of the problem involved machines with only a few states, rather than thousands as in our case, so it was often possible to find a minimized machine by hand. Also, the emphasis was on minimizing the number of *states* of the machine, rather than the number of state transitions.

In ordinary finite-state minimization, these are equivalent, but don't care minimization can actually introduce extra transitions, for example when states are duplicated. In the old days, finite-state machines were implemented using combinational logic, so the most important consideration was to reduce the number of states. In our trie representation, however, the space used is proportional to the number of transitions. Furthermore, finite-state machines are now often implemented using PLA's (programmed logic arrays), for which the number of transitions is also the best measure of space.

Pattern matching

Since trie minimization with don't cares still doesn't provide sufficient compression, and since it leads to unpredictable behavior on words not in the dictionary,

we need a different approach. It seems expensive to insist on complete hyphenation of the dictionary, so we will give up this requirement. We could allow some errors; or to be safer, we could allow some hyphens to be missed.

We now return to the pattern matching approach described in Chapter 1. Some further arguments as to why this method seems advantageous are given below. We should first reassure the reader that all the discussion so far has not been in vain, because a packed trie will be an ideal data structure for representing the patterns in the final hyphenation algorithm. Here the outputs will include the hyphenation level as well as the intercharacter position.

Hyphenating and inhibiting patterns allow considerable flexibility in the performance of the resulting algorithm. For example, we could allow a certain amount of error by using patterns that aren't always safe (but that presumably do find many correct hyphens).

We can also restrict ourselves to partial hyphenation in a natural way. That is, it turns out that a relatively small number of patterns will get a large fraction of the hyphens in the dictionary. The remaining hyphens become harder and harder to find, as we are left with mostly exceptional cases. Thus we can choose the most effective patterns first, taking more and more specialized patterns until we run out of space.

In addition, patterns perform quite well on words not in the dictionary, if those words follow "normal" pronunciation rules.

Patterns are "context-free"; that is, they can apply anywhere in a word. This seems to be an important advantage. In the trie-based approach discussed earlier in this chapter, a word is always scanned from beginning to end and each state of the trie 'remembers' the entire prefix of the word scanned so far, even if the letters scanned near the beginning no longer affect the hyphenation of the word. Suffix compression eliminates some of this unnecessary state information, by combining states that are identical with respect to future hyphenation. Minimization with don't cares takes this further, allowing 'similar' states to be combined as long as they behave identically on all characters that they have in common.

However, we have seen that it is difficult to guide the minimization with don't cares to achieve these reductions. Patterns embody such don't care situations naturally (if we can find a good way of selecting the patterns).

The context-free nature of patterns helps in another way, as explained below. Recall that we will use a packed trie to represent the patterns. To find all patterns that match in a given word, we perform a search starting at each letter of the word. Thus after completing a search starting from some letter position, we may have to

back up in the word to start the next search. By contrast, our original trie-based approach works with no backup.

Suppose we wanted to convert the pattern trie into a finite-state recognizer that works with no backup. This can be done in two stages. We first add "failure links" to each state that tell which state to go to if there is no explicit transition for the current character of the word. The failure state is the state in the trie that we would have reached, if we had started the search one letter later in the word.

Next, we can convert the failure-link machine into a true finite-state machine by filling in the missing transitions of each state with those of its failure state. (For more details of this process, see [27], [28].)

However, the above state merging will introduce a lot of additional transitions. Even using failure links requires one additional pointer per state. Thus by performing pattern matching with backup, we seem to save a good deal of space. And in practice, long backups rarely occur.

Finally, the idea of inhibiting patterns seems to be very useful. Such patterns extend the power of a finite-state machine, somewhat like adding the "not" operator to regular expressions.

Chapter 4

Pattern generation

We now discuss how to choose a suitable set of patterns for hyphenation. In order to decide which patterns are "good", we must first specify the desired properties of the resulting hyphenation algorithm.

We obviously want to maximize the number of hyphens found, minimize the error, and minimize the space required by our algorithm. For example, we could try to maximize some (say linear) function of the above three quantities, or we could hold one or two of the quantities constant and optimize the others.

For TEX82, we wanted a hyphenation algorithm meeting the following requirements. The algorithm should use only a moderate amount of space (20–30K bytes), including any exception dictionary; and it should find as many hyphens as possible, while making little or no error. This is similar to the specifications for the original TEX algorithm, except that we now hope to find substantially more hyphens.

Of course, the results will depend on the word list used. We decided to base the algorithm on our copy of Webster's Pocket Dictionary, mainly because this was the only word list we had that included all derived forms.

We also thought that a larger dictionary would contain many rare or specialized words that we might not want to worry about. In particular, we did not want such infrequent words to affect the choice of patterns, because we hoped to obtain a set of patterns embodying many of the "usual" rules for hyphenation.

In developing the TEX82 algorithm, however, the word list was tuned up considerably. A few thousand common words were weighted more heavily so that they would be more likely to be hyphenated. In fact, the current algorithm guarantees complete hyphenation of the 676 most common English words (according to [29]), as well as a short list of common technical words (e.g. `al-go-rithm`).

In addition, over 1000 "exception" words have been added to the dictionary, to ensure that they would not be incorrectly hyphenated. Most of these were found by testing the algorithm (based on the initial word list) against a larger dictionary obtained from a publisher, containing about 115,000 entries. This produced about

10,000 errors on words not in the pocket dictionary. Most of these were specialized technical terms that we decided not to worry about, but a few hundred were embarrassing enough that we decided to add them to the word list. These included compound words (`camp-fire`), proper names (`Af-ghan-i-stan`), and new words (`bio-rhythm`) that probably did not exist in 1966, when our pocket dictionary was originally put online.

After the word list was augmented, a new set of patterns was generated, and a new list of exceptions was found and added to the list. Fortunately this process seemed to converge after a few iterations.

Heuristics

The selection of patterns in an 'optimal' way seems very difficult. The problem is that several patterns may apply to a particular hyphen point, including both hyphenating and inhibiting patterns. Thus complicated interactions can arise if we try to determine, say, the minimum set of patterns finding a given number of hyphens. (The situation is somewhat analogous to a set cover problem.)

Instead, we will select patterns in a series of "passes" through the word list. In each pass we take into account only the effects of patterns chosen in previous passes. Thus we sidestep the problem of interactions mentioned above.

In addition, we will define a measure of pattern "efficiency" so that we can use a greedy approach in each pass, selecting the most efficient patterns.

Patterns will be selected one level at a time, starting with a level of hyphenating patterns. Patterns at each level will be selected in order of increasing pattern length.

Furthermore patterns of a given length applying to different intercharacter positions (for example `-tio` and `t-io`) will be selected in separate passes through the dictionary. Thus the patterns of length n at a given level will be chosen in $n+1$ passes through the dictionary.

At first we did not do this, but selected all patterns of a given length (at a given level) in a single pass, to save time. However, we found that this resulted in considerable duplication of effort, as many hyphens were covered by two or more patterns. By considering different intercharacter positions in separate passes, there is never any overlap among the patterns selected in a single pass.

In each pass, we collect statistics on all patterns appearing in the dictionary, counting the number of times we could hyphenate at a particular point in the pattern, and the number of times we could not.

For example, the pattern `tio` appears 1793 times in the pocket dictionary, and in 1773 cases we can hyphenate the word before the t, while in 20 cases we can

not. (We only count instances where the hyphen position occurs at least two letters from either edge of the word.)

These counts are used to determine the efficiency rating of patterns. For example if we are considering only "safe" patterns, that is, patterns that can always be hyphenated at a particular position, then a reasonable rating is simply the number of hyphens found. We could then decide to take, say, all patterns finding at least a given number of hyphens.

However, most of the patterns we use will make some error. How should these patterns be evaluated? In the worst case, errors can be handled by simply listing them in an exception dictionary. Assuming that one unit of space is required to represent each pattern as well as each exception, the "efficiency" of a pattern could be defined as $\mathit{eff} = \mathit{good}/(1 + \mathit{bad})$ where *good* is the number of hyphens correctly found and *bad* is the number of errors made.

(The space used by the final algorithm really depends on how much compression is produced by the packed trie used to represent the patterns, but since it is hard to predict the exact number of transitions required, we just use the number of patterns as an approximate measure of size.)

By using inhibiting patterns, however, we can often do better than listing the exceptions individually. The quantity *bad* in the above formula should then be devalued a bit depending on how effective patterns at the next level are. So a better formula might be

$$\mathit{eff} = \frac{\mathit{good}}{1 + \mathit{bad}/\mathit{bad\_eff}},$$

where *bad_eff* is the estimated efficiency of patterns at the next level (inhibiting errors at the current level).

Note that it may be difficult to determine the efficiency at the next level, when we are still deciding what patterns to take at the current level! We will use a pattern selection criterion of the form $\mathit{eff} \geq \mathit{thresh}$, but we cannot predict exactly how many patterns will be chosen and what their overall performance will be. The best we can do is use reasonable estimates based on previous runs of the pattern generation program. Some statistics from trial runs of this program are presented later in this chapter.

Collecting pattern statistics

So the main task of the pattern generation process is to collect count statistics about patterns in the dictionary. Because of time and space limitations this becomes an interesting data structure exercise.

For short (length 2 and 3) patterns, we can simply use a table of size 26^2 or 26^3, respectively, to hold the counts during a pass through the dictionary. For longer patterns, this is impractical.

Here's the first approach we used for longer patterns. In a pass through the dictionary, every occurrence of a pattern is written out to a file, along with an indication of whether or not a hyphen was allowed at the position under consideration. The file of patterns is sorted to bring identical patterns together, and then a pass is made through the sorted list to compile the count statistics for each pattern.

This approach makes it feasible to collect statistics for longer length patterns, and was used to conduct our initial experiments with pattern generation. However it is still quite time and space consuming, especially when sorting the large lists of patterns. Note that an external sorting algorithm is usually necessary.

Since only a fraction of the possible patterns of a particular length actually occur in the dictionary, we could instead store them in a hash table or one of the other data structures discussed in Chapter 2. It turns out that a modification of our packed trie data structure is well-suited to this task. The advantages of the packed trie are very fast lookup, compactness, and graceful handling of variable length patterns.

Combined with some judicious "pruning" of the patterns that are considered, the memory requirements are much reduced, allowing the entire pattern selection process to be carried out "in core" on our PDP-10 computer.

By "pruning" patterns we mean the following. If a pattern contains a shorter pattern at the same level that has already been chosen, the longer pattern obviously need not be considered, so we do not have to count its occurrences. Similarly, if a pattern appears so few times in the dictionary that under the current selection criterion it can never be chosen, then we can mark the pattern as "hopeless" so that any longer patterns at this level containing it need not be considered.

Pruning greatly reduces the number of patterns that must be considered, especially at longer lengths.

Dynamic packed tries

Unlike the static dictionary problem considered in Chapter 2, the set of patterns to be represented is not known in advance. In order to use a packed trie for storing the patterns being considered in a pass through the dictionary, we need some way to dynamically insert new patterns into the trie.

For any pattern, we start by performing a search in the packed trie as usual, following existing links until reaching a state where a new trie transition must be

added. If we are lucky, the location needed by the new transition will still be empty in the packed trie, otherwise we will have to do some repacking.

Note that we will not be using suffix compression, because this complicates things considerably. We would need back pointers or reference counts to determine what nodes need to be unmerged, and we would need a hash table or other auxiliary information in order to remerge the newly added nodes. Furthermore, suffix merging does not produce a great deal of compression on the relatively short patterns we will be dealing with.

The simplest way of resolving the packing conflict caused by the addition of a new transition is to just repack the changed state (and update the link of its parent state). To maintain good space utilization, we should try to fit the modified state among the holes in the trie. This can be done by maintaining a dynamic list of unoccupied cells in the trie, and using a first-fit search.

However, repacking turns out to be rather expensive for large states that are unlikely to fit into the holes in the trie, unless the array is very sparse. We can avoid this by packing such states into the free space immediately to the right of the occupied locations. The size threshold for attempting a first-fit packing can be adjusted depending on the density of the array, how much time we are willing to spend on insertions, or how close we are to running out of room.

After adding the critical transition as discussed above, we may need to add some more trie nodes for the remaining characters of the new pattern. These new states contain just a single transition, so they should be easy to fit into the trie.

The pattern generation program uses a second packed trie to store the set of patterns selected so far. Recall that, before collecting statistics about the patterns in each word, we must first hyphenate the word according to the patterns chosen in previous passes. This is done not only to determine the current partial hyphenation, but also to identify pruned patterns that need not be considered. Once again, the advantages of the packed trie are compactness and very fast "hyphenation".

At the end of a pass, we need to add new patterns, including "hopeless" patterns, to the trie. Thus it will be convenient to use a dynamic packed trie here as well. At the end of a level, we probably want to delete hopeless patterns from the trie in order to recover their space, if we are going to generate more levels. This turns out to be relatively easy; we just remove the appropriate output and return any freed nodes to the available list.

Below we give some statistics that will give an idea of how well a dynamic packed trie performs. We took the current set of 4447 hyphenation patterns, randomized them, and then inserted them one-by-one into a dynamic packed trie.

(Note that in the situations described above, there will actually be many searches per insertion, so we can afford some extra effort when performing insertions.) The patterns occupy 7214 trie nodes, but the packed trie will use more locations, depending on the setting of the first-fit packing threshold. The columns of the table show, respectively, the maximum state size for which a first-fit packing is attempted, the number of states packed, the number of locations tried by the first-fit procedure (this dominates the running time), the number of states repacked, and the number of locations used in the final packed trie.

thresh	pack	first_fit	unpack	trie_max
∞	6113	877,301	2781	9671
13	6060	761,228	2728	9458
9	6074	559,835	2742	9606
7	6027	359,537	2695	9606
5	5863	147,468	2531	10,366
4	5746	63,181	2414	11,209
3	5563	33,826	2231	13,296
2	5242	19,885	1910	15,009
1	4847	8956	1515	16,536
0	4577	6073	1245	18,628

Table 2. Dynamic packed trie statistics.

Experimental results

We now give some results from trial runs of the pattern generation program, and explain how the current TEX82 patterns were generated. As mentioned earlier, the development of these patterns involved some augmentation of the word list. The results described here are based on the latest version of the dictionary.

At each level, the selection of patterns is controlled by three parameters called *good_wt*, *bad_wt*, and *thresh*. If a pattern can be hyphenated *good* times at a particular position, but makes *bad* errors, then it will be selected if

$$good * good\_wt - bad * bad\_wt \geq thresh.$$

Note that the efficiency formula given earlier in this chapter can be converted into the above form.

We can first try using only safe patterns, that is, patterns that can always be hyphenated at a particular position. The table below shows the results when all safe patterns finding at least a given number of hyphens are chosen. Note that

parameters	patterns	hyphens	percent
$1 \infty 40$	401	31,083	35.2%
$1 \infty 20$	1024	45,310	51.3%
$1 \infty 10$	2272	58,580	66.3%
$1 \infty 5$	4603	70,014	79.2%
$1 \infty 3$	7052	76,236	86.2%
$1 \infty 2$	10,456	83,450	94.4%
$1 \infty 1$	16,336	87,271	98.7%

Table 9. Safe hyphenating patterns.

an infinite *bad_wt* ensures that only safe patterns are chosen. The table shows the number of patterns obtained, and the number and percentage of hyphens found.

We see that, roughly speaking, halving the threshold doubles the number of patterns, but only increases the percentage of hyphens by a constant amount. The last 20 percent or so of hyphens become quite expensive to find.

(In order to save computer time, we have only considered patterns of length 6 or less in obtaining the above statistics, so the figures do not quite represent all patterns above a given threshold. In particular, the patterns at threshold 1 do not find 100% of the hyphens, although even with indefinitely long patterns there would still be a few hyphens that would not be found, such as `re-cord`.)

The space required to represent patterns in the final algorithm is slightly more than one trie transition per pattern. Each transition occupies 4 bytes (1 byte each for character and output, plus 2 bytes for trie link). The output table requires an additional 3 bytes per entry (hyphenation position, value, and next output), but there are only a few hundred outputs. Thus to stay within the desired space limitations for TEX82, we can use at most about 5000 patterns.

We next try using two levels of patterns, to see if the idea of inhibiting patterns actually pays off. The results are shown below, where in each case the initial level of hyphenating patterns is followed by a level of inhibiting patterns that remove nearly all of the error.

The last set of patterns achieves 86.7% hyphenation using 4696 patterns. By contrast, the $1 \infty 3$ patterns from the previous table achieves 86.2% with 7052 patterns. So inhibiting patterns do help. In addition, notice that we have only used "safe" inhibiting patterns above; this means that none of the good hyphens are lost. We can do better by using patterns that also inhibit some correct hyphens.

After a good deal of further experimentation, we decided to use *five* levels of patterns in the current TEX82 algorithm. The reason for this is as follows. In

parameters	patterns	hyphens		percent	
1 20 20	816	51,359	505	58.1%	0.6%
1 ∞ 1	315	0	463	58.1%	0.1%
1 10 10	1510	64,893	1694	73.5%	1.9%
1 ∞ 1	824	0	1531	73.5%	0.2%
1 5 5	2573	76,632	5254	86.7%	5.9%
1 ∞ 1	2123	0	4826	86.7%	0.5%

Table 4. Two levels of patterns.

addition to finding a high percentage of hyphens, we also wanted a certain amount of guaranteed behavior. That is, we wanted to make essentially no errors on words in the dictionary, and also to ensure complete hyphenation of certain common words.

To accomplish this, we use a final level of safe hyphenating patterns, with the threshold set as low as feasible (in our case 4). If we then weight the list of important words by a factor of at least 4, the patterns obtained will hyphenate them completely (except when a word can be hyphenated in two different ways).

To guarantee no error, the level of inhibiting patterns immediately preceding the final level should have a threshold of 1 so that even patterns applying to a single word will be chosen. Note these do not need to be "safe" inhibiting patterns, since the final level will pick up all hyphens that should be found.

The problem is, if there are too many errors remaining before the last inhibiting level, we will need too many patterns to handle them. If we use three levels in all, then the initial level of hyphenating patterns can allow just a small amount of error.

However, we would like to take advantage of the high efficiency of hyphenating patterns that allow a greater percentage of error. So instead, we will use an initial level of hyphenating patterns with relatively high threshold and allowing considerable error, followed by a 'coarse' level of inhibiting patterns removing most of the initial error. The third level will consist of relatively safe hyphenating patterns with a somewhat lower threshold than the first level, and the last two levels will be as described above.

The above somewhat vague considerations do not specify the exact pattern selection parameters that should be used for each pass, especially the first three passes. These were only chosen after much trial and error, which would take too long to describe here. We do not have any theoretical justification for these parameters; they just seem to work well.

The table below shows the parameters used to generate the current set of TEX82 patterns, and the results obtained. For levels 2 and 4, the numbers in the "hyphens"

level	parameters	patterns	hyphens		percent	
1	1 2 20 (4)	458	67,604	14,156	76.6%	16.0%
2	2 1 8 (4)	509	7407	11,942	68.2%	2.5%
3	1 4 7 (5)	985	13,198	551	83.2%	3.1%
4	3 2 1 (6)	1647	1010	2730	82.0%	0.0%
5	1 ∞ 4 (8)	1320	6428	0	89.3%	0.0%

Table 5. Current TEX82 patterns.

column show the number of good and bad hyphens inhibited, respectively. The numbers in parentheses indicate the maximum length of patterns chosen at that level.

A total of 4919 patterns (actually only 4447 because some patterns appear more than once) were obtained, compiling into a suffix-compressed packed trie occupying 5943 locations, with 181 outputs. As shown in the table, the resulting algorithm finds 89.3% of the hyphens in the dictionary. This improves on the one and two level examples discussed above. The patterns were generated in 109 passes through the dictionary, requiring about 1 hour of CPU time.

Examples

The complete list of hyphenation patterns currently used by TEX82 appears in the appendix. The digits appearing between the letters of a pattern indicate the hyphenation level, as discussed above.

Below we give some examples of the patterns in action. For each of the following words, we show the patterns that apply, the resulting hyphenation values, and the hyphenation obtained. Note that if more than one hyphenation value is specified for a given intercharacter position, then the higher value takes priority, in accordance with our level scheme. If the final value is odd, the position is an allowable hyphen point.

```
computer 4m1p pu2t 5pute put3er co4m5pu2t3er com-put-er
algorithm l1g4 lgo3 1go 2ith 4hm al1g4o3r2it4hm al-go-rithm
hyphenation hy3ph he2n hena4 hen5at 1na n2at 1tio 2io
    hy3phe2n5a4t2ion hy-phen-ation
concatenation o2n on1c 1ca 1na n2at 1tio 2io
    co2n1cate1n2a1t2ion con-cate-na-tion
mathematics math3 ath5em th2e 1ma at1ic 4cs
    math5e1mat1i4cs math-e-mat-ics
```

```
typesetting type3 e1s2e 4t3t2 2t1in type3s2e4t3t2ing
    type-set-ting
program pr2 1gr pr2o1gram pro-gram
supercalifragilisticexpialidocious
    u1pe r1c 1ca al1i ag1i gil4 il1i il4ist is1ti st2i s1tic
    1exp x3p pi3a 2i1a i2al 2id 1do 1ci 2io 2us
    su1per1cal1ifrag1il4is1t2ic1ex3p2i3al2i1do1c2io2us
    su-per-cal-ifrag-ilis-tic-ex-pi-ali-do-cious
```

Below, we show a few interesting patterns. The reader may like to try figuring out what words they apply to. (The answers appear in the Appendix.)

```
ain5o     hach4     n3uin     5spai
ay5al     h5elo     nyp4      4tarc
ear5k     if4fr     o5a5les   4todo
e2mel     l5ogo     orew4     uir4m
```

And finally, the following patterns deserve mention:

```
3tex  fon4t  high5
```

Chapter 5

History and Conclusion

The invention of the alphabet was one of the greatest advances in the history of civilization. However, the ancient Phoenicians probably did not anticipate the fact that, centuries later, the problem of word hyphenation would become a major headache for computer typesetters all over the world.

Most cultures have evolved a linear style of communication, whereby a train of thought is converted into a sequence of symbols, which are then laid out in neat rows on a page and shipped off to a laser printer.

The trouble was, as civilization progressed and words got longer and longer, it became occasionally necessary to split them across lines. At first hyphens were inserted at arbitrary places, but in order to avoid distracting breaks such as `the-rapist`, it was soon found preferable to divide words at syllable boundaries.

Modern practice is somewhat stricter, avoiding hyphenations that might cause the reader to pronounce a word incorrectly (e.g. `considera-tion`) or where a single letter is split from a component of a compound word (e.g. `cardi-ovascular`).

The first book on typesetting, Joseph Moxon's *Mechanick Exercises* (1683), mentions the need for hyphenation but does not give any rules for it. A few dictionaries had appeared by this time, but were usually just word lists. Eventually they began to show syllable divisions to aid in pronunciation, as well as hyphenation.

With the advent of computer typesetting, interest in the problem was renewed. Hyphenation is the 'H' of 'H & J' (hyphenation and justification), which are the basic functions provided by any typesetting system. The need for automatic hyphenation presented a new and challenging problem to early systems designers.

Probably the first work on this problem, as well as many other aspects of computer typesetting, was done in the early 1950s by a French group led by G. D. Bafour. They developed a hyphenation algorithm for French, which was later adapted to English [U.S. Patent 2,762,485 (1955)].

Their method is quite simple. Hyphenations are allowed anywhere in a word except among the following letter combinations: before two consonants, two vowels,

or x; between two vowels, consonant-h, e-r, or s-s; after tw
first is not l, m, n, r, or s; or after c, j, q, v, consonant-w,
nn, or nr.

We tested this method on our pocket dictionary, and it
of the hyphens, but also about an equal amount of incorr
another way, about 65% of the erroneous hyphen positions a
along with 30% of the correct hyphens. It turns out that
this one works quite well in French; however for English th

Other early work on automatic hyphenation is describ
various conferences on computer typesetting (e.g. [30]). A
in [31], from which the quotes in the following paragraphs

At the Los Angeles Times, a sophisticated logical routi
on the grammatical rules given in Webster's, carefully refin
puter implementation. Words were analyzed into vowel
which were classified into one of four types, and rules gove
Prefix, suffix, and other special case rules were also used.
edly "85–95 percent accurate", while the hyphenation log
positions of the 20,000 positions of the computer's magn
space than would be required to store 500 8-letter words av
word."

Perry Publications in Florida developed a dictionary
with their own dictionary. An in-core table mapped each
first two letters, into a particular block of words on tape. F
was divided between four tape units, and "since the RCA
both directions," each tape drive maintained a "homing posi
the tape, with the most frequently searched blocks placed c
positions.

In addition, they observed that many words could be hyp
5th, or 7th letters. So they removed all such words from the
space), and if a word was not found in the dictionary, it w
3rd, 5th, or 7th letter.

A hybrid approach was developed at the Oklahoma Pul
some logical analysis was used to determine the number of
if certain suffix and special case rules could be applied.
hyphenation at each position in the word was estimated
tables, and the most probable breakpoints were identified.
origin of the Time magazine algorithm described in Cha

dictionary handles the remaining cases; however there was some difference of opinion as to the size of the dictionary required to obtain satisfactory results.

Many other projects to develop hyphenation algorithms have remained proprietary or were never published. For example, IBM alone worked on "over 35 approaches to the simple problem of grammatical word division and hyphenation".

By now, we might have hoped that an "industry standard" hyphenation algorithm would exist. Indeed Berg's survey of computerized typesetting [32] contains a description of what could be considered a "generic" rule-based hyphenation algorithm (he doesn't say where it comes from). However, we have seen that any logical routine must stop short of complete hyphenation, because of the generally illogical basis of English word division.

The trend in modern systems has been toward the hybrid approach, where a logical routine is supplemented by an extensive exception dictionary. Thus the in-core algorithm serves to reduce the size of the dictionary, as well as the frequency of accessing it, as much as possible.

A number of hyphenation algorithms have also appeared in the computer science literature. A very simple algorithm is described by Rich and Stone [33]. The two parts of the word must include a vowel, not counting a final e, es or ed. The new line cannot begin with a vowel or double consonant. No break is made between the letter pairs sh, gh, p, ch, th, wh, gr, pr, cr, tr, wr, br, fr, dr, vowel-r, vowel-n, or om. On our pocket dictionary, this method found about 70% of the hyphens with 45% error.

The algorithm used in the Bell Labs document compiler Roff is described by Wagner [34]. It uses suffix stripping, followed by digram analysis carried out in a back to front manner. In addition a more complicated scheme is described using four classes of digrams combined with an attempt to identify accented and nonaccented syllables, but this seemed to introduce too many errors. A version of the algorithm is described in [35]; interestingly, this reference uses the terms "hyphenating pattern" (referring to a Snobol string-matching pattern) as well as "inhibiting suffix".

Ocker [36], in a master's thesis, describes another algorithm based on the rules in Webster's dictionary. It includes recognition of prefixes, suffixes, and special letter combinations that help in determining accentuation, followed by an analysis of the "liquidity" of letter pairs to find the character pair corresponding to the greatest interruption of spoken sound.

Moitra et al [37] use an exception table, prefixes, suffixes, and a probabilistic break-value table. In addition they extend the usual notion of affixes to any letter

pattern that helps in hyphenation, including 'root words' (e.g. `line`, `pot`) intended to handle compound words.

Patterns as paradigm

Our pattern matching approach to hyphenation is interesting for a number of reasons. It has proved to be very effective and also very appropriate for the problem. In addition, since the patterns are generated from the dictionary, it is easy to accommodate changes to the word list, as our hyphenation preferences change or as new words are added. More significantly, the pattern scheme can be readily applied to different languages, if we have a hyphenated word list for the language.

The effectiveness of pattern matching suggests that this paradigm may be useful in other applications as well. Indeed more general pattern matching systems and the related notions of production systems and augmented transition networks (ATN's) are often used in artificial intelligence applications, especially natural language processing. While AI programs try to understand sentences by analyzing word patterns, we try to hyphenate words by analyzing letter patterns.

One simple extension of patterns that we have not considered is the idea of character groups such as vowels and consonants, as used by nearly all other algorithmic approaches to hyphenation. This may seem like a serious omission, because a potentially useful meta-pattern like 'vowel-consonant-consonant-vowel' would then expand to $6 \times 20 \times 20 \times 6 = 14400$ patterns. However, it turns out that a suffix-compressed trie will reduce this to just $6 + 20 + 20 + 6 = 52$ trie nodes. So our methods can take some advantage of such "meta-patterns".

In addition, the use of inhibiting as well as hyphenating patterns seems quite powerful. These can be thought of as rules and exceptions, which is another common AI paradigm.

Concerning related work in AI, we must especially mention the Meta-DENDRAL program [38], which is designed to infer automatically rules for mass-spectrometry. An example of such a rule is N—C—C—C $\rightarrow$ N—C * C—C, which says that if the molecular substructure on the left side is present, then a bond fragmentation may occur as indicated on the right side. Meta-DENDRAL analyzes a set of mass-spectral data points and tries to infer a set of fragmentation rules that can correctly predict the spectra of new molecules. The inference process starts with some fairly general rules and then refines them as necessary, using the experimental data as positive or negative evidence for the correctness of a rule.

The fragmentation rules can in general be considerably more complicated than our simple pattern rules for hyphenation. The molecular "pattern" can be a tree-like or even cyclic structure, and there may be multiple fragmentations, possibly involving "migration" of a few atoms from one fragment to another. Furthermore, there are usually extra constraints on the form of rules, both to constrain the search and to make it more likely that meaningful or "interesting" rules will be generated. Still, there are some striking similarities between these ideas and our pattern-matching approach to hyphenation.

Packed tries

Finally, the idea of packed tries deserves further investigation. An indexed trie can be viewed as a finite-state machine, where state transitions are performed by address calculation based on the current state and input character. This is extremely fast on most computers.

However indexing usually incurs a substantial space penalty because of space reserved for pointers that are not used. Our packing technique, using the idea of storing the index character to distinguish transitions belonging to different states, combines the best features of both the linked and indexed representations, namely space and speed. We believe this is a fundamental idea.

There are various issues to be explored here. Some analysis of different packing methods would be interesting, especially for the handling of dynamic updates to a packed trie.

Our hyphenation trie extends a finite-state machine with its hyphenation "actions". It would be interesting to consider other applications that can be handled by extending the basic finite-state framework, while maintaining as much of its speed as possible.

Another possibly interesting question concerns the size of the character and pointer fields in trie transitions. In our hyphenation trie half of the space is occupied by the pointers, while in our spelling checking examples from one-half to three-fourths of the space is used for pointers, depending on the size of the dictionary. In the latter case it might be better to use a larger "character" size in the trie, in order to get a better balance between pointers and data.

When performing a search in a packed trie, following links will likely make us jump around in the trie in a somewhat random manner. This can be a disadvantage, both because of the need for large pointers, and also because of the lack of locality, which could degrade performance in a virtual memory environment. There are probably ways to improve on this. For example, Fredkin [10] proposes an interesting 'n-dimensional binary trie' idea for reducing pointer size.

We have presented packed tries as a solution to the set representation problem, with special emphasis on data compression. It would be interesting to compare our results with other compression techniques, such as Huffman coding. Also, perhaps one could estimate the amount of information present in a hyphenated word list, as a lower bound on the size of any hyphenation algorithm.

Finally, our view of finite-state machines has been based on the underlying assumption of a computer with random-access memory. Addressing by indexing seems to provide power not available in some other models of computation, such as pointer machines or comparison-based models. On the other hand, a 'VLSI' or other hardware model (such as programmed logic arrays) can provide even greater power, eliminating the need for our perhaps contrived packing technique. But then other communication issues will be raised.

If all problems of hyphenation have not been solved,
at least some progress has been made since that night,
when according to legend, an RCA Marketing Manager
received a phone call from a disturbed customer.
His 301 had just hyphenated "God".

— Paul E. Justus (1972)

PATtern GENeration program for the TEX82 hyphenator

1. Introduction. This program takes a list of hyphenated words and generates a set of patterns that can be used by the TEX82 hyphenation algorithm.

The patterns consist of strings of letters and digits, where a digit indicates a 'hyphenation value' for some intercharacter position. For example, the pattern `3t2ion` specifies that if the string `tion` occurs in a word, we should assign a hyphenation value of 3 to the position immediately before the `t`, and a value of 2 to the position between the `t` and the `i`.

To hyphenate a word, we find all patterns that match within the word and determine the hyphenation values for each intercharacter position. If more than one pattern applies to a given position, we take the maximum of the values specified (i.e. the higher value takes priority). If the resulting hyphenation value is odd, this position is a feasible breakpoint; if the value is even or if no value has been specified, we are not allowed to break at this position.

In order to find quickly the patterns that match in a given word and to compute the associated hyphenation values, the patterns generated by this program are compiled by INITEX into a compact version of a finite state machine. For further details, see the TEX82 source.

```
⟨ Compiler directives 10 ⟩
program PATGEN(dictionary, patterns, output);
  label ⟨ Labels in the outer block 7 ⟩
  const ⟨ Constants in the outer block 18 ⟩
  type ⟨ Types in the outer block 11 ⟩
  var ⟨ Globals in the outer block 2 ⟩
  procedure initialize; { this procedure gets things started properly }
    var ⟨ Local variables for initialization 13 ⟩
    begin ⟨ Set up input/output translation tables 14 ⟩
    end;
```

2. The patterns are generated in a series of sequential passes through the dictionary. In each pass, we collect count statistics for a particular type of pattern, taking into account the effect of patterns chosen in previous passes. At the end of a pass, the counts are examined and new patterns are selected.

Patterns are chosen one level at a time, in order of increasing hyphenation value. In the sample run shown below, the parameters *hyph_start* and *hyph_finish* specify the first and last levels, respectively, to be generated.

Patterns at each level are chosen in order of increasing pattern length (usually starting with length 2). This is controlled by the parameters *pat_start* and *pat_finish* specified at the beginning of each level.

Furthermore patterns of the same length applying to different intercharacter positions are chosen in separate passes through the dictionary. Since patterns of length n may apply to $n+1$ different positions, choosing a set of patterns of lengths 2 through n for a given level requires $(n+1)(n+2)/2-3$ passes through the word list.

At each level, the selection of patterns is controlled by the three parameters *good_wt*, *bad_wt*, and *thresh*. A hyphenating pattern will be selected if $good * good\_wt - bad * bad\_wt \geq thresh$, where *good* and *bad* are the number of times the pattern could and could not be hyphenated, respectively, at a particular point. For inhibiting patterns, *good* is the number of errors inhibited, and *bad* is the number of previously found hyphens inhibited.

```
⟨ Globals in the outer block 2 ⟩ ≡
pat_start, pat_finish: dot_type;
hyph_start, hyph_finish: val_type;
good_wt, bad_wt, thresh: integer;
```

See also sections 4, 12, 20, 21, 23, 30, 33, 43, 51, 59, 61, and 70.

This code is used in section 1.

3. The proper choice of the parameters to achieve a desired degree of hyphenation is discussed in Chapter 4. Below we show part of a sample run of PATGEN, with the user's inputs underlined.

```
ex patgen
DICTIONARY : murray.hyf
PATTERNS : nul:
OUTPUT : murray.pat
0 patterns read in
pattern trie has 128 nodes, trie_max = 128, 0 outputs
hyph_start = 1
hyph_finish = 1
pat_start = 2
pat_finish = 3
good weight, bad weight, threshold:  1 3 3
processing dictionary with pat_len = 2, pat_dot = 1

0 good, 0 bad, 3265 missed
0.00 %, 0.00 %, 100.00 %
338 patterns, 466 nodes in count trie, triec_max = 983
46 good and 152 bad patterns added (more to come)
finding 715 good and 62 bad hyphens, efficiency = 10.72
pattern trie has 326 nodes, trie_max = 509, 2 outputs
processing dictionary with pat_len = 2, pat_dot = 0

   ...

1592 nodes and 39 outputs deleted
total of 220 patterns at hyph_level 1
hyphenate word list?  y
writing pattmp.1

2529 good, 243 bad, 736 missed
77.46 %, 7.44 %, 22.54 %
```

4. Note that before beginning a pattern selection run, a file of existing patterns may be read in. In order for pattern selection to work properly, this file should only contain patterns with hyphenation values less than *hyph_start*. Each word in the dictionary is hyphenated according to the existing set of patterns (including those chosen on previous passes of the current run) before pattern statistics are collected.

Also, a hyphenated word list may be written out at the end of a run. This list can be read back in as the 'dictionary' to continue pattern selection from this point. The new list will contain two additional kinds of "hyphens" between letters, namely hyphens that have been found by previously generated patterns, as well as erroneous hyphens that have been inserted by those patterns. These are represented by the symbols "*" and ".", respectively.

In addition, a word list can include hyphen weights, both for entire words and for individual hyphen positions. (The syntax for this is explained in the dictionary processing routines.) Thus common words can be weighted more heavily, or, more generally, words can be weighted according to their frequency of occurrence, if such information is available. The use of hyphen weights combined with an appropriate setting of the pattern selection threshold can be used to guarantee hyphenation of certain words or certain hyphen positions within a word.

⟨ Globals in the outer block 2 ⟩ +≡
dictionary, *patterns*, *pattmp*: **file of** *text_char*;

5. Below we show the first few lines of a typical word list, before and after generating a level of patterns.

```
abil-i-ty              abil*i*ty
ab-sence               ab*sence
ab-stract              ab*stract
ac-a-dem-ic            ac-a-d.em-ic
ac-cept                ac*cept
ac-cept-able           ac*cept-able
ac-cept-ed             ac*cept*ed
   ...                    ...
```

6. We augment PASCAL's control structures a bit using **goto**'s and the following symbolic labels.

```
define exit = 10 { go here to leave a procedure }
define continue = 22 { go here to resume a loop }
define done = 30 { go here to exit a loop }
define found = 40 { go here when you've found it }
define not_found = 41 { go here when you've found something else }
define end_of_PATGEN = 9999
```

7. ⟨ Labels in the outer block 7 ⟩ ≡

```
continue, end_of_PATGEN;
```

This code is used in section 1.

8. Here are some macros for common programming idioms.

```
define incr(#) ≡ # ← # + 1 { increase a variable by unity }
define decr(#) ≡ # ← # − 1 { decrease a variable by unity }
define return ≡ goto exit { terminate a procedure call }
format return ≡ nil
```

9. The following definitions are used for terminal input/output.

```
define print(#) ≡ write(tty,#)
define print_ln(#) ≡ writeln(tty,#)
define input(#) ≡ read(tty,#)
define input_ln(#) ≡
        begin if eoln(tty) then readln(tty);
        read(tty,#)
        end
define error(#) ≡
        begin print_ln(#); goto end_of_PATGEN;
        end;
```

10. ⟨ Compiler directives 10 ⟩ ≡

```
@{@&$C−,A+,D−@}  { no range check, catch arithmetic overflow, no debug overhead }
```

This code is used in section 1.

11. The character set. Since different PASCAL systems may use different character sets, we use the name *text_char* to stand for the data type of characters appearing in external text files. We also assume that *text_char* consists of the elements *chr*(*first_text_char*) through *chr*(*last_text_char*), inclusive. The definitions below should be adjusted if necessary.

Internally, characters will be represented using the type *ascii_code*.

```
define first_text_char = 0   { ordinal number of the smallest element of text_char }
define last_text_char = 127   { ordinal number of the largest element of text_char }
⟨ Types in the outer block 11 ⟩ ≡
  text_char = char;   { the data type of characters in text files }
  ascii_code = 0 .. 127;   { internal representation of input characters }
```

See also section 19.

This code is used in section 1.

12. We convert between *ascii_code* and the user's external character set by means of arrays *xord* and *xchr* that are analogous to PASCAL's *ord* and *chr* functions.

```
⟨ Globals in the outer block 2 ⟩ +≡
xord: array [text_char] of ascii_code;   { specifies conversion of input characters }
xchr: array [ascii_code] of text_char;   { specifies conversion of output characters }
```

13.
```
⟨ Local variables for initialization 13 ⟩ ≡
i: text_char;
j: ascii_code;
```

This code is used in section 1.

14. For English, our dictionary and pattern files use the letters A through Z, the digits 0 through 9, and the symbols *, -, and . (period). Lowercase letters are also accepted; they are mapped internally to upper case.

```
define invalid_code = '177   { ascii code that should not appear }
⟨ Set up input/output translation tables 14 ⟩ ≡
  for i ← chr(first_text_char) to chr(last_text_char) do xord[i] ← invalid_code;
  xord['*'] ← "*"; xord['-'] ← "-"; xord['.'] ← ".";
  xord['0'] ← "0"; xord['1'] ← "1"; xord['2'] ← "2"; xord['3'] ← "3"; xord['4'] ← "4";
  xord['5'] ← "5"; xord['6'] ← "6"; xord['7'] ← "7"; xord['8'] ← "8"; xord['9'] ← "9";
  xord['A'] ← "A"; xord['B'] ← "B"; xord['C'] ← "C"; xord['D'] ← "D"; xord['E'] ← "E";
  xord['F'] ← "F"; xord['G'] ← "G"; xord['H'] ← "H"; xord['I'] ← "I"; xord['J'] ← "J";
  xord['K'] ← "K"; xord['L'] ← "L"; xord['M'] ← "M"; xord['N'] ← "N"; xord['O'] ← "O";
  xord['P'] ← "P"; xord['Q'] ← "Q"; xord['R'] ← "R"; xord['S'] ← "S"; xord['T'] ← "T";
  xord['U'] ← "U"; xord['V'] ← "V"; xord['W'] ← "W"; xord['X'] ← "X"; xord['Y'] ← "Y";
  xord['Z'] ← "Z";
  xord['a'] ← "A"; xord['b'] ← "B"; xord['c'] ← "C"; xord['d'] ← "D"; xord['e'] ← "E";
  xord['f'] ← "F"; xord['g'] ← "G"; xord['h'] ← "H"; xord['i'] ← "I"; xord['j'] ← "J";
  xord['k'] ← "K"; xord['l'] ← "L"; xord['m'] ← "M"; xord['n'] ← "N"; xord['o'] ← "O";
  xord['p'] ← "P"; xord['q'] ← "Q"; xord['r'] ← "R"; xord['s'] ← "S"; xord['t'] ← "T";
  xord['u'] ← "U"; xord['v'] ← "V"; xord['w'] ← "W"; xord['x'] ← "X"; xord['y'] ← "Y";
  xord['z'] ← "Z";
```

See also section 15.

This code is used in section 1.

15. On output, all letters are mapped to lower case.

```
⟨ Set up input/output translation tables 14 ⟩ +≡
  for j ← 0 to 127 do xchr[j] ← ' ';
  xchr["*"] ← '*'; xchr["-"] ← '-'; xchr["."] ← '.';
  xchr["0"] ← '0'; xchr["1"] ← '1'; xchr["2"] ← '2'; xchr["3"] ← '3'; xchr["4"] ← '4';
  xchr["5"] ← '5'; xchr["6"] ← '6'; xchr["7"] ← '7'; xchr["8"] ← '8'; xchr["9"] ← '9';
  xchr["A"] ← 'a'; xchr["B"] ← 'b'; xchr["C"] ← 'c'; xchr["D"] ← 'd'; xchr["E"] ← 'e';
  xchr["F"] ← 'f'; xchr["G"] ← 'g'; xchr["H"] ← 'h'; xchr["I"] ← 'i'; xchr["J"] ← 'j';
  xchr["K"] ← 'k'; xchr["L"] ← 'l'; xchr["M"] ← 'm'; xchr["N"] ← 'n'; xchr["O"] ← 'o';
  xchr["P"] ← 'p'; xchr["Q"] ← 'q'; xchr["R"] ← 'r'; xchr["S"] ← 's'; xchr["T"] ← 't';
  xchr["U"] ← 'u'; xchr["V"] ← 'v'; xchr["W"] ← 'w'; xchr["X"] ← 'x'; xchr["Y"] ← 'y';
  xchr["Z"] ← 'z';
```

16. We assume that words use only the letters *cmin* + 1 through *cmax*. This allows us to save some time on trie operations that involve searching for packed transitions belonging to a particular state.

```
define cmin = "@"
define cmax = "Z"
define edge_of_word = "@"
```

17. Data structures. The main data structure used in this program is a dynamic packed trie. In fact we use two of them, one for the set of patterns selected so far, and one for the patterns being considered in the current pass.

For a pattern $p_1 \ldots p_k$, the information associated with that pattern is accessed by setting $t_1 \leftarrow trie\_root + p_1$ and then, for $1 < i \leq k$, setting $t_i \leftarrow trie\_link(t_{i-1}) + p_i$; the pattern information is then stored in a location addressed by t_k. Since all trie nodes are packed into a single array, in order to distinguish nodes belonging to different trie families, a special field is provided such that $trie\_char(t_i) = p_i$ for all i.

In addition the trie must support dynamic insertions and deletions. This is done by maintaining a doubly linked list of unoccupied cells and repacking trie families as necessary when insertions are made.

Each trie node consists of three fields: the character *trie_char*, and the two link fields *trie_link* and *trie_back*. In addition there is a separate boolean array *trie_base_used*. When a node is unoccupied, *trie_char* is zero and the link fields point to the next and previous unoccupied nodes, respectively, in the doubly linked list. When a node is occupied, *trie_link* points to the next trie family, and *trie_back* (renamed *trie_outp*) contains the output associated with this transition. The *trie_base_used* bit indicates that some family has been packed at this base location, and is used to prevent two families from being packed at the same location.

18. The sizes of the pattern tries may have to be adjusted depending on the particular application (i.e. the parameter settings and the size of the dictionary). The sizes below were sufficient to generate the current set of TEX82 hyphenation patterns.

```
⟨ Constants in the outer block 18 ⟩ ≡
  trie_size = 55000; { space for pattern trie }
  triec_size = 26000; { space for pattern count trie, should be less than trie_size and greater than the
      number of occurrences of any pattern in the dictionary }
  max_ops = 4080; { size of output hash table, should be a multiple of 510 }
  max_val = 7; { maximum number of levels+1, also used to denote bad patterns }
  max_dot = 15; { maximum pattern length }
  max_len = 50; { maximum word length }
```

This code is used in section 1.

19.

```
⟨ Types in the outer block 11 ⟩ +≡
  q_index = 0 .. 128; { number of transitions in a state }
  val_type = 0 .. max_val; { hyphenation values }
  dot_type = 0 .. max_dot; { dot positions }
  op_type = 0 .. max_ops; { index into output hash table }
  word_index = 0 .. max_len; { index into word }
  trie_pointer = 0 .. trie_size; triec_pointer = 0 .. triec_size;
  trie_node = record ch: ascii_code;
    lh, rh: trie_pointer
    end;
  op_word = packed record dot: dot_type;
    val: val_type;
    op: op_type
    end;
```

20. Instead of using the *trie_node* record defined above, the trie is actually stored with its components in separate packed arrays, in order to save space (although this depends on the computer's word size and the size of the trie pointers).

⟨ Globals in the outer block 2 ⟩ +≡
trie_c: **packed array** [*trie_pointer*] **of** *ascii_code*;
trie_l, *trie_r*: **packed array** [*trie_pointer*] **of** *trie_pointer*;
trie_taken: **packed array** [*trie_pointer*] **of** *boolean*;
triec_c: **packed array** [*triec_pointer*] **of** *ascii_code*;
triec_l, *triec_r*: **packed array** [*triec_pointer*] **of** *triec_pointer*;
triec_taken: **packed array** [*triec_pointer*] **of** *boolean*;
ops: **array** [*op_type*] **of** *op_word*; { output hash table }

21. When some trie state is being worked on, an unpacked version of the state is kept in positions 1 .. *qmax* of the global array *trieq*. The character fields need not be in any particular order.

⟨ Globals in the outer block 2 ⟩ +≡
trieq: **array** [1 .. 128] **of** *trie_node*; { holds a single trie state }
qmax: *q_index*;
qmax_thresh: *q_index*; { controls density of first-fit packing }

22. Trie fields are accessed using the following macros.

define *trie_char*(#) ≡ *trie_c*[#]
define *trie_link*(#) ≡ *trie_l*[#]
define *trie_back*(#) ≡ *trie_r*[#]
define *trie_outp*(#) ≡ *trie_r*[#]
define *trie_base_used*(#) ≡ *trie_taken*[#]
define *triec_char*(#) ≡ *triec_c*[#]
define *triec_link*(#) ≡ *triec_l*[#]
define *triec_back*(#) ≡ *triec_r*[#]
define *triec_good*(#) ≡ *triec_l*[#]
define *triec_bad*(#) ≡ *triec_r*[#]
define *triec_base_used*(#) ≡ *triec_taken*[#]
define *q_char*(#) ≡ *trieq*[#].*ch*
define *q_link*(#) ≡ *trieq*[#].*rh*
define *q_back*(#) ≡ *trieq*[#].*lh*
define *q_outp*(#) ≡ *trieq*[#].*lh*
define *hyf_val*(#) ≡ *ops*[#].*val*
define *hyf_dot*(#) ≡ *ops*[#].*dot*
define *hyf_nxt*(#) ≡ *ops*[#].*op*

23. Routines for pattern trie. The pattern trie holds the set of patterns chosen prior to the current pass, including bad or "hopeless" patterns at the current level that occur too few times in the dictionary to be of use. Each transition of the trie includes an output field pointing to the hyphenation information associated with this transition.

```
⟨ Globals in the outer block 2 ⟩ +≡
trie_max: trie_pointer; { maximum occupied trie node }
trie_bmax: trie_pointer; { maximum base of trie family }
trie_count: trie_pointer; { number of occupied trie nodes, for space usage statistics }
op_count: op_type; { number of outputs in hash table }
```

24. Initially, the dynamic packed trie has just one state, namely the root, with all transitions present (but with null links). This is convenient because the root will never need to be repacked and also we won't have to check that the base is nonnegative when packing other states.

```
define trie_root = 1
procedure init_pattern_trie;
  var c: ascii_code; h: op_type;
  begin for c ← 0 to 127 do
    begin trie_char(trie_root + c) ← c; { indicates node occupied }
    trie_link(trie_root + c) ← 0; trie_outp(trie_root + c) ← 0;
    end;
  trie_base_used(trie_root) ← true; trie_bmax ← trie_root; trie_max ← trie_root + 127; trie_count ← 128;
  qmax_thresh ← 5;
  trie_link(0) ← trie_max + 1; trie_back(trie_max + 1) ← 0;
    { trie_link(0) is used as the head of the doubly linked list of unoccupied cells }
  for h ← 1 to max_ops do hyf_val(h) ← 0; { clear output hash table }
  op_count ← 0;
  end;
```

25. The *first_fit* procedure finds a hole in the packed trie into which the state in *trieq* will fit. This is normally done by going through the linked list of unoccupied cells and testing if the state will fit at each position. However if a state has too many transitions (and is therefore unlikely to fit among existing transitions) we don't bother and instead just pack it immediately to the right of the occupied region (starting at *trie_max* + 1).

```
function first_fit: trie_pointer;
  label found, not_found;
  var s,t: trie_pointer; q: q_index;
  begin ⟨ Set s to the trie base location at which this state should be packed 26 ⟩;
  for q ← 1 to qmax do { pack it }
    begin t ← s + q_char(q);
    trie_link(trie_back(t)) ← trie_link(t); trie_back(trie_link(t)) ← trie_back(t); { link around filled cell }
    trie_char(t) ← q_char(q); trie_link(t) ← q_link(q); trie_outp(t) ← q_outp(q);
    if t > trie_max then trie_max ← t;
    end;
  trie_base_used(s) ← true; first_fit ← s
  end;
```

26. The threshold for large states is initially 5 transitions. If more than one level of patterns is being generated, the threshold is set to 7 on subsequent levels because the pattern trie will be sparser after bad patterns are deleted (see *delete_bad_patterns*).

```
⟨ Set s to the trie base location at which this state should be packed 26 ⟩ ≡
  if qmax > qmax_thresh then t ← trie_back(trie_max + 1) else t ← 0;
  while true do
    begin t ← trie_link(t); s ← t − q_char(1); { get next unoccupied cell }
    ⟨ Ensure trie linked up to s + 128 27 ⟩;
    if trie_base_used(s) then goto not_found;
    for q ← qmax downto 2 do { check if state fits here }
      if trie_char(s + q_char(q)) > 0 then goto not_found;
    goto found;
  not_found: end;
found:
```

This code is used in section 25.

27. The trie is only initialized (as a doubly linked list of empty cells) as far as necessary. Here we extend the initialization if necessary, and check for overflow.

```
⟨ Ensure trie linked up to s + 128 27 ⟩ ≡
  if s + 128 > trie_size then error('Pattern␣trie␣too␣full!');
  while trie_bmax < s do
    begin incr(trie_bmax); trie_base_used(trie_bmax) ← false; trie_char(trie_bmax + 127) ← 0;
    trie_link(trie_bmax + 127) ← trie_bmax + 128; trie_back(trie_bmax + 128) ← trie_bmax + 127;
    end
```

This code is used in section 26.

28. The *unpack* procedure finds all transitions associated with the state with base *s*, puts them into the array *trieq*, and sets *qmax* to the number of transitions found. Freed cells are put at the beginning of the free list.

```
procedure unpack(s : trie_pointer);
  var c: ascii_code; t: trie_pointer;
  begin qmax ← 0;
  for c ← cmin to cmax do { search for transitions belonging to this state }
    begin t ← s + c;
    if trie_char(t) = c then { found one }
      begin incr(qmax);
      q_char(qmax) ← c; q_link(qmax) ← trie_link(t); q_outp(qmax) ← trie_outp(t);
        { now free trie node }
      trie_back(trie_link(0)) ← t; trie_link(t) ← trie_link(0); trie_link(0) ← t; trie_back(t) ← 0;
      trie_char(t) ← 0;
      end;
    end;
  trie_base_used(s) ← false;
  end;
```

29. The function *new_trie_op* returns the 'opcode' for the output consisting of hyphenation value v, hyphen position d, and next output n. The hash function used by *new_trie_op* is based on the idea that 313/510 is an approximation to the golden ratio [cf. *The Art of Computer Programming* **3** (1973), 510–512]; but the choice is comparatively unimportant in this particular application.

```
function new_trie_op(v : val_type; d : dot_type; n : op_type): op_type;
  label exit;
  var h: op_type;
  begin h ← ((n + 313 * d + 361 * v) mod max_ops) + 1;  { trial hash location }
  while true do
    begin if hyf_val(h) = 0 then  { empty position found }
      begin incr(op_count);
      if op_count = max_ops then error('Too␣many␣outputs!');
      hyf_val(h) ← v; hyf_dot(h) ← d; hyf_nxt(h) ← n; new_trie_op ← h; return;
      end;
    if (hyf_val(h) = v) ∧ (hyf_dot(h) = d) ∧ (hyf_nxt(h) = n) then  { already in hash table }
      begin new_trie_op ← h; return;
      end;
    if h > 1 then decr(h) else h ← max_ops;  { try again }
    end;
exit: end;
```

30. ⟨ Globals in the outer block 2 ⟩ +≡

```
pat: array [dot_type] of ascii_code;  { current pattern }
pat_len: dot_type;  { pattern length }
```

31. Now that we have provided the necessary routines for manipulating the dynamic packed trie, here is a procedure that inserts a pattern of length *pat_len*, stored in the *pat* array, into the pattern trie. It also adds a new output.

```
procedure insert_pattern(val : val_type; dot : dot_type);
  var i: dot_type; s,t: trie_pointer;
  begin i ← 1; s ← trie_root + pat[i]; t ← trie_link(s);
  while (t > 0) ∧ (i < pat_len) do  { follow existing trie }
    begin incr(i); t ← t + pat[i];
    if trie_char(t) ≠ pat[i] then ⟨ Insert critical transition, possibly repacking 32 ⟩;
    s ← t; t ← trie_link(s);
    end;
  q_link(1) ← 0; q_outp(1) ← 0; qmax ← 1;
  while i < pat_len do  { insert rest of pattern }
    begin incr(i); q_char(1) ← pat[i]; t ← first_fit; trie_link(s) ← t; s ← t + pat[i]; incr(trie_count);
    end;
  trie_outp(s) ← new_trie_op(val, dot, trie_outp(s));
  end;
```

32. We have accessed a transition not in the trie. We insert it, repacking the state if necessary.

```
⟨ Insert critical transition, possibly repacking 32 ⟩ ≡
  begin if trie_char(t) = 0 then
    begin   { we're lucky, no repacking needed }
    trie_link(trie_back(t)) ← trie_link(t); trie_back(trie_link(t)) ← trie_back(t);
    trie_char(t) ← pat[i]; trie_link(t) ← 0; trie_outp(t) ← 0;
    if t > trie_max then trie_max ← t;
    end
  else begin   { whoops, have to repack }
    t ← t − pat[i]; unpack(t);
    incr(qmax); q_char(qmax) ← pat[i]; q_link(qmax) ← 0; q_outp(qmax) ← 0;
    t ← first_fit; trie_link(s) ← t; t ← t + pat[i];
    end;
  incr(trie_count);
  end;
```

This code is used in section 31.

33. Routines for pattern count trie. The pattern count trie is used to store the set of patterns considered in the current pass, along with the counts of good and bad instances. The fields of this trie are the same as the pattern trie, except that there is no output field, and leaf nodes are also used to store counts (*triec_good* and *triec_bad*). Except where noted, the following routines are analogous to the pattern trie routines.

```
⟨ Globals in the outer block 2 ⟩ +≡
triec_max, triec_bmax, triec_count: triec_pointer;  { same as for pattern trie }
triec_kmax: triec_pointer;  { shows growth of trie during pass }
pat_count: integer;  { number of patterns in count trie }
```

34. [See *init_pattern_trie*.] The variable *triec_kmax* always contains the size of the count trie rounded up to the next multiple of 4096, and is used to show the growth of the trie during each pass.

```
define triec_root = 1
procedure init_count_trie;
  var c: ascii_code;
  begin for c ← 0 to 127 do
    begin triec_char(triec_root + c) ← c;
    triec_link(triec_root + c) ← 0; triec_back(triec_root + c) ← 0;
    end;
  triec_base_used(triec_root) ← true; triec_bmax ← triec_root; triec_max ← triec_root + 127;
  triec_count ← 128; triec_kmax ← 4096;
  triec_link(0) ← triec_max + 1; triec_back(triec_max + 1) ← 0;
  pat_count ← 0;
  end;
```

35. [See *first_fit*.]

```
function firstc_fit: triec_pointer;
  label found, not_found;
  var a, b: triec_pointer; q: q_index;
  begin ⟨ Set b to the count trie base location at which this state should be packed 36 ⟩;
  for q ← 1 to qmax do  { pack it }
    begin a ← b + q_char(q);
    triec_link(triec_back(a)) ← triec_link(a); triec_back(triec_link(a)) ← triec_back(a);
    triec_char(a) ← q_char(q); triec_link(a) ← q_link(q); triec_back(a) ← q_back(q);
    if a > triec_max then triec_max ← a;
    end;
  triec_base_used(b) ← true; firstc_fit ← b
  end;
```

36. The threshold for attempting a first-fit packing is 3 transitions, which is lower than for the pattern trie because speed is more important here.

```
⟨ Set b to the count trie base location at which this state should be packed 36 ⟩ ≡
  if qmax > 3 then a ← triec_back(triec_max + 1) else a ← 0;
  while true do
    begin a ← triec_link(a); b ← a − q_char(1);
    ⟨ Ensure triec linked up to b + 128 37 ⟩;
    if triec_base_used(b) then goto not_found;
    for q ← qmax downto 2 do
      if triec_char(b + q_char(q)) > 0 then goto not_found;
    goto found;
  not_found: end;
found:
```

This code is used in section 35.

37.

```
⟨ Ensure triec linked up to b + 128 37 ⟩ ≡
  if b + 128 > triec_kmax then
    begin if triec_kmax = triec_size then error('Count␣trie␣too␣full!');
    print(triec_kmax div 1024 : 1, 'K␣');
    if triec_kmax + 4096 > triec_size then triec_kmax ← triec_size
    else triec_kmax ← triec_kmax + 4096;
    end;
  while triec_bmax < b do
    begin incr(triec_bmax); triec_base_used(triec_bmax) ← false; triec_char(triec_bmax + 127) ← 0;
    triec_link(triec_bmax + 127) ← triec_bmax + 128; triec_back(triec_bmax + 128) ← triec_bmax + 127;
    end
```

This code is used in section 36.

38. [See *unpack*.]

```
procedure unpackc(b : triec_pointer);
  var c: ascii_code; a: triec_pointer;
  begin qmax ← 0;
  for c ← cmin to cmax do {search for transitions belonging to this state}
    begin a ← b + c;
    if triec_char(a) = c then {found one}
      begin incr(qmax); q_char(qmax) ← c; q_link(qmax) ← triec_link(a);
      q_back(qmax) ← triec_back(a);
      triec_back(triec_link(0)) ← a; triec_link(a) ← triec_link(0); triec_link(0) ← a; triec_back(a) ← 0;
      triec_char(a) ← 0;
      end;
    end;
  triec_base_used(b) ← false;
  end;
```

39. [See *insert_pattern*.] Patterns being inserted into the count trie are always substrings of the current word, so they are contained in the array *word* with length *pat_len* and finishing position *fpos*.

```
function insertc_pat(fpos : word_index): triec_pointer;
  var spos: word_index; a,b: triec_pointer;
  begin spos ← fpos − pat_len; { starting position of pattern }
  incr(spos); b ← triec_root + word[spos]; a ← triec_link(b);
  while (a > 0) ∧ (spos < fpos) do { follow existing trie }
    begin incr(spos); a ← a + word[spos];
    if triec_char(a) ≠ word[spos] then ⟨Insert critical count transition, possibly repacking 40⟩;
    b ← a; a ← triec_link(a);
    end;
  q_link(1) ← 0; q_back(1) ← 0; qmax ← 1;
  while spos < fpos do { insert rest of pattern }
    begin incr(spos); q_char(1) ← word[spos]; a ← firstc_fit; triec_link(b) ← a; b ← a + word[spos];
    incr(triec_count);
    end;
  insertc_pat ← b; incr(pat_count);
  end;
```

40. ⟨Insert critical count transition, possibly repacking 40⟩ ≡

```
  begin if triec_char(a) = 0 then { lucky }
    begin triec_link(triec_back(a)) ← triec_link(a); triec_back(triec_link(a)) ← triec_back(a);
    triec_char(a) ← word[spos]; triec_link(a) ← 0; triec_back(a) ← 0;
    if a > triec_max then triec_max ← a;
    end
  else begin { have to repack }
    a ← a − word[spos]; unpackc(a);
    incr(qmax); q_char(qmax) ← word[spos]; q_link(qmax) ← 0; q_back(qmax) ← 0; a ← firstc_fit;
    triec_link(b) ← a; a ← a + word[spos];
    end;
  incr(triec_count);
  end;
```

This code is used in section 39.

41. Routines for traversing pattern tries. At the end of a pass, we traverse the count trie using the following recursive procedure, selecting good and bad patterns and inserting them into the pattern trie.

```
procedure traverse_count_trie(b : triec_pointer; i : dot_type);
    { traverse subtries of family b; i is current depth in trie }
  var c: ascii_code;  { a local variable that must be saved on recursive calls }
    a: triec_pointer;  { does not need to be saved }
  begin incr(i);
  for c ← cmin to cmax do  { find transitions belonging to this family }
    begin a ← b + c;
    if triec_char(a) = c then  { found one }
      begin pat[i] ← c;
      if i < pat_len then traverse_count_trie(triec_link(a), i)
      else ⟨ Decide what to do with this pattern 42 ⟩;
      end;
    end;
  end;
```

42. When we have come to the end of a pattern, *triec_good*(*a*) and *triec_bad*(*a*) contain the number of times this pattern helps or hinders the cause. We use the counts to determine if this pattern should be selected, or if it is hopeless, or if we can't decide yet. In the latter case, we set *more_to_come* true to indicate that there might still be good patterns extending the current type of patterns.

```
⟨ Decide what to do with this pattern 42 ⟩ ≡
  if good_wt * triec_good(a) < thresh then  { hopeless pattern }
    begin insert_pattern(max_val, pat_dot); incr(bad_pat_count)
    end
  else if good_wt * triec_good(a) − bad_wt * triec_bad(a) ≥ thresh then  { good pattern }
      begin insert_pattern(hyph_level, pat_dot); incr(good_pat_count);
      good_count ← good_count + triec_good(a); bad_count ← bad_count + triec_bad(a);
      end
    else more_to_come ← true
```

This code is used in section 41.

43. Some global variables are used to accumulate statistics about the performance of a pass.

```
⟨ Globals in the outer block 2 ⟩ +≡
good_pat_count, bad_pat_count: integer;  { number of patterns added at end of pass }
good_count, bad_count, miss_count: integer;  { hyphen counts }
level_pattern_count: integer;  { number of good patterns at level }
more_to_come: boolean;
```

44. The recursion in *traverse_count_trie* is initiated by the following procedure, which also prints some statistics about the patterns chosen. The "efficiency" is an estimate of pattern effectiveness.

```
  define bad_eff ≡ (thresh/good_wt)
procedure collect_count_trie;
  begin good_pat_count ← 0; bad_pat_count ← 0; good_count ← 0; bad_count ← 0; more_to_come ← false;
  traverse_count_trie(triec_root,0);
  print(good_pat_count : 1, ' good and ', bad_pat_count : 1, ' bad patterns added');
  level_pattern_count ← level_pattern_count + good_pat_count;
  if more_to_come then print_ln(' (more to come)') else print_ln(' ');
  print('finding ', good_count : 1, ' good and ', bad_count : 1, ' bad hyphens');
  if good_pat_count > 0 then
    print_ln(', efficiency = ', good_count/(good_pat_count + bad_count/bad_eff) : 1 : 2)
  else print_ln(' ');
  print_ln('pattern trie has ', trie_count : 1, ' nodes, ',
      'trie_max = ', trie_max : 1, ', ', op_count : 1, ' outputs');
  end;
```

45. At the end of a level, we traverse the pattern trie and delete bad patterns by removing their outputs. If no output remains, the node is also deleted.

```
function delete_patterns(s : trie_pointer): trie_pointer;
      { delete bad patterns in subtrie s, return 0 if entire subtrie freed, otherwise s }
  var c: ascii_code; t: trie_pointer; all_freed: boolean;  { must be saved on recursive calls }
    h,n: op_type;  { do not need to be saved }
  begin all_freed ← true;
  for c ← cmin to cmax do  { find transitions belonging to this family }
    begin t ← s + c;
    if trie_char(t) = c then
      begin ⟨ Link around bad outputs 46 ⟩;
      if trie_link(t) > 0 then trie_link(t) ← delete_patterns(trie_link(t));
      if (trie_link(t) > 0) ∨ (trie_outp(t) > 0) ∨ ((t ≤ trie_root + 127) ∧ (t ≥ trie_root)) then
        all_freed ← false
      else ⟨ Deallocate this node 47 ⟩;
      end;
    end;
  if all_freed then  { entire state is freed }
    begin trie_base_used(s) ← false; s ← 0;
    end;
  delete_patterns ← s;
  end;
```

46. ⟨ Link around bad outputs 46 ⟩ ≡

```
  begin h ← 0; hyf_nxt(0) ← trie_outp(t); n ← hyf_nxt(0);
  while n > 0 do
    begin if hyf_val(n) = max_val then hyf_nxt(h) ← hyf_nxt(n)
    else h ← n;
    n ← hyf_nxt(h);
    end;
  trie_outp(t) ← hyf_nxt(0);
  end
```

This code is used in section 45.

47. Cells freed by *delete_patterns* are put at the end of the free list.

```
⟨ Deallocate this node 47 ⟩ ≡
  begin trie_link(trie_back(trie_max + 1)) ← t; trie_back(t) ← trie_back(trie_max + 1);
  trie_link(t) ← trie_max + 1; trie_back(trie_max + 1) ← t; trie_char(t) ← 0;
  decr(trie_count);
  end
```

This code is used in section 45.

48. The recursion in *delete_patterns* is initiated by the following procedure, which also prints statistics about the number of nodes deleted, and zeros bad outputs in the hash table. Note that the hash table may become somewhat disorganized when more levels are added, but this defect isn't serious.

```
procedure delete_bad_patterns;
  var old_op_count: op_type; old_trie_count: trie_pointer; t: trie_pointer; h: op_type;
  begin old_op_count ← op_count; old_trie_count ← trie_count;
  t ← delete_patterns(trie_root);
  for h ← 1 to max_ops do
    if hyf_val(h) = max_val then
      begin hyf_val(h) ← 0; decr(op_count);
      end;
  print_ln(old_trie_count − trie_count : 1, ' nodes and ',
      old_op_count − op_count : 1, ' outputs deleted'); qmax_thresh ← 7;
      { pattern trie will be sparser because of deleted patterns }
  end;
```

49. After all patterns have been generated, we will traverse the pattern trie and output all patterns. Note that if a pattern appears more than once, only the maximum value at each position will be output.

```
procedure output_patterns(s : trie_pointer; pat_len : dot_type);
      { output patterns in subtrie s; pat_len is current depth in trie }
  var c: ascii_code;  { must be saved on recursive calls }
    t: trie_pointer; h: op_type; d: dot_type;
  begin incr(pat_len);
  for c ← cmin to cmax do
    begin t ← s + c;
    if trie_char(t) = c then
      begin pat[pat_len] ← c; h ← trie_outp(t);
      if h > 0 then ⟨ Output this pattern 50 ⟩;
      if trie_link(t) > 0 then output_patterns(trie_link(t), pat_len);
      end;
    end;
  end;
```

50. ⟨ Output this pattern 50 ⟩ ≡

```
begin for d ← 0 to pat_len do hval[d] ← 0;
repeat d ← hyf_dot(h);
  if hval[d] < hyf_val(h) then hval[d] ← hyf_val(h);
  h ← hyf_nxt(h);
until h = 0;
if hval[0] > 0 then write(hval[0] : 1);
for d ← 1 to pat_len do
  begin if pat[d] = edge_of_word then write('.')
  else write(xchr[pat[d]]);
  if hval[d] > 0 then write(hval[d] : 1);
  end;
writeln;
end
```

This code is used in section 49.

51. Dictionary processing routines. The procedures in this section are the "inner loop" of the pattern generation process. To speed the program up, key parts of these routines should be coded in machine language.

```
⟨ Globals in the outer block 2 ⟩ +≡
word: array [word_index] of ascii_code; { current word }
dots: array [word_index] of ascii_code; { current hyphens }
dotw: array [word_index] of integer; { dot weights }
hval: array [word_index] of val_type; { hyphenation values }
no_more: array [word_index] of boolean; { positions 'knocked out' }
wlen: word_index; { length of current word }
word_wt: integer; { global word weight }
wt_chg: boolean; { indicates word_wt has changed }
buf: array [1 .. 80] of text_char; { array to hold lines of input }
buf_ptr: 1 .. 80; { index into buf }
```

52. We assume the word list uses only the letters A through Z. "Dots" between letters can be one of four possibilities: "-" indicating a hyphen, "*" indicating a found hyphen, "." indicating an error, or nothing. Furthermore single-digit word weights are allowed. A digit at the beginning of a word indicates a global word weight that is to be applied to all following words (until the next global word weight). A digit at some intercharacter position indicates a weight for that position only.

The *read_word* procedure scans a line of input representing a word, and places the letters into the array *word*, with $word[1] = word[wlen] = edge\_of\_word$. The dot appearing between $word[dpos]$ and $word[dpos+1]$ is placed in $dots[dpos]$, and the corresponding dot weight in $dotw[dpos]$.

```
procedure read_word;
  var c: ascii_code;
  begin readln(dictionary, buf); word[1] ← edge_of_word; wlen ← 1; buf_ptr ← 1; c ← xord[buf[buf_ptr]];
  if (c ≤ "9") ∧ (c ≥ "0") then
    begin word_wt ← (c − "0"); { global word weight }
    wt_chg ← true; incr(buf_ptr);
    end;
  repeat c ← xord[buf[buf_ptr]];
    if c ≥ "A" then { record the letter c }
      begin incr(wlen); word[wlen] ← c; dots[wlen] ← 0; dotw[wlen] ← word_wt;
      end
    else if c ≥ "0" then dotw[wlen] ← (c − "0") { dot weight }
      else dots[wlen] ← c; { record the dot c }
    incr(buf_ptr);
  until buf[buf_ptr] = '␣';
  incr(wlen); word[wlen] ← edge_of_word; dotw[wlen − 3] ← 1;
      { dot position two letters from end is not weighted }
  end;
```

53. Here is a procedure that uses the existing patterns to hyphenate the current word. The hyphenation value applying between the characters *word*[*dpos*] and *word*[*dpos* + 1] is stored in *hval*[*dpos*].

In addition, *no_more*[*dpos*] is set to *true* if this position is "knocked out" by either a good or bad pattern at this level. That is, if the pattern with current length and hyphen position is a superstring of either a good or bad pattern at this level, then we don't need to collect count statistics for the pattern because it can't possibly be chosen in this pass. Thus we don't even need to insert such patterns into the count trie, which saves a good deal of space.

```
procedure hyphenate;
  label done;
  var spos, dpos, fpos: word_index; t: trie_pointer; h: op_type; v: val_type;
  begin for spos ← wlen − 3 downto 0 do
    begin no_more[spos] ← false; hval[spos] ← 0; fpos ← spos + 1; t ← trie_root + word[fpos];
    repeat h ← trie_outp(t);
      while h > 0 do ⟨ Store output h in the hval and no_more arrays, and advance h 54 ⟩;
      t ← trie_link(t);
      if t = 0 then goto done;
      incr(fpos); t ← t + word[fpos];
    until trie_char(t) ≠ word[fpos];
  done: end;
  end;
```

54. ⟨ Store output h in the *hval* and *no_more* arrays, and advance h 54 ⟩ ≡

```
  begin dpos ← spos + hyf_dot(h); v ← hyf_val(h);
  if (v < max_val) ∧ (hval[dpos] < v) then hval[dpos] ← v;
  if (v ≥ hyph_level) then { check if position knocked out }
    if ((fpos − pat_len) ≤ (dpos − pat_dot)) ∧ ((dpos − pat_dot) ≤ spos) then no_more[dpos] ← true;
  h ← hyf_nxt(h);
  end
```

This code is used in section 53.

55. The *change_dots* procedure updates the *dots* array representing the printing values of the hyphens. Initially, hyphens in the word list are represented by "-" and non-hyphen positions are represented by 0. A hyphen that is correctly found by some pattern is changed to "*", while erroneous hyphens are represented by ".". Similary, erroneous hyphens that are correctly inhibited are changed back to 0, and found hyphens that are inhibited are changed to "-". The routine also collects statistics about the number of good, bad, and missed hyphens.

```
define incr_wt(#) ≡ # ← # + dotw[dpos]
procedure change_dots;
  var dpos: word_index;
  begin for dpos ← wlen − 3 downto 3 do
    begin if hval[dpos] > 0 then
      if odd(hval[dpos]) then
        begin if dots[dpos] = "-" then dots[dpos] ← "*"
        else if dots[dpos] = 0 then dots[dpos] ← ".";
        end
      else begin if dots[dpos] = "." then dots[dpos] ← 0
        else if dots[dpos] = "*" then dots[dpos] ← "-";
        end;
    if dots[dpos] = "*" then incr_wt(good_count)
    else if dots[dpos] = "." then incr_wt(bad_count)
      else if dots[dpos] = "-" then incr_wt(miss_count);
    end;
  end;
```

56. The following procedure outputs the word as hyphenated by the current patterns, including any word weights.

```
procedure output_hyphenated_word;
  var dpos: word_index;
  begin if wt_chg then
    begin  { output global word weight }
    write(pattmp, word_wt : 1); wt_chg ← false
    end;
  if wlen < 6 then ⟨ Output short word 57 ⟩
  else begin write(pattmp, xchr[word[2]]);
    for dpos ← 3 to wlen − 3 do
      begin write(pattmp, xchr[word[dpos]]);
      if dots[dpos] > 0 then write(pattmp, xchr[dots[dpos]]);
      if (dotw[dpos] ≠ word_wt) ∧ (dpos < wlen − 3) then write(pattmp, dotw[dpos] : 1);
      end;
    writeln(pattmp, xchr[word[wlen − 2]], xchr[word[wlen − 1]]);
    end;
  end;
```

57. In case the word list contains words less than 4 letters long (which will never be hyphenated), we just output them without change.

```
⟨ Output short word 57 ⟩ ≡
  begin for dpos ← 2 to wlen − 1 do write(pattmp, xchr[word[dpos]]);
  writeln(pattmp);
  end
```

This code is used in section 56.

58. For each dot position in the current word, the *do_word* routine first checks to see if we need to consider it. It might be knocked out or a dot we don't care about. That is, when considering hyphenating patterns, for example, we don't need to count hyphens already found. If a relevant dot is found, we increment the count in the count trie for the corresponding pattern, inserting it first if necessary.

```
procedure do_word;
  label continue, done;
  var spos, dpos, fpos: word_index; a: triec_pointer; goodp: boolean;
  begin for dpos ← wlen − 3 downto 3 do
    begin spos ← dpos − pat_dot; fpos ← spos + pat_len;
    ⟨ Check this dot position and goto continue if don't care 60 ⟩;
    incr(spos); a ← triec_root + word[spos];
    while spos < fpos do
      begin   { follow existing count trie }
      incr(spos); a ← triec_link(a) + word[spos];
      if triec_char(a) ≠ word[spos] then
        begin   { insert new count pattern }
        a ← insertc_pat(fpos); goto done;
        end;
      end;
  done: if goodp then incr_wt(triec_good(a)) else incr_wt(triec_bad(a));
  continue: end;
  end;
```

59. The globals *good_dot* and *bad_dot* will be set to "-" and 0, or "." and "*", depending on whether the current level is odd or even, respectively.

```
⟨ Globals in the outer block 2 ⟩ +≡
good_dot, bad_dot: ascii_code;
```

60. If the dot position *dpos* is out of bounds, knocked out, or a "don't care", we skip this position. Otherwise we set the flag *goodp* indicating whether this is a good or bad dot.

```
⟨ Check this dot position and goto continue if don't care 60 ⟩ ≡
  if (spos < 0) ∨ (fpos > wlen) ∨ no_more[dpos] then goto continue;
  if dots[dpos] = good_dot then goodp ← true
  else if dots[dpos] = bad_dot then goodp ← false
    else goto continue;
```

This code is used in section 58.

61. If *hyphp* is set to *true*, *do_dictionary* will write out a copy of the dictionary as hyphenated by the current set of patterns. If *procesp* is set to *true*, *do_dictionary* will collect pattern statistics for patterns with length *pat_len* and hyphen position *pat_dot*, at level *hyph_level*.

```
⟨ Globals in the outer block 2 ⟩ +≡
procesp, hyphp: boolean;
pat_dot: dot_type;   { hyphen position, measured from beginning of pattern }
hyph_level: val_type;   { hyphenation level }
filnam: packed array [1 .. 8] of char;   { for pattmp }
```

62. The following procedure makes a pass through the word list, and also prints out statistics about number of hyphens found and storage used by the count trie.

```
procedure do_dictionary;
  begin good_count ← 0; bad_count ← 0; miss_count ← 0; word_wt ← 1; wt_chg ← false;
  reset(dictionary);
  ⟨ Set up good_dot and bad_dot 63 ⟩;
  if procesp then
    begin init_count_trie;
    print_ln('processing␣dictionary␣with␣pat_len␣=␣', pat_len : 1, '␣pat_dot␣=␣', pat_dot : 1);
    end;
  if hyphp then
    begin ⟨ Initialize file name 64 ⟩;
    filnam[8] ← chr(hyph_level + ord('0')); rewrite(pattmp, filnam); print_ln('writing␣', filnam);
    end;
  ⟨ Process words until end of file 65 ⟩;
  print_ln('␣'); print_ln(good_count : 1, '␣good,␣', bad_count : 1, '␣bad,␣', miss_count : 1, '␣missed');
  print_ln((100 * good_count/(good_count + miss_count)) : 1 : 2, '␣%,␣',
      (100 * bad_count/(good_count + miss_count)) : 1 : 2, '␣%,␣',
      (100 * miss_count/(good_count + miss_count)) : 1 : 2, '␣%');
  if procesp then print_ln(pat_count : 1, '␣patterns,␣', triec_count : 1, '␣nodes␣in␣count␣trie,␣',
        'triec_max␣=␣', triec_max : 1);
  if hyphp then close(pattmp);
  end;
```

63. ⟨ Set up *good_dot* and *bad_dot* 63 ⟩ ≡

```
  if odd(hyph_level) then
    begin good_dot ← "-"; bad_dot ← 0;
    end
  else begin good_dot ← "."; bad_dot ← "*";
    end
```

This code is used in section 62.

64. ⟨ Initialize file name 64 ⟩ ≡

```
  begin filnam[1] ← 'p'; filnam[2] ← 'a'; filnam[3] ← 't'; filnam[4] ← 't'; filnam[5] ← 'm';
  filnam[6] ← 'p'; filnam[7] ← '.';
  end
```

This code is used in section 62.

65. ⟨ Process words until end of file 65 ⟩ ≡

```
  while ¬eof(dictionary) do
    begin read_word; hyphenate; change_dots;
    if hyphp then output_hyphenated_word;
    if procesp then do_word;
    end
```

This code is used in section 62.

66. Reading patterns. Before beginning a run, we can read in a file of existing patterns. This is useful for extending a previous pattern selection run to get some more levels. (Since these runs are quite time-consuming, it is convenient to choose patterns one level at a time, pausing to look at the results of the previous level, and possibly amending the dictionary.)

```
procedure read_patterns;
  var c: ascii_code;
  begin level_pattern_count ← 0; reset(patterns);
  while ¬eof(patterns) do
    begin readln(patterns, buf); incr(level_pattern_count);
    ⟨ Get pattern 67 ⟩;
    ⟨ Get dots and insert pattern 68 ⟩;
    end;
  print_ln(level_pattern_count : 1, ' patterns read in');
  print_ln('pattern trie has ', trie_count : 1, ' nodes, ',
      'trie_max = ', trie_max : 1, ', ', op_count : 1, ' outputs');
  end;
```

67. When a new pattern has been input into *buf*, we extract the letters of the pattern, and also convert "." to the *edge_of_word* symbol.

```
⟨ Get pattern 67 ⟩ ≡
  pat_len ← 0; buf_ptr ← 1;
  while buf[buf_ptr] ≠ ' ' do
    begin c ← xord[buf[buf_ptr]];
    if c ≥ "A" then
      begin incr(pat_len); pat[pat_len] ← c;
      end
    else if c = "." then
        begin incr(pat_len); pat[pat_len] ← edge_of_word;
        end;
    incr(buf_ptr);
    end
```

This code is used in section 66.

68. Then we scan *buf* again looking for hyphenation values (digits), and insert the pattern each time a value is found.

```
⟨ Get dots and insert pattern 68 ⟩ ≡
  pat_dot ← 0; buf_ptr ← 1;
  while buf[buf_ptr] ≠ ' ' do
    begin c ← xord[buf[buf_ptr]];
    if (c ≤ "9") ∧ (c ≥ "0") then insert_pattern(c − "0", pat_dot)
    else incr(pat_dot);
    incr(buf_ptr);
    end
```

This code is used in section 66.

69. The main program. This is where PATGEN actually starts. We initialize the pattern trie, get *hyph_level* and *pat_len* limits from the terminal, and generate patterns.

```
begin initialize; {set up input/output translation tables}
init_pattern_trie; read_patterns; procesp ← true; hyphp ← false;
print('hyph_start␣=␣'); input(hyph_start);
print('hyph_finish␣=␣'); input(hyph_finish);
for hyph_level ← hyph_start to hyph_finish do
  begin level_pattern_count ← 0;
  for pat_dot ← 0 to max_dot do level_no_more[pat_dot] ← false;
  if hyph_level > hyph_start then print_ln('␣');
  print('pat_start␣=␣'); input(pat_start);
  print('pat_finish␣=␣'); input(pat_finish);
  print('good␣weight,␣bad␣weight,␣threshold:␣'); input(good_wt, bad_wt, thresh);
  ⟨Generate a level 71⟩;
  delete_bad_patterns;
  print_ln('total␣of␣', level_pattern_count : 1, '␣patterns␣at␣hyph_level␣', hyph_level : 1);
  end;
output_patterns(trie_root, 0);
⟨Make final pass to hyphenate word list 72⟩;
end_of_PATGEN: end.
```

70. The patterns of a given length (at a given level) are chosen with dot positions ordered in an "organ-pipe" fashion. For example, for *pat_len* = 4 we choose patterns for different dot positions in the order 2, 1, 3, 0, 4. The variables *dot1* and *dot2* control this iteration in a clever manner.

```
⟨Globals in the outer block 2⟩ +≡
dot1, dot2: dot_type;
level_no_more: array [dot_type] of boolean;
```

71. The array *level_no_more* remembers which positions are permanently "knocked out". That is, if there aren't any possible good patterns remaining at a certain dot position, we don't need to consider longer patterns at this level containing that position.

```
⟨Generate a level 71⟩ ≡
  for pat_len ← pat_start to pat_finish do
    begin pat_dot ← pat_len div 2; dot1 ← pat_dot * 2; dot2 ← pat_len * 2 − 1;
    repeat pat_dot ← dot1 − pat_dot; dot1 ← dot2 − dot1;
      if level_no_more[pat_dot] then goto continue;
      do_dictionary; collect_count_trie;
      if ¬more_to_come then level_no_more[pat_dot] ← true;
    continue: until pat_dot = pat_len;
    for pat_dot ← pat_len downto 0 do
      if level_no_more[pat_dot] then level_no_more[pat_dot + 1] ← true;
    end;
```

This code is used in section 69.

72. When all patterns have been found, the user has a chance to see what they do. The resulting pattmp file can be used as the new 'dictionary' if we want to continue pattern generation from this point.

```
⟨Make final pass to hyphenate word list 72⟩ ≡
  procesp ← false; hyphp ← true; hyph_level ← hyph_finish;
  print('hyphenate␣word␣list?␣'); input_ln(buf[1]);
  if (buf[1] = 'Y') ∨ (buf[1] = 'y') then do_dictionary
```

This code is used in section 69.

73. Index.

⟨ Check this dot position and **goto** *continue* if don't care 60 ⟩ Used in section 58.
⟨ Compiler directives 10 ⟩ Used in section 1.
⟨ Constants in the outer block 18 ⟩ Used in section 1.
⟨ Deallocate this node 47 ⟩ Used in section 45.
⟨ Decide what to do with this pattern 42 ⟩ Used in section 41.
⟨ Ensure *triec* linked up to $b + 128$ 37 ⟩ Used in section 36.
⟨ Ensure *trie* linked up to $s + 128$ 27 ⟩ Used in section 26.
⟨ Generate a level 71 ⟩ Used in section 69.
⟨ Get dots and insert pattern 68 ⟩ Used in section 66.
⟨ Get pattern 67 ⟩ Used in section 66.
⟨ Globals in the outer block 2, 4, 12, 20, 21, 23, 30, 33, 43, 51, 59, 61, 70 ⟩ Used in section 1.
⟨ Initialize file name 64 ⟩ Used in section 62.
⟨ Insert critical count transition, possibly repacking 40 ⟩ Used in section 39.
⟨ Insert critical transition, possibly repacking 32 ⟩ Used in section 31.
⟨ Labels in the outer block 7 ⟩ Used in section 1.
⟨ Link around bad outputs 46 ⟩ Used in section 45.
⟨ Local variables for initialization 13 ⟩ Used in section 1.
⟨ Make final pass to hyphenate word list 72 ⟩ Used in section 69.
⟨ Output short word 57 ⟩ Used in section 56.
⟨ Output this pattern 50 ⟩ Used in section 49.
⟨ Process words until end of file 65 ⟩ Used in section 62.
⟨ Set up input/output translation tables 14, 15 ⟩ Used in section 1.
⟨ Set up *good_dot* and *bad_dot* 63 ⟩ Used in section 62.
⟨ Set b to the count trie base location at which this state should be packed 36 ⟩ Used in section 35.
⟨ Set s to the trie base location at which this state should be packed 26 ⟩ Used in section 25.
⟨ Store output h in the *hval* and *no_more* arrays, and advance h 54 ⟩ Used in section 53.
⟨ Types in the outer block 11, 19 ⟩ Used in section 1.

TEX82 hyphenation patterns

.ach4	.en3s	.mo3ro	.under5	age4o	a2n	apoc5	as1tr	avi4er
.ad4der	.eq5ui5t	.mu5ta	.un1e	4ageu	an3age	ap5ola	asur5a	av3ig
.af1t	.er4ri	.muta5b	.un5k	ag1i	3analy	apor5i	a2ta	av5oc
.al3t	.es3	.ni4c	.un5o	4ag4l	a3nar	apos3t	at3abl	a1vor
.am5at	.eu3	.od2	.un3u	ag1n	an3arc	aps5es	at5ac	3away
.an5c	.eye5	.odd5	.up3	a2go	anar4i	a3pu	at3alo	aw3i
.ang4	.fes3	.of5te	.ure3	3agog	a3nati	aque5	at5ap	aw4ly
.ani5m	.for5mer	.or5ato	.us5a	ag3oni	4and	2a2r	ate5c	aws4
.ant4	.ga2	.or3c	.ven4de	a5guer	ande4s	ar3act	at5ech	ax4ic
.an3te	.ge2	.or1d	.ve5ra	ag5ul	an3dis	a5rade	at3ego	ax4id
.anti5s	.gen3t4	.or3t	.wil5i	a4gy	an1dl	ar5adis	at3en.	ay5al
.ar5s	.ge5og	.os3	.ye4	a3ha	an4dow	ar3al	at3era	aye4
.ar4tie	.gi5a	.os4tl	4ab.	a3he	a5nee	a5ramete	ater5n	ays4
.ar4ty	.gi4b	.oth3	a5bal	ah4l	a3nen	aran4g	a5terna	azi4er
.as3c	.go4r	.out3	a5ban	a3ho	an5est.	ara3p	at3est	azz5i
.as1p	.hand5i	.ped5al	abe2	ai2	a3neu	ar4at	at5ev	5ba.
.as1s	.han5k	.pe5te	ab5erd	a5ia	2ang	a5ratio	4ath	bad5ger
.aster5	.he2	.pe5tit	abi5a	a3ic.	ang5ie	ar5ativ	ath5em	ba4ge
.atom5	.hero5i	.pi4e	ab5it5ab	ai5ly	an1gl	a5rau	a5then	bal1a
.au1d	.hes3	.pio5n	ab5lat	a4i4n	a4n1ic	ar5av4	at4ho	ban5dag
.av4i	.het3	.pi2t	ab5o5liz	ain5in	a3nies	araw4	ath5om	ban4e
.awn4	.hi3b	.pre3m	4abr	ain5o	an3i3f	arbal4	4ati.	ban3i
.ba4g	.hi3er	.ra4c	ab5rog	ait5en	an4ime	ar4chan	a5tia	barbi5
.ba5na	.hon5ey	.ran4t	ab3ul	a1j	a5nimi	ar5dine	at5i5b	bari4a
.bas4e	.hon3o	.ratio5na	a4car	ak1en	a5nine	ar4dr	at1ic	bas4si
.ber4	.hov5	.ree2	ac5ard	al5ab	an3io	ar5eas	at3if	1bat
.be5ra	.id4l	.re5mit	ac5aro	al3ad	a3nip	a3ree	ation5ar	ba4z
.be3sm	.idol3	.res2	a5ceou	a4lar	an3ish	ar3ent	at3itu	2b1b
.be5sto	.im3m	.re5stat	ac1er	4aldi	an3it	a5ress	a4tog	b2be
.bri2	.im5pin	.ri4g	a5chet	2ale	a3niu	ar4fi	a2tom	b3ber
.but4ti	.in1	.rit5u	4a2ci	al3end	an4kli	ar4fl	at5omiz	bbi4na
.cam4pe	.in3ci	.ro4q	a3cie	a4lenti	5anniz	ar1i	a4top	4b1d
.can5c	.ine2	.ros5t	ac1in	a5le5o	ano4	ar5ial	a4tos	4be.
.capa5b	.in2k	.row5d	a3cio	al1i	an5ot	ar3ian	a1tr	beak4
.car5ol	.in3s	.ru4d	ac5rob	al4ia.	anoth5	a3riet	at5rop	beat3
.ca4t	.ir5r	.sci3e	act5if	ali4e	an2sa	ar4im	at4sk	4be2d
.ce4la	.is4i	.self5	ac3ul	al5lev	an4sco	ar5inat	at4tag	be3da
.ch4	.ju3r	.sell5	a4cum	4allic	an4sn	ar3io	at5te	be3de
.chill5i	.la4cy	.se2n	2ad	4alm	an2sp	ar2iz	at4th	be3di
.ci2	.la4m	.se5rie	ad4din	a5log.	ans3po	ar2mi	a2tu	be3gi
.cit5r	.lat5er	.sh2	ad5er.	a4ly.	an4st	ar5o5d	at5ua	be5gu
.co3e	.lath5	.si2	2adi	4alys	an4sur	a5roni	at5ue	1bel
.co4r	.le2	.sing4	a3dia	5a5lyst	antal4	a3roo	at3ul	be1li
.cor5ner	.leg5e	.st4	ad3ica	5alyt	an4tie	ar2p	at3ura	be3lo
.de4moi	.len4	.sta5bl	adi4er	3alyz	4anto	ar3q	a2ty	4be5m
.de3o	.lep5	.sy2	a3dio	4ama	an2tr	arre4	au4b	be5nig
.de3ra	.lev1	.ta4	a3dit	am5ab	an4tw	ar4sa	augh3	be5nu
.de3ri	.li4g	.te2	a5diu	am3ag	an3ua	ar2sh	au3gu	4bes4
.des4c	.lig5a	.ten5an	ad4le	ama5ra	an3ul	4as.	au4l2	be3sp
.dictio5	.li2n	.th2	ad3ow	am5asc	a5nur	as4ab	aun5d	be5str
.do4t	.li3o	.ti2	ad5ran	a4matis	4ao	as3ant	au3r	3bet
.du4c	.li4t	.til4	ad4su	a4m5ato	apar4	ashi4	au5sib	bet5iz
.dumb5	.mag5a5	.tim5o5	4adu	am5era	ap5at	a5sia.	aut5en	be5tr
.earth5	.mal5o	.ting4	a3duc	am3ic	ap5ero	a3sib	au1th	be3tw
.eas3i	.man5a	.tin5k	ad5um	am5if	a3pher	a3sic	a2va	be3w
.eb4	.mar5ti	.ton4a	ae4r	am5ily	4aphi	5a5si4t	av3ag	be5yo
.eer4	.me2	.to4p	aeri4e	am1in	a4pilla	ask3i	a5van	2bf
.eg2	.mer3c	.top5i	a2f	ami4no	ap5illar	as4l	ave4no	4b3h
.el5d	.me5ter	.tou5s	aff4	a2mo	ap3in	a4soc	av3era	bi2b
.el3em	.mis1	.trib5ut	a4gab	a5mon	ap3ita	as5ph	av5ern	bi4d
.enam3	.mist5i	.un1a	aga4n	amor5i	a3pitu	as4sh	av5ery	3bie
.en3g	.mon3e	.un3ce	ag5ell	amp5en	a2pl	as3ten	av1i	bi5en

bi4er b5uto 3chemi co3pa 4daf d2gy 5dren e4ben efil4
2b3if b1v ch5ene cop3ic 2dag d1h2 dri4b e4bit e3fine
1bil 4b5w ch3er. co4pl da2m2 5di. dril4 e3br ef5i5nite
bi3liz 5by. ch3ers 4corb dan3g 1d4i3a dro4p e4cad 3efit
bina5r4 bys4 4chiin coro3n dard5 dia5b 4drow ecan5c efor5es
bin4d 1ca 5chine. cos4e dark5 di4cam 5drupli ecca5 e4fuse.
bi5net cab3in ch5iness cov1 4dary d4ice 4dry e1ce 4egal
bi3ogr ca1bl 5chini cove4 3dat 3dict 2di32 ec5essa eger4
bi5ou cach4 5chio cow5a 4dativ 3did ds4p ec2i eg5ib
bi2t ca5den 3chit coz5e 4dato 5di3en d4sw e4cib eg4ic
3bi3tio 4cag4 chi2z co5zi 5dav4 d1if d4sy ec5ificat eg5ing
bi3tr 2c5ah 3cho2 c1q dav5e di3ge d2th ec5ifie e5git5
3bit5ua ca3lat ch4ti cras5t 5day di4lato 1du ec5ify eg5n
b5itz cal4la 1ci 5crat. d1b d1in d1u1a ec3im e4go.
b1j call5in 3cia 5cratic d5c 1dina du2c eci4t e4gos
bk4 4calo ci2a5b cre3at d1d4 3dine. d1uca e5cite eg1ul
b212 can5d cia5r 5cred 2de. 5dini duc5er e4clam e5gur
blath5 can4e ci5c 4c3reta deaf5 di5niz 4duct. e4clus 5egy
b4le. can4ic 4cier cre4v deb5it 1dio 4ducts e2col e1h4
blen4 can5is 5cific. cri2 de4bon dio5g du5el e4comm eher4
5blesp can3iz 4cii cri5f decan4 di4pl du4g e4compe ei2
b3lis can4ty ci4la c4rin de4cil dir2 d3ule e4conc e5ic
b4lo cany4 3cili cris4 de5com di1re dum4be e2cor ei5d
blun4t ca5per 2cim 5criti 2d1ed dirt5i du4n ec3ora eig2
4b1m car5om 2cin cro4pl 4dee. dis1 4dup eco5ro ei5g1
4b3n cast5er c4ina crop5o de5if 5disi du4pe e1cr e3imb
bne5g cas5tig 3cinat cros4e deli4e d4is3t d1v e4crem e3inf
3bod 4casy cin3em cru4d del5i5q d2iti d1w ec4tan e1ing
bod3i ca4th c1ing 4c3s2 de5lo 1di1v d2y ec4te e5inst.
bo4e 4cativ c5ing. 2c1t d4em d1j 5dyn e1cu eir4d
bol3ic cav5al 5cino cta4b 5dem. d5k2 dy4se e4cul eit3e
bom4bi c3c cion4 ct5ang 3demic 4d5la dys5p ec3ula ei3th
bon4a ccha5 4cipe c5tant dem5ic. 3dle. e1a4b 2e2da e5ity
bon5at cci4a ci3ph c2te de5mil 3dled e3act 4ed3d e1j
3boo ccompa5 4cipic c3ter de4mons 3dles. ead1 e4d1er e4jud
5bor. ccon4 4cista c4ticu demor5 4dless ead5ie ede4s ej5udi
4b1ora ccou3t 4cisti ctim3i 1den 2d3lo ea4ge 4edi eki4n
bor5d 2ce. 2c1it ctu4r de4nar 4d5lu ea5ger e3dia ek4la
5bore 4ced. cit3iz c4tw de3no 2dly ea4l ed3ib e1la
5bori 4ceden 5ciz cud5 denti5f d1m eal5er ed3ica e4la.
5bos4 3cei ck1 c4uf de3nu 4d1n4 eal3ou ed3im e4lac
b5ota 5cel. ck3i c4ui de1p 1do eam3er ed1it elan4d
both5 3cell 1c4l4 cu5ity de3pa 3do. e5and edi5z el5ativ
bo4to 1cen 4clar 5culi depi4 do5de ear3a 4edo e4law
bound3 3cenc c5laratio cul4tis de2pu 5doe ear4c e4dol elaxa4
4bp 2cen4e 5clare 3cultu d3eq 2d5of ear5es edon2 e3lea
4brit 4ceni cle4m cu2ma d4erh d4og ear4ic e4dri el5ebra
broth3 3cent 4clic c3ume 5derm do4la ear4il e4dul 5elec
2b5s2 3cep clim4 cu4mi dern5iz doli4 ear5k ed5ulo e4led
bsor4 ce5ram cly4 3cun der5s do5lor ear2t ee2c el3ega
2bt 4cesa c5n cu3pi des2 dom5iz eart3e eed3i e5len
bt4l 3cessi 1co cu5py d2es. do3nat ea5sp ee2f e4lier
b4to ces5si5b co5ag cur5a4b de1sc doni4 e3ass eel3i e1les
b3tr ces5t coe2 cu5ria de2s5o doo3d east3 ee4ly el2f
buf4fer cet4 2cog 1cus des3ti dop4p ea2t ee2m el2i
bu4ga c5e4ta co4gr cuss4i de3str d4or eat5en ee4na e3libe
bu3li cew4 coi4 3c4ut de4su 3dos eath3i ee4pi e4l5ic.
bumi4 2ch co3inc cu4tie de1t 4d5out e5atif ee2s4 el3ica
bu4n 4ch. col5i 4c5utiv de2to do4v e4a3tu eest4 e3lier
bunt4i 4ch3ab 5colo 4cutr de1v 3dox ea2v ee4ty el5igib
bu3re 5chanic col3or 1cy dev3il d1p eav3en e5ex e5lim
bus5ie ch5a5nis com5er cze4 4dey 1dr eav5i e1f e4l3ing
buss4e che2 con4a 1d2a 4d1f drag5on eav5o e4f3ere e3lio
5bust cheap3 c4one 5da. d4ga 4drai 2e1b 1eff e2lis
4buta 4ched con3g 2d3a4b d3ge4t dre4 e4bel. e4fic el5ish
3butio che5lo con5t dach4 dg1i drea5r e4bels 5efici e3liv3

4ella e3ny. er3ine 4es2to 1fa flin4 4geno go3ni 1head
el4lab 4en3z e1rio e3ston fa3bl flo3re 4geny 5goo 3hear
ello4 e5of 4erit 2estr fab3r f2ly5 1geo go5riz he4can
e5loc eo2g er4iu e5stro fa4ce 4fm ge3om gor5ou h5ecat
el5og e4oi4 eri4v estruc5 4fag 4fn g4ery 5gos. h4ed
el3op. e3ol e4riva e2sur fain4 1fo 5gesi gov1 he5do5
el2sh eop3ar er3m4 es5urr fall5e 5fon geth5 g3p he314i
el4ta e1or er4nis es4w 4fa4ma fon4de 4geto 1gr hel4lis
e5lud eo3re 4ernit eta4b fam5is fon4t ge4ty 4grada hel4ly
el5ug eo5rol 5erniz eten4d 5far fo2r ge4v g4rai h5elo
e4mac eos4 er3no e3teo far5th fo5rat 4g1g2 gran2 hem4p
e4mag e4ot 2ero ethod3 fa3ta for5ay g2ge 5graph. he2n
e5man eo4to er5ob et1ic fa3the fore5t g3ger g5rapher hena4
em5ana e5out e5roc e5tide 4fato for4i gglu5 5graphic hen5at
em5b e5ow ero4r etin4 fault5 fort5a ggo4 4graphy heo5r
e1me e2pa er1ou eti4no 4f5b fos5 gh3in 4gray hep5
e2mel e3pai er1s e5tir 4fd 4f5p gh5out gre4n h4era
e4met ep5anc er3set e5titio 4fe. fra4t gh4to 4gress. hera3p
em3ica e5pel ert3er et5itiv feas4 f5rea 5gi. 4grit her4ba
emi4e e3pent 4ertl 4etn feath3 fres5c 1gi4a g4ro here5a
em5igra ep5etitio er3tw et5ona fe4b fri2 gia5r gruf4 h3ern
em1in2 ephe4 4eru e3tra 4feca fril4 g1ic gs2 h5erou
em5ine e4pli eru4t e3tre 5fect frol5 5gicia g5ste h3ery
em3i3ni e1po 5erwau et3ric 2fed 2f3s g4ico gth3 h1es
e4mis e4prec e1s4a et5rif fe3li 2ft gien5 gu4a he2s5p
em5ish ep5reca e4sage. et3rog fe4mo f4to 5gies. 3guard he4t
e5miss e4pred e4sages et5ros fen2d f2ty gil4 2gue het4ed
em3iz ep3reh es2c et3ua fend5e 3fu g3imen 5gui5t heu4
5emniz e3pro e2sca et5ym fer1 fu5el 3g4in. 3gun h1f
emo4g e4prob es5can et5z 5ferr 4fug gin5ge 3gus h1h
emoni5o ep4sh e3scr 4eu fev4 fu4min 5g4ins 4gu4t hi5an
em3pi ep5ti5b es5cu e5un 4f1f fu5ne 5gio g3w hi4co
e4mul e4put e1s2e e3up f4fes fu3ri 3gir 1gy high5
em5ula ep5uta e2sec eu3ro f4fie fusi4 gir4l 2g5y3n h4il2
emu3n e1q es5ecr eus4 f5fin. fus4s g3isl gy5ra himer4
e3my equi3l es5enc eute4 f2f5is 4futa gi4u h3ab4l h4ina
en5amo e4q3ui3s e4sert. euti5l f4fly 1fy 5giv hach4 hion4e
e4nant er1a e4serts eu5tr f2fy 1ga 3giz hae4m hi4p
ench4er era4b e4serva eva2p5 4fh gaf4 gl2 hae4t hir4l
en3dic 4erand 4esh e2vas 1fi 5gal. gla4 h5agu hi3ro
e5nea er3ar e3sha ev5ast fi3a 3gali glad5i ha3la hir4p
e5nee 4erati. esh5en e5vea 2f3ic. ga3lo 5glas hala3m hir4r
en3em 2erb e1si ev3ell 4f3ical 2gam 1gle ha4m his3el
en5ero er4bl e2sic evel3o f3ican ga5met gli4b han4ci his4s
en5esi er3ch e2sid e5veng 4ficate g5amo g3lig han4cy hith5er
en5est er4che es5iden even4i f3icen gan5is 3glo 5hand. hi2v
en3etr 2ere. es5igna ev1er fi3cer ga3niz glo3r han4g 4hk
e3new e3real e2s5im e5verb fic4i gani5za g1m hang5er 4h114
en5ics ere5co es4i4n e1vi 5ficia 4gano g4my hang5o h1an4
e5nie ere3in esis4te ev3id 5ficie gar5n4 gn4a h5a5niz h2lo
e5nil er5el. esi4u evi4l 4fics gass4 g4na. han4k h1o3ri
e3nio er3emo e5skin e4vin fi3cu gath3 gnet4t han4te 4h1m
en3ish er5ena es4mi evi4v fi5del 4gativ g1ni hap3l hmet4
en3it er5ence e2sol e5voc fight5 4gaz g2nin hap5t 2h1n
e5niu 4erene es3olu e5vu fil5i g3b g4nio ha3ran h5odiz
5eniz er3ent e2son e1wa fill5in gd4 g1no ha5ras h5ods
4enn ere4q es5ona e4wag 4fily 2ge. g4non har2d ho4g
4eno er5ess e1sp e5wee 2fin 2ged 1go hard3e hoge4
eno4g er3est es3per e3wh 5fina geez4 3go. har4le hol5ar
e4nos eret4 es5pira ewil5 fin2d5 gel4in gob5 harp5en 3hol4e
en3ov er1h es4pre ew3ing fi2ne ge5lis 5goe har5ter ho4ma
en4sw er1i 2ess e3wit f1in3g ge5liz 3g4o4g has5s home3
ent5age e1ria4 es4si4b 1exp fin4n 4gely go3is haun4 hon4a
4enthes 5erick estan4 5eyc fis4ti 1gen gon2 5haz ho5ny
en3ua e3rien es3tig 5eye. f4l2 ge4nat 4g3o3na haz3a 3hood
en5uf eri4er es5tim eys4 f5less ge5niz gondo5 h1b hoon4

hor5at 4iceo ig3in 4ingu ir4min it3uat k1m 3less 13leg
ho5ris 4ich ig3it 2ini iro4g i5tud k5nes 5less. 131el
hort3e 2ici i4g4l i5ni. 5iron. it3ul 1k2no 13eva 131e4n
ho5ru i5cid i2go i4nia ir5ul 4itz. ko5r lev4er. 131e4t
hos4e ic5ina ig3or in3io 2is. iiu kosh4 lev4era 112i
ho5sen i2cip ig5ot iniis is5ag 2iv k3ou lev4ers 121in4
hosip ic3ipa i5gre i5nite. is3ar iv3ell kro5n 3ley 151ina
1hous i4cly igu5i 5initio isas5 iv3en. 4k1s2 4leye 114o
house3 i2c5oc igiur in3ity 2isic i4v3er. k4sc 21f 11oqui5
hov5el 4iicr i3h 4ink is3ch i4vers. ks4l 15fr 115out
4h5p 5icra 4i5i4 4inl 4ise iv5il. k4sy 411g4 151ow
4hr4 i4cry i3j 2inn is3er iv5io k5t 15ga 21m
hree5 ic4te 4ik 2iino 3isf iviit k1w 1gar3 15met
hro5niz ictu2 iila i4no4c is5han i5vore lab3ic 14ges 1m3ing
hro3po ic4t3ua il3a4b ino4s is3hon iv3o3ro 14abo 1go3 14mod
4his2 ic3ula i4lade i4not ish5op i4v3ot laci4 213h 1mon4
h4sh ic4um i2l5am 2ins is3ib 4i5w 14ade 1i4ag 211n2
h4tar ic5uo ila5ra in3se isi4d ix4o la3dy 1i2am 31o.
ht1en i3cur i3leg insur5a i5sis 4iy lag4n liar5iz 1ob5al
ht5es 2id ili1er 2int. is5itiv 4izar lam3o 1i4as 1o4ci
h4ty i4dai ilev4 2in4th 4is4k izi4 3land 1i4ato 41of
hu4g id5anc il5f iniu islan4 5izont lan4dl 1i5bi 3logic
hu4min id5d ilii i5nus 4isms 5ja lan5et 5licio 15ogo
hun5ke ide3al il3ia 4iny i2so jac4q lan4te 1i4cor 3logu
hun4t ide4s il2ib 2io iso5mer ja4p lar4g 4lics 1om3er
hus3t4 i2di il3io 4io. isip ije lar3i 4lict. 5long
hu4t id5ian il4ist ioge4 is2pi jer5s las4e 14icu lon4i
hiw idi4ar 2ilit io2gr is4py 4jestie la5tan 13icy 13o3niz
h4wart i5die il2iz iiol 4isis 4jesty 4lateli 13ida lood5
hy3pe id3io ill5ab io4m is4sal jew3 4lativ lid5er 5lope.
hy3ph idi5ou 4iln ion3at issen4 jo4p 4lav 3lidi lop3i
hy2s idiit il3oq ion4ery is4ses 5judg la4v4a 1if3er 13opm
2ii1a id5iu il4ty ion3i is4ta. 3ka. 211b 14iff lora4
i2al i3dle il5ur io5ph isite k3ab 1bin4 1i4fl lo4rato
iam4 i4dom il3v ior3i isiti k5ag 411c2 5ligate lo5rie
iam5ete id3ow i4mag i4os ist4ly kais4 1ce4 3ligh lor5ou
i2an i4dr im3age io5th 4istral kal4 13ci 1i4gra 5los.
4ianc i2du ima5ry i5oti i2su k1b 21d 3lik los5et
ian3i id5uo imenta5r io4to is5us k2ed 12de 414i4l 5losophiz
4ian4t 2ie4 4imet i4our 4ita. 1kee ld4ere lim4bl 5losophy
ia5po ied4e imii 2ip ita4bi ke4g ld4eri lim3i los4t
iass4 5ie5ga im5ida ipe4 i4tag ke5li ldi4 1i4mo lo4ta
i4ativ ield3 imi5le iphras4 4ita5m k3en4d ld5is 14im4p loun5d
ia4tric ien5a4 i5mini ip3i i3tan k1er 13dr 14ina 21out
i4atu ien4e 4imit ip4ic i3tat kes4 14dri 114ine 41ov
ibe4 i5enn im4ni ip4re4 2ite k3est. le2a lin3ea 21p
ib3era i3enti i3mon ip3ul it3era ke4ty le4bi lin3i 1pa5b
ib5ert i1er. i2mu i3qua i5teri k3f left5 link5er 13pha
ib5ia i3esc im3ula iq5uef it4es kh4 5leg. li5og 15phi
ib3in i1est 2in. iq3uid 2ith k1i 5legg 414iq 1p5ing
ib5it. i3et i4n3au iq3ui3t i1ti 5ki. le4mat lis4p 13pit
ib5ite 4if. 4inav 4ir 4itia 5k2ic lem5atic 1lit 14pl
i1bl if5ero incel4 i1ra 4i2tic k4ill 4len. 12it. 15pr
ib31i iff5en in3cer ira4b it3ica kilo5 3lenc 5litica 411r
i5bo if4fr 4ind i4rac 5i5tick k4im 5lene. 15i5tics 211s2
i1br 4ific. in5dling ird5e it3ig k4in. 1lent liv3er 14sc
i2b5ri i3fie 2ine ire4de it5ill kin4de le3ph 1liz 12se
i5bun i3fl i3nee i4ref i2tim k5iness le4pr 41j 14sie
4icam 4ift iner4ar i4rel4 2itio kin4g lera5b lka3 41t
5icap 2ig i5ness i4res 4itis ki4p ler4e 13kal lt5ag
4icar iga5b 4inga ir5gi i4tism kis4 3lerg lka4t ltane5
i4car. ig3era 4inge iri1 i2t5o5m k5ish 3l4eri 111 11te
i4cara ight3i in5gen iri5de 4iton kk4 14ero 14law lten4
icas5 4igi 4ingi ir4is i4tram kil les2 12le ltera4
i4cay i3gib in5gling iri3tu it5ry 4kley le5sco 15lea lth3i
iccu4 ig3il 4ingo 5i5r2iz 4itt 4kly 5lesq 13lec 15ties.

ltis4 4me. m4nin n5act ne4po nk3in nti2f o2fi ol3ume
l1tr 2med mn4o nag5er. ne2q n1kl n3tine of5ite ol3un
ltu2 4med. 1mo nak4 n1er 4n1l n4t3ing ofit4t o5lus
ltur3a 5media 4mocr na4li nera5b n5m nti4p o2g5a5r ol2v
lu5a me3die 5mocratiz na5lia n4erar nme4 ntrol5li og5ativ o21y
lu3br m5e5dy mo2di 4nalt n2ere nmet4 nt4s o4gato om5ah
luch4 me2g mo4go na5mit n4er5i 4n1n2 ntu3me o1ge oma5l
lu3ci mel5on mois2 n2an ner4r nne4 nu1a o5gene om5atiz
lu3en mel4t moi5se nanci4 1nes nni3al nu4d o5geo om2be
luf4 me2m 4mok nan4it 2nes. nni4v nu5en o4ger om4bl
lu5id mem1o3 mo5lest nank4 4nesp nob4l nuf4fe o3gie o2me
lu4ma 1men mo3me nar3c 2nest no3ble n3uin 1o1gis om3ena
5lumi men4a mon5et 4nare 4nesw n5ocl 3nu3it og3it om5erse
15umn. men5ac mon5ge nar3i 3netic 4n3o2d n4um o4gl o4met
5lumnia men4de moni3a nar4l ne4v 3noe nu1me o5g2ly om5etry
lu3o 4mene mon4ism n5arm n5eve 4nog n5umi 3ogn o3mia
luo3r men4i mon4ist n4as ne4w noge4 3nu4n o4g om3ic.
4lup mens4 mo3niz nas4c 13f nois5i n3uo ogu om3ica
luss4 mensu5 monol4 nas5ti 4gab no5l4i nu3tr 1og o5mid
lus3te 3ment mo3ny. n2at 13gel 5nologis n1v2 2ogy om1in
1lut men4te mo2r na3tal nge4n4e 3nomic n1w4 o1h2 o5mini
15ven me5on 4mora. nato5miz n5gere n5o5miz nym4 ohab5 5ommend
15vet4 m5ersa mos2 n2au n3geri no4mo nyp4 oi2 omo4ge
211v 2mes mo5sey nau3se ng5ha no3my 4nz oic3es o4mon
1ly 3mesti mo3sp 3naut n3gib no4n n3za oi3der om3pi
4lya me4ta moth3 nav4e ng1in non4ag 4oa oiff4 ompro5
4lyb met3al m5ouf 4n1b4 n5git non5i oad3 oig4 o2n
ly5me meite 3mous ncar5 n4gla n5oniz o5a5les oi5let on1a
ly3no me5thi mo2v n4ces. ngov4 4nop oard3 o3ing on4ac
21ys4 m4etr 4m1p n3cha ng5sh 5nop5o5li oas4e oint5er o3nan
15yse 5metric mpara5 n5cheo n1gu nor5ab oast5e o5ism on1c
1ma me5trie mpa5rab n5chil n4gum no4rary oat5i oi5son 3oncil
2mab me3try mpar5i n3chis n2gy 4nosc ob3a3b oist5en 2ond
ma2ca me4v m3pet nc1in 4n1h4 nos4e o5bar oi3ter on5do
ma5chine 4m1f mphas4 nc4it nha4 nos5t obe4l o5j o3nen
ma4cl 2mh m2pi ncour5a nhab3 no5ta o1bi 2ok on5est
mag5in 5mi. mpi4a n1cr nhe4 1nou o2bin o3ken on4gu
5magn mi3a mp5ies n1cu 3n4ia 3noun ob5ing ok5ie on1ic
2mah mid4a m4p1in n4dai ni3an nov3el3 o3br o1la o3nio
maid5 mid4g m5pir n5dan ni4ap now13 ob3ul o4lan on1is
4mald mig4 mp5is n1de ni3ba n1p4 o1ce olass4 o5niu
ma31ig 3milia mpo3ri nd5est. ni4bl npi4 och4 o12d on3key
ma51in m5i5lie mpos5ite ndi4b ni4d npre4c o3chet oldie on4odi
mal4li m4ill m4pous n5d2if ni5di n1q ocif3 o13er on3omy
mal4ty min4a mpov5 n1dit ni4er n1r o4cil o3lesc on3s
5mania 3mind mp4tr n3diz ni2fi nru4 o4clam o3let onspi4
man5is m5inee m2py n5duc ni5ficat 2n1s2 o4cod o14fi onspir5a
man3iz m4ingl 4m3r ndu4r n5igr ns5ab oc3rac o12i onsu4
4map min5gli 4m1s2 nd2we nik4 nsati4 oc5ratiz o3lia onten4
ma5rine. m5ingly m4sh 2ne. n1im ns4c ocre3 o3lice on3t4i
ma5riz min4t m5si n3ear ni3miz n2se 5ocrit o15id. ontif5
mar4ly m4inu 4mt ne2b n1in n4s3es octor5a o3li4f on5um
mar3v miot4 1mu neb3u 5nine. nsid1 oc3ula o5lil onva5
ma5sce m2is mula5r4 ne2c nin4g nsig4 o5cure o13ing oo2
mas4e mis4er. 5mult 5neck ni4o n2s1 od5ded o5lio ood5e
mas1t mis51 multi3 2ned 5nis. ns3m od3ic o5lis. ood5i
5mate mis4ti 3mum ne4gat nis4ta n4soc odi3o o13ish oo4k
math3 m5istry mun2 neg5ativ n2it ns4pe o2do4 o5lite oop3i
ma3tis 4mith 4mup 5nege n4ith n5spi odor3 o5litio o3ord
4matiza m2iz mu4u ne4la 3nitio nsta5bl od5uct. o5liv oost5
4m1b 4mk 4mw nel5iz n3itor n1t od5ucts olli4e o2pa
mba4t5 4mil 1na ne5mi ni3tr nta4b o4el o15ogiz ope5d
m5bil mim 2n1a2b ne4mo n1j nter3s o5eng olo4r op1er
m4b3ing mma5ry n4abu 1nen 4nk2 nt2i o3er o15pl 3opera
mbi4v 4min 4nac. 4nene n5kero n5tib oe4ta o12t 4operag
4m5c mn4a na4ca 3neo n3ket nti4er o3ev o13ub 2oph

o5phan o4tes pear4l pind4 proit rb4o rev5olu riv3et r5pent
o5pher 4oth pe2c p4ino 2pis2 r1c re4wh riv3i rp5er.
op3ing oth5esi 2p2ed 3pi1o p2se r2ce r1f r3j r3pet
o3pit oth3i4 3pede pion4 ps4h rcen4 rfu4 r3ket rp4h4
o5pon ot3ic. 3pedi p3ith p4sib r3cha r4fy rk4le rp3ing
o4posi ot5ica pedia4 pi5tha 2p1t rch4er rg2 rk4lin r3po
o1pr o3tice ped4ic pi2tu pt5a4b r4ci4b rg3er r1l r1r4
opiu o3tif p4ee 2p3k2 p2te rc4it r3get r1e4 rre4c
opy5 o3tis pee4d 1p2l2 p2th rcum3 r3gic r2led rre4f
o1q oto5s pek4 3plan pti3m r4dal rgi4n r4lig r4reo
o1ra ou2 pe4la plas5t ptu4r rd2i rg3ing r4lis rre4st
o5ra. ou3bl peli4e pli3a p4tw rdi4a r5gis r15ish rri4o
o4r3ag ouch5i pe4nan pli5er pub3 rdi4er r5git r3lo4 rri4v
or5aliz ou5et p4enc 4plig pue4 rdin4 r1gl r1m rron4
or5ange ou4l pen4th pli4n puf4 rd3ing rgo4n rma5c rros4
ore5a ounc5er pe5on ploi4 pul3c 2re. r3gu r2me rrys4
o5real oun2d p4era. plu4m pu4m re1al rh4 r3men 4rs2
or3ei ou5v pera5bl plum4b pu2n re3an 4rh. rm5ers r1sa
ore5sh ov4en p4erag 4p1m pur4r re5arr 4rhal rm3ing rsa5ti
or5est. over4ne p4eri 2p3n 5pus 5reav ri3a r4ming. rs4c
orew4 over3s peri5st po4c pu2t re4aw ria4b r4mio r2se
or4gu ov4ert per4mal 5pod. 5pute r5ebrat ri4ag r3mit r3sec
4o5ria o3vis perme5 po5em put3er rec5oll r4ib r4my rse4cr
or3ica oviti4 p4ern po3et5 pu3tr rec5ompe rib3a r4nar rs5er.
o5ril o5v4ol per3o 5po4g put4ted re4cre ric5as r3nel rs3es
or1in ow3der per3ti poin2 put4tin 2r2ed r4ice r4ner rse5v2
o1rio ow3el pe5ru 5point p3w re1de 4rici r5net r1sh
or3ity ow5est per1v poly5t qu2 re3dis 5ricid r3ney r5sha
o3riu ow1i pe2t po4ni qua5v red5it ri4cie r5nic r1si
or2mi own5i pe5ten po4p 2que. re4fac r4ico r1nis4 r4si4b
orn2e o4wo pe5tiz 1p4or 3quer re2fe rid5er r3nit rson3
o5rof oy1a 4pf po4ry 3quet re5fer. ri3enc r3niv r1sp
or3oug 1pa 4pg 1pos 2rab re3fi ri3ent rno4 r5sw
or5pe pa4ca 4ph. pos1s ra3bi re4fy ri1er r4nou rtach4
3orrh pa4ce phar5i p4ot rach4e reg3is ri5et r3nu r4tag
or4se pac4t phe3no po4ta r5acl re5it rig5an rob3l r3teb
ors5en p4ad ph4er 5poun raf5fi re1li 5rigi r2oc rten4d
orst4 5pagan ph4es. 4p1p raf4t re5lu ril3iz ro3cr rte5o
or3thi p3agat ph1ic ppa5ra r2ai r4en4ta 5riman ro4e r1ti
or3thy p4ai 5phie p2pe ra4lo ren4te rim5i ro1fe rt5ib
or4ty pain4 ph5ing p4ped ram3et re1o 3rimó ro5fil rti4d
o5rum p4al 5phisti p5pel r2ami re5pin rim4pe rok2 r4tier
o1ry pan4a 3phiz p3pen rane5o re4posi r2ina ro5ker r3tig
os3al pan3el ph2l p3per ran4ge re1pu 5rina. 5role. rti13i
os2c pan4ty 3phob p3pet r4ani r1er4 rin4d rom5ete rti14l
os4ce pa3ny 3phone ppo5site ra5no r4eri rin4e rom4i r4tily
o3scop pa1p 5phoni pr2 rap3er rero4 rin4g rom4p r4tist
4oscopi pa4pu pho4r pray4e 3raphy re5ru ri1o ron4al r4tiv
o5scr para5bl 4phs 5preci rar5c r4es. 5riph ron4e r3tri
os4i4e par5age ph3t pre5co rare4 re4spi riph5e ro5n4is rtroph4
os5itiv par5di 5phu pre3em rar5ef ress5ib ri2pl ron4ta rt4sh
os3ito 3pare 1phy pref5ac 4rari1 res2t rip5lic 1room ru3a
os3ity par5el pi3a pre4la r2as re5stal r4iq 5root ru3e4l
osi4u p4a4ri pian4 pre3r ration4 re3str r2is ro3pel ru3en
os4l par4is pi4cie p3rese rau4t re4ter r4is. rop3ic ru4gl
o2so pa2te pi4cy 3press ra5vai re4ti4z ris4c ror3i ru3in
os4pa pa5ter p4id pre5ten rav3el re3tri r3ish ro5ro rum3pl
os4po 5pathic p5ida pre3v ra5zie reu2 ris4p ros5per ru2n
os2ta pa5thy pi3de 5pri4e r1b re5uti ri3ta3b ros4s runk5
o5stati pa4tric 5pidi prin4t3 r4bab rev2 r5ited. ro4the run4ty
os5til pav4 3piec pri4s r4bag re4val rit5er. ro4ty r5usc
os5tit 3pay pi3en pris3o rbi2 rev3el rit5ers ro4va ruti5n
o4tan 4p1b pi4grap p3roca rbi4f r5ev5er. rit3ic rov5el rv4e
otele4g pd4 pi3lo prof5it r2bin re5vers ri2tu rox5 rvel4i
ot3er. 4pe. pi2n pro3l r5bine re5vert rit5ur r1p r3ven
ot5ers 3pe4a p4in. pros3e rb5ing. re5vil riv5el r4pea rv5er.

r5vest
r3vey
r3vic
rvi4v
r3vo
r1w
ry4c
5rynge
ry3t
sa2
2s1ab
5sack
sac3ri
s3act
5sai
salar4
sal4m
sa5lo
sal4t
3sanc
san4de
s1ap
sa5ta
5sa3tio
sat3u
sau4
sa5vor
5saw
4s5b
scan4t5
sca4p
scav5
s4ced
4scei
s4ces
sch2
s4cho
3s4cie
5scin4d
scle5
s4cli
scof4
4scopy
scour5a
s1cu
4s5d
4se.
se4a
seas4
sea5w
se2c3o
3sect
4s4ed
se4d4e
s5edl
se2g
seg3r
5sei
se1le
5self
5selv
4seme
se4mol
sen5at
4senc
sen4d
s5ened
sen5g
s5enin
4sentd
4sentl
sep3a3
4s1er.
s4erl
ser4o
4servo
s1e4s
se5sh
ses5t
5se5um
5sev
sev3en
sew4i
5sex
4s3f
2s3g
s2h
2sh.
sh1er
5shev
sh1in
sh3io
3ship
shiv5
sho4
sh5old
shon3
shor4
short5
4shw
si1b
s5icc
3side.
5sides
5sidi
si5diz
4signa
sil4e
4sily
2s1in
s2ina
5sine.
s3ing
1sio
5sion
sion5a
si2r
sir5a
1sis
3sitio
5siu
1siv
5siz
sk2
4ske
s3ket
sk5ine
sk5ing
s1l2
s3lat
s2le
slith5
2s1m
s3ma
small3
sman3
smel4
s5men
5smith
smol5d4
s1n4
1so
so4ce
soft3
so4lab
sol3d2
so3lic
5solv
3som
3s4on.
sona4
son4g
s4op
5sophic
s5ophiz
s5ophy
sor5c
sor5d
4sov
so5vi
2spa
5spai
spa4n
spen4d
2s5peo
2sper
s2phe
3spher
spho5
spil4
sp5ing
4spio
s4ply
s4pon
spor4
4spot
squal4l
s1r
2ss
s1sa
ssas3
s2s5c
s3sel
s5seng
s4ses.
s5set
s1si
s4sie
ssi4er
ss5ily
s4sl
ss4li
s4sn
sspend4
ss2t
ssur5a
ss5w
2st.
s2tag
s2tal
stam4i
5stand
s4ta4p
5stat.
s4ted
stern5i
s5tero
ste2w
stew5a
s3the
st2i
s4ti.
s5tia
s1tic
5stick
s4tie
s3tif
st3ing
5stir
s1tle
5stock
stom3a
5stone
s4top
3store
st4r
s4trad
5stratu
s4tray
s4trid
4stry
4st3w
s2ty
1su
su1al
su4b3
su2g3
su5is
suit3
s4ul
su2m
sum3i
su2n
su2r
4sv
sw2
4swo
s4y
4syc
3syl
syn5o
sy5rin
1ta
3ta.
2tab
ta5bles
5taboliz
4taci
ta5do
4taf4
tai5lo
ta2l
ta5la
tal5en
tal3i
4talk
tal4lis
ta5log
ta5mo
tan4de
tanta3
ta5per
ta5pl
tar4a
4tarc
4tare
ta3riz
tas4e
ta5sy
4tatic
ta4tur
taun4
tav4
2taw
tax4is
2t1b
4tc
t4ch
tch5et
4t1d
4te.
tead4i
4teat
tece4
5tect
2t1ed
te5di
1tee
teg4
te5ger
te5gi
3tel.
teli4
5tels
te2ma2
tem3at
3tenan
3tenc
3tend
4tenes
1tent
ten4tag
1teo
te4p
te5pe
ter3c
5ter3d
1teri
ter5ies
ter3is
teri5za
5ternit
ter5v
4tes.
4tess
t3ess.
teth5e
3teu
3tex
4tey
2t1f
4t1g
2th.
than4
th2e
4thea
th3eas
the5at
the3is
3thet
th5ic.
th5ica
4thil
5think
4thl
th5ode
5thodic
4thoo
thor5it
tho5riz
2ths
1tia
ti4ab
ti4ato
2ti2b
4tick
t4ico
t4ic1u
5tidi
3tien
tif2
ti5fy
2tig
5tigu
till5in
1tim
4timp
tim5ul
2t1in
t2ina
3tine.
3tini
1tio
ti5oc
tion5ee
5tiq
ti3sa
3tise
tis4m
ti5so
tis4p
5tistica
ti3tl
ti4u
1tiv
tiv4a
1tiz
ti3za
ti3zen
2tl
t5la
tlan4
3tle.
3tled
3tles.
t5let.
t5lo
4t1m
tme4
2t1n2
1to
to3b
to5crat
4todo
2tof
to2gr
to5ic
to2ma
tom4b
to3my
ton4ali
to3nat
4tono
4tony
to2ra
to3rie
tor5iz
tos2
5tour
4tout
to3war
4t1p
1tra
tra3b
tra5ch
traci4
trac4it
trac4te
tras4
tra5ven
trav5es5
tre5f
tre4m
trem5i
5tria
tri5ces
5tricia
4trics
2trim
tri4v
tro5mi
tron5i
4trony
tro5phe
tro3sp
tro3v
tru5i
trus4
4t1s2
t4sc
tsh4
t4sw
4t3t2
t4tes
t5to
ttu4
1tu
tu1a
tu3ar
tu4bi
tud2
4tue
4tuf4
5tu3i
3tum
tu4nis
2t3up.
3ture
5turi
tur3is
tur5o
tu5ry
3tus
4tv
tw4
4t1wa
twis4
4two
1ty
4tya
2tyl
type3
ty5ph
4tz
tz4e
4uab
uac4
ua5na
uan4i
uar5ant
uar2d
uar3i
uar3t
u1at
uav4
ub4e
u4bel
u3ber
u4bero
u1b4i
u4b5ing
u3ble.
u3ca
uci4b
uc4it
ucle3
u3cr
u3cu
u4cy
ud5d
ud3er
ud5est
udev4
u1dic
ud3ied
ud3ies
ud5is
u5dit
u4don
ud4si
u4du
u4ene
uens4
uen4te
uer4il
3ufa
u3fl
ugh3en
ug5in
2ui2
uil5iz
ui4n
u1ing
uir4m
uita4
uiv3
uiv4er.
u5j
4uk
u1la
ula5b
u5lati
ulch4
5ulche
ul3der
ul4e
u1len
ul4gi
ul2i
u5lia
ul3ing
ul5ish
ul4lar
ul4li4b
ul4lis
4ul3m
u1l4o
4uls
uls5es
ul1ti
ultra3
4ultu
u3lu
ul5ul
ul5v
um5ab
um4bi
um4bly
u1mi
u4m3ing
umor5o
um2p
unat4
u2ne
un4er
u1ni
un4im
u2nin
un5ish
uni3v
un3s4
un4sw
unt3ab
un4ter.
un4tes
unu4
un5y
un5z
u4ors
u5os
u1ou
u1pe
uper5s
u5pia

up3ing
u3pl
up3p
upport5
upt5ib
uptu4
u1ra
4ura.
u4rag
u4ras
ur4be
urc4
ur1d
ure5at
ur4fer
ur4fr
u3rif
uri4fic
ur1in
u3rio
u1rit
ur3iz
ur2l
url5ing.
ur4no
uros4
ur4pe
ur4pi
urs5er
ur5tes
ur3the
urti4
ur4tie
u3ru
2us
u5sad
u5san
us4ap
usc2
us3ci
use5a
u5sia
u3sic
us4lin
us1p
us5sl
us5tere
us1tr
u2su
usur4
uta4b
u3tat
4ute.
4utel
4uten
uten4i
4u1t2i
uti5liz
u3tine
ut3ing
ution5a
u4tis
5u5tiz
u4til
ut5of
uto5g
uto5matic
u5ton
u4tou
uts4
u3u
uu4m
u1v2
uxu3
uz4e
1va
5va.
2v1a4b
vac5il
vac3u
vag4
va4ge
va5lie
val5o
val1u
va5mo
va5niz
va5pi
var5ied
3vat
4ve.
4ved
veg3
v3el.
vel3li
ve4lo
v4ely
ven3om
v5enue
v4erd
5vere.
v4erel
v3eren
ver5enc
v4eres
ver3ie
vermi4n
3verse
ver3th
v4e2s
4ves.
ves4te
ve4te
vet3er
ve4ty
vi5ali
5vian
5vide.
5vided
4v3iden
5vides
5vidi
v3if
vi5gn
vik4
2vil
5vilit
v3i3liz
v1in
4vi4na
v2inc
vin5d
4ving
vio3l
v3io4r
vi1ou
vi4p
vi5ro
vis3it
vi3so
vi3su
4viti
vit3r
4vity
3viv
5vo.
voi4
3vok
vo4la
v5ole
5volt
3volv
vom5i
vor5ab
vori4
vo4ry
vo4ta
4votee
4vv4
v4y
w5abl
2wac
wa5ger
wag5o
wait5
w5al.
wam4
war4t
was4t
wa1te
wa5ver
w1b
wea5rie
weath3
wed4n
weet3
wee5v
wel4l
w1er
west3
w3ev
whi4
wi2
wil2
will5in
win4de
win4g
wir4
3wise
with3
wiz5
w4k
wl4es
wl3in
w4no
1wo2
wom1
wo5ven
w5p
wra4
wri4
writa4
w3sh
ws4l
ws4pe
w5s4t
4wt
wy4
x1a
xac5e
x4ago
xam3
x4ap
xas5
x3c2
x1e
xe4cuto
x2ed
xer4i
xe5ro
x1h
xhi2
xhil5
xhu4
x3i
xi5a
xi5c
xi5di
x4ime
xi5miz
x3o
x4ob
x3p
xpan4d
xpecto5
xpe3d
x1t2
x3ti
x1u
xu3a
xx4
y5ac
3yar4
y5at
y1b
y1c
y2ce
yc5er
y3ch
ych4e
ycom4
ycot4
y1d
y5ee
y1er
y4erf
yes4
ye4t
y5gi
4y3h
y1i
y3la
y1la5bl
y3lo
y5lu
ymbol5
yme4
ympa3
yn3chr
yn5d
yn5g
yn5ic
5ynx
y1o4
yo5d
y4o5g
yom4
yo5net
y4ons
y4os
y4ped
yper5
yp3i
y3po
y4poc
yp2ta
y5pu
yra5m
yr5ia
y3ro
yr4r
ys4c
y3s2e
ys3ica
ys3io
3ysis
y4so
yss4
ys1t
ys3ta
ysur4
y3thin
yt3ic
y1w
zai
z5a2b
zar2
4zb
2ze
ze4n
ze4p
z1er
ze3ro
zet4
2z1i
z4il
z4is
5zl
4zm
1zo
zo4m
zo5ol
zte4
4z1z2
z4zy

Answers

moun-tain-ous vil-lain-ous
be-tray-al de-fray-al por-tray-al
hear-ken
ex-treme-ly su-preme-ly
tooth-aches
bach-e-lor ech-e-lon
riff-raff
anal-o-gous ho-mol-o-gous
gen-u-ine
any-place
co-a-lesce
fore-warn fore-word
de-spair
ant-arc-tic corn-starch
mast-odon
squirmed

References

[1] Knuth, Donald E. *TEX and* METAFONT, *New Directions in Typesetting.* Digital Press, 1979.

[2] *Webster's Third New International Dictionary.* G. & C. Merriam, 1961.

[3] Knuth, Donald E. *The WEB System of Structured Documentation.* Preprint, Stanford Computer Science Dept., September 1982.

[4] Knuth, Donald E. *The Art of Computer Programming, Vol. 3, Sorting and Searching.* Addison-Wesley, 1973.

[5] Standish, T. A. *Data Structure Techniques.* Addison-Wesley, 1980.

[6] Aho, A. V., Hopcroft, J. E., and Ullman, J. D. *Algorithms and Data Structures.* Addison-Wesley, 1982.

[7] Bloom, B. Space/time tradeoffs in hash coding with allowable errors. *CACM 13*, July 1970, 422–436.

[8] Carter, L., Floyd, R., Gill, J., Markowsky, G., and Wegman, M. Exact and approximate membership testers. *Proc. 10th ACM SIGACT Symp.*, 1978, 59–65.

[9] de la Briandais, Rene. File searching using variable length keys. *Proc. Western Joint Computer Conf. 15*, 1959, 295–298.

[10] Fredkin, Edward. Trie memory. *CACM 3*, Sept. 1960, 490–500.

[11] Trabb Pardo, Luis. Set representation and set intersection. Ph.D. thesis, Stanford Computer Science Dept., December 1978.

[12] Mehlhorn, Kurt. Dynamic binary search. *SIAM J. Computing 8*, May 1979, 175–198.

[13] Maly, Kurt. Compressed tries. *CACM 19*, July 1976, 409–415.

[14] Knuth, Donald E. *TEX82.* Preprint, Stanford Computer Science Dept., September 1982.

[15] Resnikoff, H. L. and Dolby, J. L. The nature of affixing in written English. *Mechanical Translation 8*, 1965, 84–89. Part II, June 1966, 23-33.

[16] *The Merriam-Webster Pocket Dictionary.* G. & C. Merriam, 1974.

[17] Gorin, Ralph. SPELL.REG[UP,DOC] at SU-AI.

[18] Peterson, James L. Computer programs for detecting and correcting spelling errors. *CACM 23*, Dec. 1980, 676–687.

[19] Nix, Robert. Experience with a space-efficient way to store a dictionary. *CACM 24*, May 1981, 297–298.

[20] Morris, Robert and Cherry, Lorinda L. Computer detection of typographical errors. *IEEE Trans. Prof. Comm. PC-18*, March 1975, 54–64.

[21] Downey, P., Sethi, R., and Tarjan, R. Variations on the common subexpression problem. *JACM 27*, Oct. 1980, 758–771.

[22] Tarjan, R. E. and Yao, A. Storing a sparse table. *CACM 22*, Nov. 1979, 606–611.

[23] Zeigler, S. F. Smaller faster table driven parser. Unpublished manuscript, Madison Academic Computing Center, U. of Wisconsin, 1977.

[24] Aho, Alfred V. and Ullman, Jeffrey D. *Principles of Compiler Design*, sections 3.8 and 6.8. Addison-Wesley, 1977.

[25] Pfleeger, Charles P. State reduction in incompletely specified finite-state machines. *IEEE Trans. Computers C-22*, Dec. 1973, 1099–1102.

[26] Kohavi, Zvi. *Switching and Finite Automata Theory*, section 10-4. McGraw-Hill, 1970.

[27] Knuth, D. E., Morris, J. H., and Pratt, V. R. Fast pattern matching in strings. *SIAM J. Computing 6*, June 1977, 323–350.

[28] Aho, A. V. In R. V. Book (ed.), *Formal Language Theory: Perspectives and Open Problems.* Academic Press, 1980.

[29] Kucera, Henry and Francis, W. Nelson. *Computational Analysis of Present-Day American English.* Brown University Press, 1967.

[30] Research and Engineering Council of the Graphic Arts Industry. Proceedings of the 13th Annual Conference, 1963.

[31] Stevens, M. E. and Little, J. L. *Automatic Typographic-Quality Typesetting Techniques: A State-of-the-Art Review.* National Bureau of Standards, 1967.

[32] Berg, N. Edward. *Electronic Composition, A Guide to the Revolution in Typesetting.* Graphical Arts Technical Foundation, 1975.

[33] Rich, R. P. and Stone, A. G. Method for hyphenating at the end of a printed line. *CACM 8*, July 1965, 444–445.

[34] Wagner, M. R. The search for a simple hyphenation scheme. Bell Laboratories Technical Memorandum MM-71-1371-8.

[35] Gimpel, James F. *Algorithms in Snobol 4.* Wiley-Interscience, 1976.

[36] Ocker, Wolfgang A. A program to hyphenate English words. *IEEE Trans. Prof. Comm. PC-18*, June 1975, 78–84.

[37] Moitra, A., Mudur, S. P., and Narwekar, A. W. Design and analysis of a hyphenation procedure. *Software Prac. Exper. 9*, 1979, 325–337.

[38] Lindsay, R., Buchanan, B. G., Feigenbaum, E. A., and Lederberg, J. *DENDRAL*. McGraw-Hill, 1980.