SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 11, 1119-1184 (1981)

# Breaking Paragraphs into Lines*

DONALD E. KNUTH AND MICHAEL F. PLASS
*Computer Science Department, Stanford University, Stanford, California 94305, U.S.A.*

## SUMMARY

**This paper discusses a new approach to the problem of dividing the text of a paragraph into lines of approximately equal length. Instead of simply making decisions one line at a time, the method considers the paragraph as a whole, so that the final appearance of a given line might be influenced by the text on succeeding lines. A system based on three simple primitive concepts called 'boxes', 'glue', and 'penalties' provides the ability to deal satisfactorily with a wide variety of typesetting problems in a unified framework, using a single algorithm that determines optimum breakpoints. The algorithm avoids backtracking by a judicious use of the techniques of dynamic programming. Extensive computational experience confirms that the approach is both efficient and effective in producing high-quality output. The paper concludes with a brief history of line-breaking methods, and an appendix presents a simplified algorithm that requires comparatively few resources.**

KEY WORDS Typesetting Composition Line breaking Justification Dynamic programming Word processing Layout Spacing Box/glue/penalty algebra Shortest paths TEX (Tau Epsilon Chi) History of printing

## INTRODUCTION

One of the most important operations necessary when text materials are prepared for printing or display is the task of dividing long paragraphs into individual lines. When this job has been done well, people will not be aware of the fact that the words they are reading have been arbitrarily broken apart and placed into a somewhat rigid and unnatural rectangular framework; but if the job has been done poorly, readers will be distracted by bad breaks that interrupt their train of thought. In some cases it can be difficult to find suitable breakpoints; for example, the narrow columns often used in newspapers allow for comparatively little flexibility, and the appearance of mathematical formulas in technical text introduces special complications regardless of the column width. But even in comparatively simple cases like the typesetting of an ordinary novel, good line breaking will contribute greatly to the appearance and desirability of the finished product. In fact, some authors actually write better material when they are assured that it will look sufficiently beautiful when it appears in print.

The line-breaking problem is informally called the problem of 'justification', since it is the 'J' of 'H & J' (hyphenation and justification) in today's commercial composition and word-processing systems. However, this tends to be a misnomer, because printers

[*] This research was supported in part by the National Science Foundation under grants IST-7921977 and MCS-7723738; by Office of Naval Research grant N00014-76-C-0330; by the IBM Corporation; and by the Addison-Wesley Publishing Company. 'TEX' and 'Tau Epsilon Chi' are registered trademarks of the American Mathematical Society.

0038-0644/81/111119-66 $06.60

*Received 25 December 1980*
*Revised 6 February 1981*

have traditionally used justification to mean the process of taking an individual line of type and adjusting its spacing to produce a desired length. Even when text is being typeset with ragged right margins (therefore 'unjustified'), it needs to be broken into lines of approximately the same size. The job of adjusting spaces so that left and right margins are uniformly straight is comparatively laborious when one must work with metal type, so the task of typesetting a paragraph with last century's technology was conceptually a task of justification; nowadays, however, it is no trick at all for computers to adjust the spacing as desired, so the line-breaking task dominates the work. This shift in relative difficulty probably accounts for the shift in the meaning of 'justification'; we shall use the term 'line breaking' in this paper to emphasize the fact that the central problem of concern here is to find breakpoints.

The traditional way to break lines is analogous to what we ordinarily do when using a typewriter: A bell rings (at least conceptually) when we approach the right margin, and at that time we decide how best to finish off that line, without looking ahead to see where the next line or lines might end. Once the typewriter carriage has been returned to the left margin, we begin afresh without needing to remember anything about the previous text except where the new line starts. Thus, we don't have to keep track of many things at once; such a system is ideally suited to human operation, and it also leads to simple computer programs.

Book printing is different from typing primarily in that the spaces are of variable width. Traditional practice has been to assign a minimum and maximum width to interword spaces, together with a normal width representing the ideal situation. The standard algorithm for line breaking (see, for example, Barnett[1], page 55) then proceeds as follows: Keep appending words to the current line, assuming the normal spacing, until reaching a word that does not fit. Break after this word, if it is possible to do so without compressing the spaces to less than the given minimum; otherwise break before this word, if it is possible to do so without expanding the spaces to more than the given maximum. Otherwise hyphenate the offending word, putting as much of it on the current line as will fit; if no suitable hyphenation points can be found, this may result in a line whose spaces exceed the given maximum.

There is no need to confine computers to such a simple procedure, since the data for an entire paragraph is generally available in the computer's memory. Experience has shown that significant improvements are possible if the computer takes advantage of its opportunity to 'look ahead' at what is coming later in the paragraph, before making a final decision about where any of the lines will be broken. This not only tends to avoid cases where the traditional algorithm has to resort to wide spaces, it also reduces the number of hyphenations necessary. In other words, line breaking decisions provide another example of the desirability of 'late binding' in computer software.

One of the principal reasons for using computers in typesetting is to save money, but at the same time we don't want the output to look cheaper. A properly programmed computer should, in fact, be able to solve the line-breaking problem better than a skilled typesetter could do by hand in a reasonable amount of time (unless we give this person the liberty to change the wording in order to obtain a better fit). For example, Duncan[2] studied the interword spacing of 958 lines that were manually typeset by a "most respectable publishers' printer" that he chose not to identify by name, and he found that nearly 5% of the lines were quite loosely set; the spaces on those lines exceeded 10 units (i.e., $\frac{10}{18}$ of an em), and two of the lines even had spaces exceeding 13 units. We shall see later that a good line-breaking algorithm can do better than this.

Besides the avoidance of hyphens and wide spaces, we can improve on the traditional line-breaking method by keeping the spaces nearly equal to the normal size, so that they rarely approach the minimum or maximum limits. We can also try to avoid rapid changes in the spacing of adjacent lines; we can make special efforts not to hyphenate two lines in a row, and not to hyphenate the second-last line of a paragraph; we can try to control the white space on the final line of the paragraph; and so on. Given any mathematical way to rate the quality of a particular choice of breakpoints, we can ask the computer to find breakpoints that optimize this function.

But how is the computer to solve such a problem efficiently? When a given paragraph has $n$ optional breakpoints, there are $2^n$ ways to break it into lines, and even the fastest conceivable computers could not run through all such possibilities in a reasonable amount of time. In fact, the job of breaking a paragraph as nicely as possible into equal-size lines sounds suspiciously like the infamous bin-packing problem, which is well known to be NP complete.[3] Fortunately, however, each line is to consist of contiguous information from the paragraph, so the line-breaking problem is amenable to the techniques of discrete dynamic programming[4,5]; this means there is a reasonably efficient way to attack it. We shall see that the optimum breakpoints can be found in practice with only about twice as much computation as needed by the traditional algorithm; the new method is sometimes even faster than the old, when we consider the time saved by not needing to hyphenate so often. Furthermore the new algorithm is capable of doing other things like setting a paragraph one line longer or one line shorter, in order to improve the layout of a page.

## FORMULATING THE PROBLEM

Let us now state the line-breaking problem explicitly in mathematical terms. We shall use the basic concepts and terminology of the TEX typesetting system[6], but in simplified form, since the complexities of general typesetting would obscure the main principles of line breaking.

For the purposes of this paper, a *paragraph* is a sequence $x_1x_2 \ldots x_m$ of $m$ items, where each individual item $x_i$ is either a *box* specification, a *glue* specification, or a *penalty* specification.

- A box refers to something that is to be typeset: either a character from some font of type, or a black rectangle such as a horizontal or vertical rule, or something built up from several characters such as an accented letter or a mathematical formula. The contents of a box may be extremely complicated, or they may be extremely simple; the line-breaking algorithm does not peek inside a box to see what it contains, so we may consider the boxes to be sealed and locked. As far as we are concerned, the only relevant thing about a box is its *width*: When item $x_i$ of a paragraph specifies a box, the width of that box is a real number $w_i$ representing the amount of space that the box will occupy on a line. The width of a box may be zero, and in fact it may also be negative, although negative widths must be used with care and understanding according to the precise rules laid down below.

- Glue refers to blank space that can vary its width in specified ways; it is an elastic mortar used between boxes in a typeset line. When item $x_i$ of a paragraph specifies glue, there are three real numbers $(w_i, y_i, z_i)$ of importance to the line-breaking algorithm:

$w_i$ is the 'ideal' or 'normal' width;
$y_i$ is the 'stretchability';
$z_i$ is the 'shrinkability'.

For example, the space between words in a line is often specified by the values $w_i = \frac{1}{3}$em, $y_i = \frac{1}{6}$em, $z_i = \frac{1}{9}$em, where one em is the set size of the type being used (approximately the width of an uppercase 'M' in classical type styles). The actual amount of space occupied by this glue can be adjusted when justifying a line to some desired width; if the normal width is too small, the adjustment is proportional to $y_i$, and if the normal width is too large the adjustment is proportional to $z_i$. The numbers $w_i$, $y_i$, and $z_i$ may be negative, subject to certain natural restrictions explained later; for example, a negative value of $w_i$ indicates a backspace. When $y_i = z_i = 0$, the glue has a fixed width $w_i$. Incidentally, the word 'glue' is perhaps not the best term, because it sounds a bit messy; a word like 'spring' would be better, since metal springs expand or compress to fill up space in essentially the way we want. However, we shall continue to say 'glue', a term used since the early days of TEX (1977), because many people claim to like it. A glob of glue is often called a *skip* by TEX users, and it seems preferable to speak of boxes and skips rather than boxes and springs or boxes and glues. A skip, by any other name, is of course the same abstract concept, embodied by the three values $(w_i, y_i, z_i)$.

- Penalty specifications refer to potential places to end one line of a paragraph and begin another, with a certain 'aesthetic cost' indicating how desirable or undesirable such a breakpoint would be. When item $x_i$ of a paragraph specifies a penalty, there is a number $p_i$ that helps us decide whether or not to end a line at this point, as explained below. Intuitively, a high penalty $p_i$ indicates a relatively poor place to break, while a negative value of $p_i$ stands for a good breaking-off place. The penalty $p_i$ may also be $+\infty$ or $-\infty$, where '$\infty$' denotes a large number that is infinite for practical purposes, although it really is finite; in TEX, any penalty $\geqslant 1000$ is treated as $+\infty$, and any penalty $\leqslant -1000$ is treated as $-\infty$. When $p_i = +\infty$, the break is strictly prohibited; when $p_i = -\infty$, the break is mandatory. Penalty specifications also have widths $w_i$, with the following meaning: If a line break occurs at this place in the paragraph, additional typeset material of width $w_i$ will be added to the line just before the break occurs. For example, a potential place at which a word might be hyphenated would be indicated by letting $p_i$ be the penalty for hyphenating there and letting $w_i$ be the width of the hyphen. Penalty specifications are of two kinds, *flagged* and *unflagged*, denoted by $f_i = 1$ and $f_i = 0$. The line-breaking algorithm we shall discuss tries to avoid having two consecutive breaks at flagged penalties (e.g., two hyphenations in a row).

Thus, box items are specified by one number $w_i$, while glue items have three numbers $(w_i, y_i, z_i)$ and penalty items have three numbers $(w_i, p_i, f_i)$. For simplicity, we shall assume that a paragraph $x_1 \ldots x_m$ is actually specified by six sequences, namely

$t_1 \ldots t_m$, where $t_i$ is the type of item $x_i$, either 'box', 'glue', or 'penalty';
$w_1 \ldots w_m$, where $w_i$ is the width corresponding to $x_i$;

$y_1 \dots y_m$, where $y_i$ is the stretchability corresponding to $x_i$ if $t_i =$ 'glue', otherwise $y_i = 0$;
$z_1 \dots z_m$, where $z_i$ is the shrinkability corresponding to $x_i$ if $t_i =$ 'glue', otherwise $z_i = 0$;
$p_1 \dots p_m$, where $p_i$ is the penalty at $x_i$ if $t_i =$ 'penalty', otherwise $p_i = 0$;
$f_1 \dots f_m$, where $f_i = 1$ if $x_i$ is a flagged penalty, otherwise $f_i = 0$.

Any fixed unit of measure can be used in connection with $w_i$, $y_i$, and $z_i$; TEX uses printers' points, which are slightly less than $\frac{1}{72}$ inch. In this paper we shall specify all widths in terms of *machine units* equal to $\frac{1}{18}$em, assuming a particular size of type, since the widths turn out to be integer multiples of this unit in many cases; the numbers in our examples will be as simple as possible when expressed in terms of machine units.

Perhaps the reader feels this is altogether too much mathematical machinery to deal with something that is quite straightforward. However, each of the concepts defined here must be dealt with somehow when breaking paragraphs into lines, and it is important to give precise rules even for the comparatively simple job of setting straight text. We shall see later that these primitive notions of boxes, glue, and penalties will actually support a surprising variety of other line-breaking applications, so that a careful attention to details will solve many other problems as a free bonus.

For the time being, it will be best to think of the simple application to straight text material such as the typesetting of a paragraph in a newspaper or in a short story, since this will help us internalize the abstract concepts represented by $w_i$, $y_i$, etc. A typesetting system like TEX will put such an actual paragraph into the abstract form we want in the following way:

(1) If the paragraph is to be indented, the first item $x_1$ will be an empty box whose width is the amount of indentation.

(2) Each word of the paragraph becomes a sequence of boxes for the characters of the word, including punctuation marks that belong with that word. The widths $w_i$ are determined by the fonts of type being used. Flagged penalty items are inserted into these words wherever an acceptable hyphenation could be used to divide a word at the end of a line. (Such hyphenation points do not need to be included unless necessary, as we shall see later, but for the moment let us assume that all of the permissible hyphenations have been specified.)

(3) There is glue between words, corresponding to the recommended spacing conventions of the fonts of type in use. The glue might be different in different contexts; for example, TEX will make the glue specifications following punctuation marks slightly different from the normal interword glue.

(4) Explicit hyphens and dashes in the text will be followed by flagged penalty items having width zero. This specifies a permissible line break after a hyphen or a dash. Some style conventions also allow breaks before em-dashes, in which case an unflagged width-zero penalty would precede the dash.

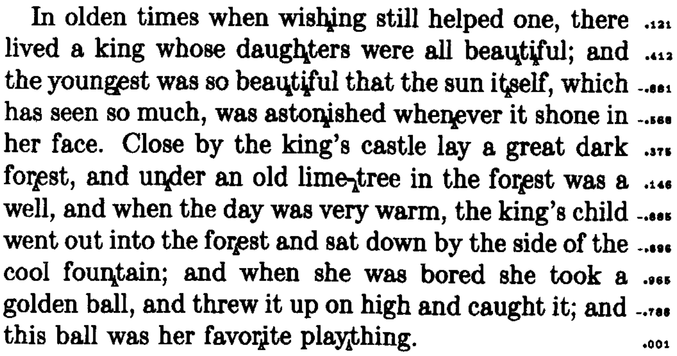
(5) At the very end of a paragraph, two items are appended so that the final line will be treated properly. First comes a glue item $x_{m-1}$ that specifies the white space allowable at the right of the last line; then comes a penalty item $x_m$ with

$p_m = -\infty$ to force a break at the paragraph end. TEX ordinarily uses a 'finishing glue' with $w_{m-1} = 0$, $y_{m-1} = \infty$ (actually 100000 points, which is finite but large enough to behave like $\infty$), and $z_{m-1} = 0$; thus the normal space at the end of a paragraph is zero but it can stretch a great deal. The net effect is that the other spaces on the final line will shrink, if that line exceeds the desired measure; otherwise the other spaces will remain essentially at their normal value (because the finishing glue will do all the stretching necessary to fill up the end of the line). More subtle choices of the finishing glue $x_{m-1}$ will be discussed later.

For example, let's consider the paragraph of Figure 1, which is taken from Grimm's Fairy Tales.[7] The five rules above convert the text into the following sequence of 601 items:

| | | | |
|---|---|---|---|
| $x_1$ = empty box for indentation | $w_1 = 18$ | | |
| $x_2$ = box for 'I' | $w_2 = 6$ | | |
| $x_3$ = box for 'n' | $w_3 = 10$ | | |
| $x_4$ = glue for interword space | $w_4 = 6$, | $y_4 = 3$, | $z_4 = 2$ |
| $x_5$ = box for 'o' | $w_5 = 9$ | | |
| ........ | | | |
| $x_{309}$ = box for 'l' | $w_{309} = 5$ | | |
| $x_{310}$ = box for 'i' | $w_{310} = 5$ | | |
| $x_{311}$ = box for 'm' | $w_{311} = 15$ | | |
| $x_{312}$ = box for 'e' | $w_{312} = 8$ | | |
| $x_{313}$ = box for '-' | $w_{313} = 6$ | | |
| $x_{314}$ = penalty for explicit hyphen | $w_{314} = 0$, | $p_{314} = 50$, | $f_{314} = 1$ |
| $x_{315}$ = box for 't' | $w_{315} = 7$ | | |
| ........ | | | |
| $x_{592}$ = box for 'y' | $w_{592} = 10$ | | |
| $x_{593}$ = penalty for optional hyphen | $w_{593} = 6$, | $p_{593} = 50$, | $f_{593} = 1$ |
| $x_{594}$ = box for 't' | $w_{594} = 7$ | | |
| $x_{595}$ = box for 'h' | $w_{595} = 10$ | | |
| $x_{596}$ = box for 'i' | $w_{596} = 5$ | | |
| $x_{597}$ = box for 'n' | $w_{597} = 10$ | | |
| $x_{598}$ = box for 'g' | $w_{598} = 9$ | | |
| $x_{599}$ = box for '.' | $w_{599} = 5$ | | |
| $x_{600}$ = finishing glue | $w_{600} = 0$, | $y_{600} = \infty$, | $z_{600} = 0$ |
| $x_{601}$ = forced break | $w_{601} = 0$, | $p_{601} = -\infty$, | $f_{601} = 1$ |

In this particular example, a penalty of 50 has been assessed for every line that ends with a hyphen.



In olden times when wishing still helped one, there .121
lived a king whose daughters were all beautiful; and .412
the youngest was so beautiful that the sun itself, which -.881
has seen so much, was astonished whenever it shone in -.568
her face. Close by the king's castle lay a great dark .375
forest, and under an old lime-tree in the forest was a .146
well, and when the day was very warm, the king's child -.885
went out into the forest and sat down by the side of the -.896
cool fountain; and when she was bored she took a .965
golden ball, and threw it up on high and caught it; and -.788
this ball was her favorite plaything. .001

*Figure 1. An example paragraph that has been typeset by the 'first-fit' method. Small triangles show permissible places to divide words with hyphens; the adjustment ratio for spaces appears at the right of each line.*

Optional hyphenation points have been indicated with triangles in Figure 1. It is considered bad form to insert a hyphen unless at least two letters precede it and three follow it; furthermore the syllable following a hyphen shouldn't have a silent 'e', so we do not admit a hyphenation like 'sylla-ble'. Smooth reading also means that the word fragment preceding a hyphen should be long enough that it can be pronounced correctly, before the reader sees the completion of the word on the next line; thus, a hyphenation like 'pro-cess' would be disturbing. This pronunciation rule accounts for the fact that the second-last word of Figure 1 does not admit the potential hyphenation 'fa-vorite', since the fragment 'fa-' might well be the beginning of 'fa-ther' which is pronounced quite differently.

The choice of proper hyphenation points is an important but difficult subject that is beyond the scope of this paper. We shall not mention it further except to assume that (a) such potential breakpoints are available to our line-breaking algorithm when needed; (b) we prefer not to hyphenate when there is a way to avoid it without seriously messing up the spacing.

The rules for breaking a paragraph into lines should be intuitively clear from this example, but it is important to state them explicitly. We shall assume that every paragraph ends with a forced break item $x_m$ (penalty $-\infty$). A *legal breakpoint* in a paragraph is a number $b$ such that either (i) $x_b$ is a penalty item with $p_b<\infty$, or (ii) $x_b$ is a glue item and $x_{b-1}$ is a box item. In other words, one can break at a penalty, provided that the penalty isn't $\infty$, or at glue, provided that the glue immediately follows a box. These two cases are the only acceptable breakpoints. Note, for example, that several glue items may appear consecutively, but it is possible to break only at the first of them, and only if this one does not immediately follow a penalty item. A penalty of $\infty$ can be inserted before glue to make it unbreakable.

The job of line breaking consists of choosing legal breakpoints $b_1<\cdots<b_k$, which specify the ends of $k$ lines into which the paragraph will be broken. Each penalty item $x_i$ whose penalty $p_i$ is $-\infty$ must be included among these breakpoints; thus, the final breakpoint $b_k$ must be equal to $m$. For convenience we let $b_0=0$, and we define indices $a_1<\cdots<a_k$ to mark the beginning of the lines, as follows: The value of $a_j$ is the smallest integer $i$ between $b_{j-1}$ and $b_j$ such that $x_i$ is a box item or a penalty item with $p_i=-\infty$; if none of the $x_i$ in the range $b_{j-1}<i<b_j$ meet this criterion, we let $a_j=b_j$. Then the $j$th line consists of all items $x_i$ for $a_j\leqslant i<b_j$, plus item $x_{b_j}$ if it is a penalty item. In other words we get the lines of the broken paragraph by cutting it into pieces at the chosen breakpoints, then removing glue and penalty items at the beginning of each resulting line.

## DESIRABILITY CRITERIA

According to this definition of line breaking, there are $2^n$ ways to break a paragraph into lines, if the paragraph has $n$ legal breakpoints that aren't forced. For example, there are 129 legal breakpoints in the paragraph of Figure 1, not counting $x_{601}$, so it can be broken into lines in $2^{129}$ ways, a number that exceeds $10^{38}$. But of course most of these choices are absurd, and we need to specify some criteria to separate acceptable choices from the ridiculous ones. For this purpose we need to know (a) the desired lengths of lines, and (b) the lengths of lines corresponding to each choice of breakpoints, including the amount of stretchability and shrinkability that is present. Then we can compare the desired lengths to the lengths actually obtained.

We shall assume that a list of desired lengths $l_1, l_2, l_3, \ldots$ is given; normally these are all the same, but in general we might want lines of different lengths, as when fitting text around an illustration. The actual length $L_j$ of the $j$th line, after breakpoints have been chosen as above, is computed in the following obvious way: We add together the widths $w_i$ of all the box and glue items in the range $a_j \leqslant i < b_j$, and we add $w_{b_j}$ to this total if $x_{b_j}$ is a penalty item. The $j$th line also has a total stretchability $Y_j$ and total shrinkability $Z_j$, obtained by summing all of the $y_i$ and $z_i$ for glue items in the range $a_j \leqslant i < b_j$. Now we can compare the actual length $L_j$ to the desired length $l_j$ by seeing if there is enough stretchability or shrinkability to change $L_j$ into $l_j$; we define the *adjustment ratio* $r_j$ of the $j$th line as follows:

If $L_j = l_j$ (a perfect fit), let $r_j = 0$.

If $L_j < l_j$ (a short line), let $r_j = (l_j - L_j)/Y_j$, assuming that $Y_j > 0$; the value of $r_j$ is undefined if $Y_j \leqslant 0$ in this case.

If $L_j > l_j$ (a long line), let $r_j = (l_j - L_j)/Z_j$, assuming that $Z_j > 0$; the value of $r_j$ is undefined if $Z_j \leqslant 0$ in this case.

Thus, for example, $r_j = \frac{1}{3}$ if the total stretchability of line $j$ is three times what would be needed to expand the glue so that the line length would change from $L_j$ to $l_j$.

According to this definition of adjustment ratios, the $j$th line can be justified by letting the width of all glue items $x_i$ on that line be

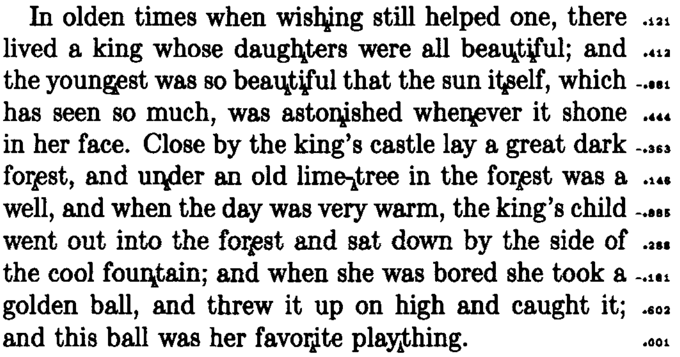
$$\begin{aligned} &w_i + r_j y_i, \quad \text{if } r_j \geqslant 0;\\ &w_i + r_j z_i, \quad \text{if } r_j < 0; \end{aligned}$$

For if we add up the total width of that line after such adjustments are made, we get either $L_j + r_j Y_j = l_j$ or $L_j + r_j Z_j = l_j$, depending on the sign of $r_j$. This distributes the necessary stretching or shrinking by amounts proportional to the individual glue components $y_i$ or $z_i$, as desired.

For example, the small numbers at the right of the individual lines in Figure 1 show the values of $r_j$ in those lines. A negative ratio like $-.881$ in the third line means that the spaces in that line are narrower than their ideal size; a fairly large positive ratio like .965 in the third-last line indicates a very 'loose' fit.

Although there are $2^{129}$ ways to break the paragraph of Figure 1 into lines, it turns out that only 49 of these will result in breaks whose adjustment ratios $r_j$ do not exceed 1 in absolute value; this means that the spaces between words after justification will lie between $w_i - z_i$ and $w_i + y_i$. Furthermore, only 30 of these 49 ways to make 'nice' breaks will do so without introducing hyphens. One of these ways is obtained by moving 'the' from the eighth line down to the ninth.

Our main goal is to find a way to avoid choosing any breakpoints that lead to lines in which the words are spaced very far apart, or in which they are very close together, because such lines are distracting and harder to read. We might therefore say that the line-breaking problem is to find breaks such that $|r_j| \leqslant 1$ in each line, with the minimum number of hyphenations subject to this condition. Such an approach was taken by Duncan et al.[8] in the early 1960s, and they obtained fairly good results. However, this criterion depends only on the values $w_i - z_i$ and $w_i + y_i$, not $w_i$ itself, so it does not use all the degrees of freedom present in our data. Furthermore, such stringent conditions may not be possible to achieve; for example, if each line of our example were to be 418 units wide, instead of the present



In olden times when wishing still helped one, there .121
lived a king whose daughters were all beautiful; and .412
the youngest was so beautiful that the sun itself, which -.881
has seen so much, was astonished whenever it shone .444
in her face. Close by the king's castle lay a great dark -.363
forest, and under an old lime-tree in the forest was a .146
well, and when the day was very warm, the king's child -.885
went out into the forest and sat down by the side of .288
the cool fountain; and when she was bored she took a -.161
golden ball, and threw it up on high and caught it; .602
and this ball was her favorite plaything. .001

*Figure 2. The paragraph of Figure 1 when the 'best-fit' method has been used to find successive breakpoints.*

width of 421 units, there would be no way to set the text of Figure 1 without having at least one very tight line ($r_j < -1$) or at least one very loose line ($r_j > 1$).

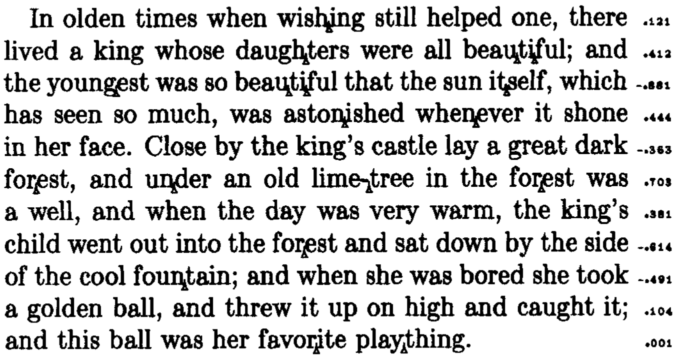
We can do a better job of line breaking if we deal with a continuously varying criterion of quality, not simply the yes/no tests of whether $|r_j| \leqslant 1$ or not. Let us therefore give a quantitative evaluation of the *badness* of the $j$th line by finding a formula that is nearly zero when $|r_j|$ is small but grows rapidly when $|r_j|$ takes values exceeding 1. Experience with TEX has shown that good results are obtained if we define the badness of line $j$ as follows:

$$\beta_j = \begin{cases} \infty, & \text{if } r_j \text{ is undefined or } r_j < -1; \\ 100|r_j|^3, & \text{otherwise.} \end{cases}$$

Thus, for example, the individual lines of Figure 1 have badness ratings that are approximately equal to 0, 7, 68, 18, 5, 0, 69, 72, 90, 49, 0, respectively. Note that a line is considered to be 'infinitely bad' if $r_j < -1$; this means that glue will never be shrunk to less than $w_i - z_i$. However, values of $r_j > 1$ are only finitely bad, so they will be permitted if there is no better alternative.

A slight improvement over the method used to produce Figure 1 leads to Figure 2. Once again each line has been broken without looking ahead to the end of the paragraph and without going back to reconsider previous choices, but this time each break was chosen so as to minimize the 'badness plus penalty' of that line. In other words, when choosing between alternative ways to end the $j$th line, given the ending of the previous line, we obtain Figure 2 if we take the minimum possible value of $\beta_j + \pi_j$; here $\beta_j$ is the badness as defined above, and $\pi_j$ is the amount of penalty $p_{b_j}$ if the $j$th line ends at a penalty item, otherwise $\pi_j = 0$. Figure 2 improves on Figure 1 by moving words down from lines 4, 8, and 10 to the next line.

The method that produces Figure 1 might be called the 'first-fit' algorithm, and the corresponding method for Figure 2 might be called the 'best-fit' algorithm. We have seen that best-fit is superior to first-fit in this particular case, but other paragraphs can be contrived in which first-fit finds a better solution; so a single example is not sufficient to decide which method is preferable. In order to make an unbiased comparison of the methods, we need to get some statistics on their 'typical' behavior. Therefore 300 experiments were performed, using the text of Figures 1 and 2, with line widths ranging from 350 to 649 in unit steps; although the text for each experiment was the same, the varying line widths made the problems quite different, since line-breaking algorithms are quite sensitive to slight changes in the measurements. The 'tightest'



In olden times when wishing still helped one, there .121
lived a king whose daughters were all beautiful; and .412
the youngest was so beautiful that the sun itself, which -.881
has seen so much, was astonished whenever it shone .444
in her face. Close by the king's castle lay a great dark -.363
forest, and under an old lime-tree in the forest was .703
a well, and when the day was very warm, the king's .381
child went out into the forest and sat down by the side -.614
of the cool fountain; and when she was bored she took -.491
a golden ball, and threw it up on high and caught it; .104
and this ball was her favorite plaything. .001

*Figure 3. This is the 'best possible' way to break the lines in the paragraph of Figures 1 and 2, in the sense of fewest total 'demerits' as defined in the text.*

and 'loosest' lines in each resulting paragraph were recorded, as well as the number of hyphens introduced, and the comparisons came out as follows:

| | min $r_j$ | max $r_j$ | hyphens |
|---|---|---|---|
| first-fit < best-fit | 69% | 35% | 12% |
| first-fit = best-fit | 26% | 50% | 77% |
| first-fit > best-fit | 5% | 15% | 11% |

Thus, in 69% of the cases, the minimum adjustment ratio $r_j$ in the lines typeset by first-fit was less than the corresponding value obtained by best-fit; the maximum adjustment ratio in the first-fit lines was less than the maximum for best-fit about 35% of the time; etc. We can summarize this data by saying that the first-fit method usually typesets at least one line that is tighter than the tightest line set by best-fit, and it also usually produces a line that is as loose or looser than the loosest line of best-fit. The number of hyphens is about the same for both methods, although best-fit would produce fewer if the penalty for hyphenation were increased. A more detailed study of the experimental data shows that the superiority of best-fit is especially pronounced in the cases where the lines are rather narrow.

We can actually do better than both of these methods by finding an 'optimum' way to choose the breakpoints. For example, Figure 3 shows how to improve on both Figures 1 and 2 by making line 6 a bit looser, thereby avoiding a rather tight 7th line and a fairly loose 10th line. This pattern of breakpoints was found by an algorithm that will be discussed in detail below. It is globally optimum in the sense of having fewest total 'demerits' over all choices of breakpoints, where the demerits assessed for the $j$th line are computed by the formula

$$\delta_j = \begin{cases} (1+\beta_j+\pi_j)^2+\alpha_j, & \text{if } \pi_j \geqslant 0; \\ (1+\beta_j)^2-\pi_j^2+\alpha_j, & \text{if } -\infty<\pi_j<0; \\ (1+\beta_j)^2+\alpha_j, & \text{if } \pi_j=-\infty. \end{cases}$$

Here $\beta_j$ and $\pi_j$ are the badness rating and the penalty, as before; and $\alpha_j$ is zero unless both line $j$ and the previous line ended on flagged penalty items, in which case $\alpha_j$ is the additional penalty assessed for consecutive hyphenated lines (e.g., 3000). We shall say that we have found the best choice of breakpoints if we have minimized the sum of $\delta_j$ over all lines $j$.

The above formula for $\delta_j$ is quite arbitrary, like our formula for $\beta_j$, but it works well in practice because it has the following desirable properties: (a) Minimizing the sum of squares of badnesses not only tends to minimize the maximum badness per line, it also provides secondary optimization; for example, when one particularly bad line is inevitable, the other line breaks will also be optimized. (b) The demerit function $\delta_j$ increases as $\pi_j$ increases, except in the case $\pi_j = -\infty$ when we don't need to consider the penalty because such breaks are forced. (c) Adding 1 to $\beta_j$ instead of using the badness $\beta_j$ by itself will minimize the total number of lines in cases where there are breaks with approximately zero badness.

For example, the following table shows the respective demerits charged to the individual lines of the paragraphs in Figures 1, 2, and 3:

| First fit | Best fit | Optimum fit |
|---:|---:|---:|
| 1 | 1 | 1 |
| 64 | 64 | 64 |
| 4803 | 4803 | 4803 |
| 374 | 96 | 96 |
| 39 | 33 | 33 |
| 2 | 2 | 1274 |
| 4958 | 4958 | 43 |
| 5313 | 11 | 581 |
| 8252 | 3 | 166 |
| 2497 | 519 | 1 |
| 1 | 1 | 1 |
| 26304 | 10491 | 7063 |

In the first-fit and best-fit methods, each line is likely to come out about as badly as any other; but the optimum-fit method tends to have its bad cases near the beginning, since there is less flexibility in the opening lines.

Figure 4 on the following page shows another comparison of the same three methods on the same text, this time with a line width of 500 units. Note that the optimum algorithm finds a solution that does not hyphenate any words, because of its ability to 'look ahead'; the other two methods, which proceed one line at a time, miss this solution because they do not know that a slightly worse first line leads in this case to fewer problems later on. The demerits per line in Figure 4 are

| First fit | Best fit | Optimum fit |
|---:|---:|---:|
| 1734 | 1734 | 2357 |
| 4692 | 4692 | 6 |
| 3440 | 3440 | 938 |
| 3066 | 9 | 212 |
| 3 | 1 | 1 |
| 1 | 22 | 2 |
| 276 | 210 | 27 |
| 5 | 24 | 10 |
| 1 | 10 | 476 |
|  | 1 | 1 |
| 13218 | 10143 | 4030 |

In this example the 3440 demerits on the third line for 'first fit' and 'best fit' are primarily due to the penalty of 50 for an inserted hyphen.

(a) In olden times when wishing still helped one, there lived a king −.741
whose daughters were all beautiful; and the youngest was so .877
beautiful that the sun itself, which has seen so much, was aston- −.425
ished whenever it shone in her face. Close by the king's castle lay −.816
a great dark forest, and under an old lime-tree in the forest was −.191
a well, and when the day was very warm, the king's child went .107
out into the forest and sat down by the side of the cool fountain; −.538
and when she was bored she took a golden ball, and threw it up −.234
on high and caught it; and this ball was her favorite plaything. .000

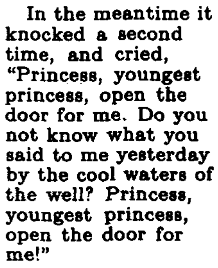
(b) In olden times when wishing still helped one, there lived a king −.741
whose daughters were all beautiful; and the youngest was so .877
beautiful that the sun itself, which has seen so much, was aston- −.425
ished whenever it shone in her face. Close by the king's castle .269
lay a great dark forest, and under an old lime-tree in the forest .027
was a well, and when the day was very warm, the king's child .333
went out into the forest and sat down by the side of the cool .513
fountain; and when she was bored she took a golden ball, and .340
threw it up on high and caught it; and this ball was her favorite −.257
plaything. .002

(c) In olden times when wishing still helped one, there lived a .780
king whose daughters were all beautiful; and the youngest was .246
so beautiful that the sun itself, which has seen so much, was .667
astonished whenever it shone in her face. Close by the king's .514
castle lay a great dark forest, and under an old lime-tree in the .027
forest was a well, and when the day was very warm, the king's .173
child went out into the forest and sat down by the side of the .346
cool fountain; and when she was bored she took a golden ball, .275
and threw it up on high and caught it; and this ball was her .593
favorite plaything. .002

*Figure 4. A somewhat wider setting of the same sample paragraph, by* (*a*) *the first-fit method,* (*b*) *the best-fit method, and* (*c*) *the optimum-fit method. Notice the tight line followed by a loose line at the beginning of examples* (*a*) *and* (*b*), *while no hyphenation was needed in* (*c*); *on the other hand,* (*a*) *is one line shorter than* (*b*) *and* (*c*).

The first-fit method found a way to set the paragraph of Figure 4 in only nine lines, while the optimum-fit method yields ten. Publishers who prefer to save a little paper, as long as the line breaks are fairly decent, might therefore prefer the first-fit solution in spite of all its demerits. There are various ways to modify the specifications so that the optimum-fit method will give more preference to short solutions; for example, the stretchability of the glue on the final line could be decreased from its present huge size to about the width of the line, thereby making the optimum algorithm prefer final lines that are nearly full. We could also replace the constant '1' in the definition of demerits $\delta_j$ by a variable parameter. The algorithm we shall describe below can in fact be set up to produce the optimum solution having the minimum number of lines.

The text in these examples is quite straightforward, and we have been setting type in reasonably wide columns; thus we have not been considering especially difficult or



In the meantime it
knocked a second
time, and cried,
"Princess, youngest
princess, open the
door for me. Do you
not know what you
said to me yesterday
by the cool waters of
the well? Princess,
youngest princess,
open the door for
me!"

*Figure 5. Here the best-fit method is unable to find a satisfactory way to break the lines, with respect to justified setting, because the columns are so narrow. For example, the third line contains only two spaces, and the third-last line only one; these spaces would have to stretch considerably if the lines were justified. The first line of this paragraph also illustrates the 'sticking-out' problem that arises in unjustified settings.*

unusual line-breaking problems. Yet we have seen that an optimizing algorithm can produce noticeably better results even in such routine cases. The improved algorithm will clearly be of significant value in more difficult situations, for example when mathematical formulas are embedded in the text, or when the lines must be narrow as in a newspaper.

Anyone who is curious about the fate of the beautiful princess mentioned in Figures 1 through 4 can find the answer in Figure 6, which presents the whole story. The columns in this example are unusually narrow, allowing only about 21 or 22 characters per line; a width of about 35 characters is normal for newspapers, and magazines often use columns about twice as wide as those in Figure 6. The line-at-a-time algorithms cannot cope satisfactorily with such stringent restrictions, but Figure 6 shows that the optimizing algorithm is able to break the text into reasonably equal lines.

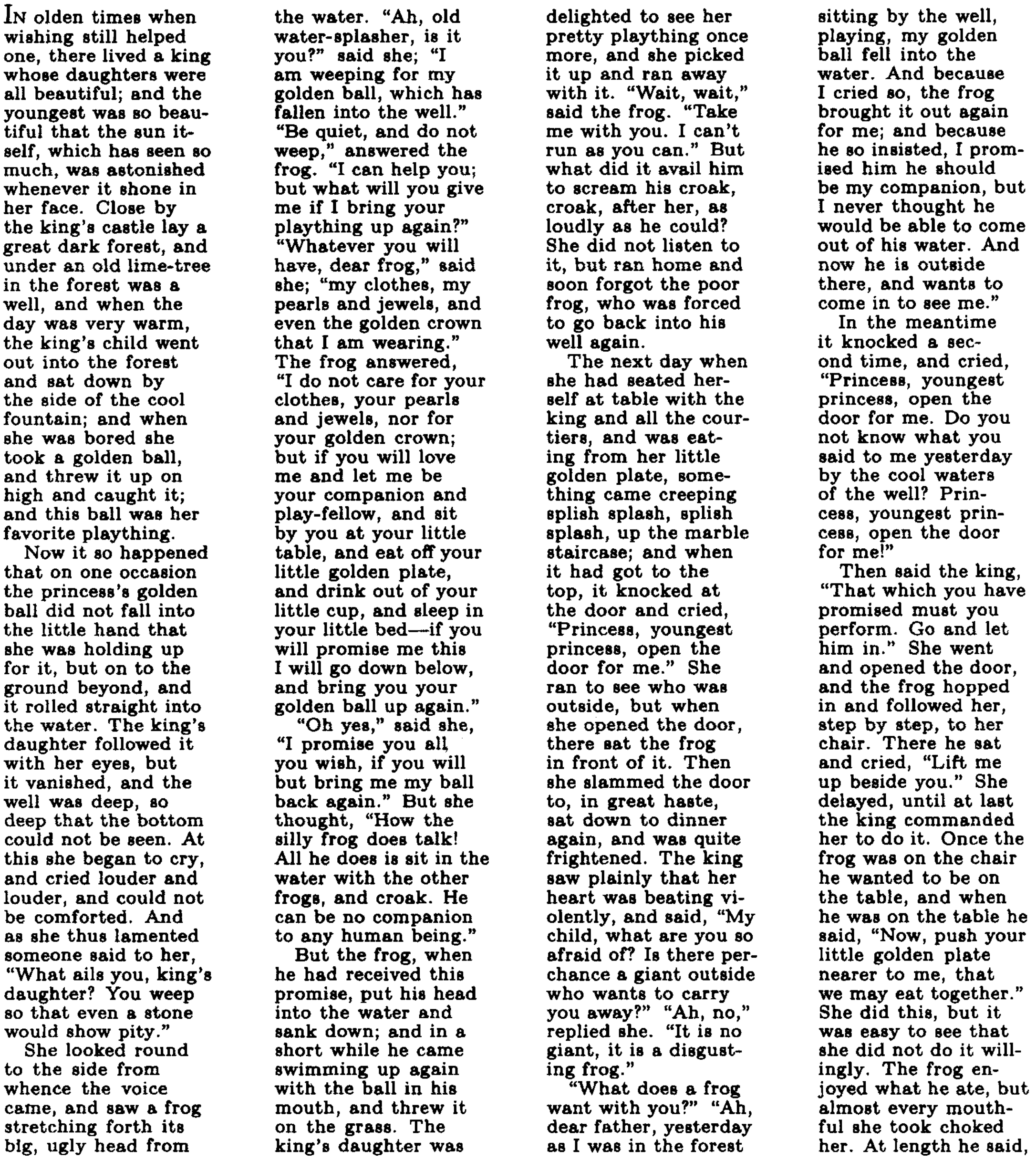
Incidentally, our line-breaking criteria have been developed with justified text in mind; but the algorithm has been used in Figure 6 to produce ragged right margins. Another criterion of badness, which is based solely on the difference between the desired length $l_j$ and the actual length $L_j$, should actually be used in order to get the best breakpoints for ragged-right typesetting, and the space between words should be allowed to stretch but not to shrink so that $L_j$ never exceeds $l_j$. Furthermore, ragged-right typesetting should not allow words to 'stick out', i.e., to begin to the right of where the following line ends (see the word 'it' in Figure 5). Thus, it turns out that an algorithm intended for high quality line breaking in ragged-right formats is actually a little bit harder to write than one for justified text, contrary to the prevailing opinion that justification is more difficult. On the other hand, Figure 6 indicates that an algorithm designed for justification usually can be tuned to produce adequate breakpoints when justification is suppressed.

The difficulties of setting narrow columns are illustrated in an interesting way by the pattern of words

"Now, push your little golden plate nearer ..."

that appears in the third-last paragraph of Figure 6. We don't want to hyphenate any of these words, for reasons stated earlier; and it turns out that all of the four-word sequences containing the word 'little', namely

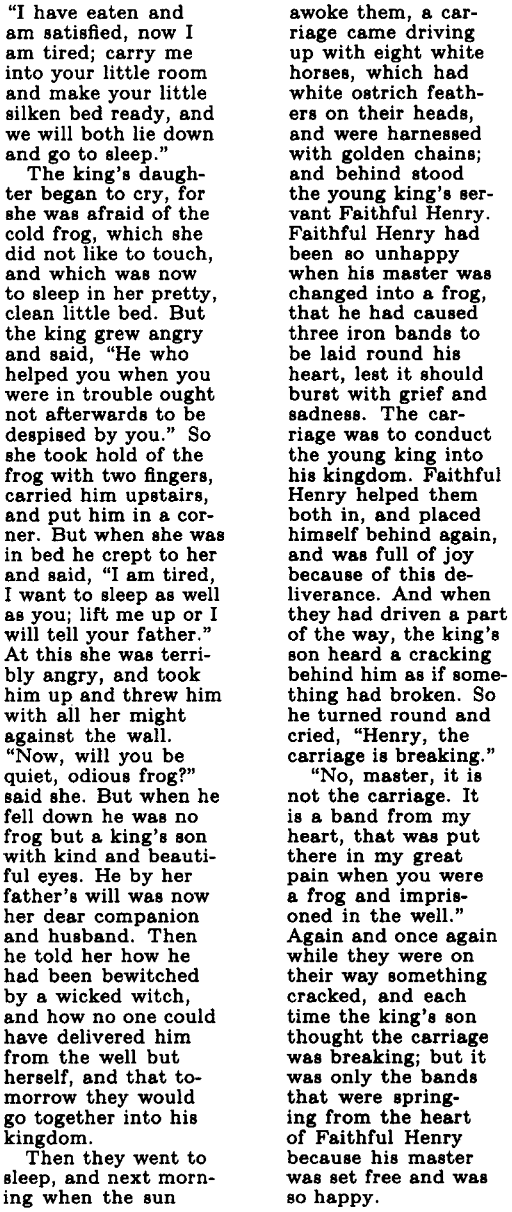
"Now, push your little
push your little golden
your little golden plate
little golden plate nearer



IN olden times when<br>wishing still helped<br>one, there lived a king<br>whose daughters were<br>all beautiful; and the<br>youngest was so beau-<br>tiful that the sun it-<br>self, which has seen so<br>much, was astonished<br>whenever it shone in<br>her face. Close by<br>the king's castle lay a<br>great dark forest, and<br>under an old lime-tree<br>in the forest was a<br>well, and when the<br>day was very warm,<br>the king's child went<br>out into the forest<br>and sat down by<br>the side of the cool<br>fountain; and when<br>she was bored she<br>took a golden ball,<br>and threw it up on<br>high and caught it;<br>and this ball was her<br>favorite plaything.<br>  Now it so happened<br>that on one occasion<br>the princess's golden<br>ball did not fall into<br>the little hand that<br>she was holding up<br>for it, but on to the<br>ground beyond, and<br>it rolled straight into<br>the water. The king's<br>daughter followed it<br>with her eyes, but<br>it vanished, and the<br>well was deep, so<br>deep that the bottom<br>could not be seen. At<br>this she began to cry,<br>and cried louder and<br>louder, and could not<br>be comforted. And<br>as she thus lamented<br>someone said to her,<br>"What ails you, king's<br>daughter? You weep<br>so that even a stone<br>would show pity."<br>  She looked round<br>to the side from<br>whence the voice<br>came, and saw a frog<br>stretching forth its<br>big, ugly head from<br>the water. "Ah, old<br>water-splasher, is it<br>you?" said she; "I<br>am weeping for my<br>golden ball, which has<br>fallen into the well."<br>"Be quiet, and do not<br>weep," answered the<br>frog. "I can help you;<br>but what will you give<br>me if I bring your<br>plaything up again?"<br>"Whatever you will<br>have, dear frog," said<br>she; "my clothes, my<br>pearls and jewels, and<br>even the golden crown<br>that I am wearing."<br>The frog answered,<br>"I do not care for your<br>clothes, your pearls<br>and jewels, nor for<br>your golden crown;<br>but if you will love<br>me and let me be<br>your companion and<br>play-fellow, and sit<br>by you at your little<br>table, and eat off your<br>little golden plate,<br>and drink out of your<br>little cup, and sleep in<br>your little bed—if you<br>will promise me this<br>I will go down below,<br>and bring you your<br>golden ball up again."<br>  "Oh yes," said she,<br>"I promise you all<br>you wish, if you will<br>but bring me my ball<br>back again." But she<br>thought, "How the<br>silly frog does talk!<br>All he does is sit in the<br>water with the other<br>frogs, and croak. He<br>can be no companion<br>to any human being."<br>  But the frog, when<br>he had received this<br>promise, put his head<br>into the water and<br>sank down; and in a<br>short while he came<br>swimming up again<br>with the ball in his<br>mouth, and threw it<br>on the grass. The<br>king's daughter was<br>delighted to see her<br>pretty plaything once<br>more, and she picked<br>it up and ran away<br>with it. "Wait, wait,"<br>said the frog. "Take<br>me with you. I can't<br>run as you can." But<br>what did it avail him<br>to scream his croak,<br>croak, after her, as<br>loudly as he could?<br>She did not listen to<br>it, but ran home and<br>soon forgot the poor<br>frog, who was forced<br>to go back into his<br>well again.<br>  The next day when<br>she had seated her-<br>self at table with the<br>king and all the cour-<br>tiers, and was eat-<br>ing from her little<br>golden plate, some-<br>thing came creeping<br>splish splash, splish<br>splash, up the marble<br>staircase; and when<br>it had got to the<br>top, it knocked at<br>the door and cried,<br>"Princess, youngest<br>princess, open the<br>door for me." She<br>ran to see who was<br>outside, but when<br>she opened the door,<br>there sat the frog<br>in front of it. Then<br>she slammed the door<br>to, in great haste,<br>sat down to dinner<br>again, and was quite<br>frightened. The king<br>saw plainly that her<br>heart was beating vi-<br>olently, and said, "My<br>child, what are you so<br>afraid of? Is there per-<br>chance a giant outside<br>who wants to carry<br>you away?" "Ah, no,"<br>replied she. "It is no<br>giant, it is a disgust-<br>ing frog."<br>  "What does a frog<br>want with you?" "Ah,<br>dear father, yesterday<br>as I was in the forest<br>sitting by the well,<br>playing, my golden<br>ball fell into the<br>water. And because<br>I cried so, the frog<br>brought it out again<br>for me; and because<br>he so insisted, I prom-<br>ised him he should<br>be my companion, but<br>I never thought he<br>would be able to come<br>out of his water. And<br>now he is outside<br>there, and wants to<br>come in to see me."<br>  In the meantime<br>it knocked a sec-<br>ond time, and cried,<br>"Princess, youngest<br>princess, open the<br>door for me. Do you<br>not know what you<br>said to me yesterday<br>by the cool waters<br>of the well? Prin-<br>cess, youngest prin-<br>cess, open the door<br>for me!"<br>  Then said the king,<br>"That which you have<br>promised must you<br>perform. Go and let<br>him in." She went<br>and opened the door,<br>and the frog hopped<br>in and followed her,<br>step by step, to her<br>chair. There he sat<br>and cried, "Lift me<br>up beside you." She<br>delayed, until at last<br>the king commanded<br>her to do it. Once the<br>frog was on the chair<br>he wanted to be on<br>the table, and when<br>he was on the table he<br>said, "Now, push your<br>little golden plate<br>nearer to me, that<br>we may eat together."<br>She did this, but it<br>was easy to see that<br>she did not do it will-<br>ingly. The frog en-<br>joyed what he ate, but<br>almost every mouth-<br>ful she took choked<br>her. At length he said,

are too long to fit in one line. Therefore the word 'little' will have to appear in a line that contains only three words and two spaces, no matter what text precedes this particular sequence.

The final paragraphs of the story present other difficulties, some of which involve complex interactions spanning many lines of the text, making it impossible to find breakpoints that would avoid occasional wide spacing if the text were justified. Figure 7 shows what happens when a portion of Figure 6 is, in fact, justified; this is the most difficult part of the entire story, in which one of the lines in the optimum solution is



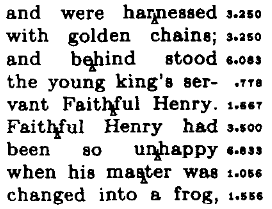
"I have eaten and
am satisfied, now I
am tired; carry me
into your little room
and make your little
silken bed ready, and
we will both lie down
and go to sleep."
The king's daugh-
ter began to cry, for
she was afraid of the
cold frog, which she
did not like to touch,
and which was now
to sleep in her pretty,
clean little bed. But
the king grew angry
and said, "He who
helped you when you
were in trouble ought
not afterwards to be
despised by you." So
she took hold of the
frog with two fingers,
carried him upstairs,
and put him in a cor-
ner. But when she was
in bed he crept to her
and said, "I am tired,
I want to sleep as well
as you; lift me up or I
will tell your father."
At this she was terri-
bly angry, and took
him up and threw him
with all her might
against the wall.
"Now, will you be
quiet, odious frog?"
said she. But when he
fell down he was no
frog but a king's son
with kind and beauti-
ful eyes. He by her
father's will was now
her dear companion
and husband. Then
he told her how he
had been bewitched
by a wicked witch,
and how no one could
have delivered him
from the well but
herself, and that to-
morrow they would
go together into his
kingdom.
Then they went to
sleep, and next morn-
ing when the sun
awoke them, a car-
riage came driving
up with eight white
horses, which had
white ostrich feath-
ers on their heads,
and were harnessed
with golden chains;
and behind stood
the young king's ser-
vant Faithful Henry.
Faithful Henry had
been so unhappy
when his master was
changed into a frog,
that he had caused
three iron bands to
be laid round his
heart, lest it should
burst with grief and
sadness. The car-
riage was to conduct
the young king into
his kingdom. Faithful
Henry helped them
both in, and placed
himself behind again,
and was full of joy
because of this de-
liverance. And when
they had driven a part
of the way, the king's
son heard a cracking
behind him as if some-
thing had broken. So
he turned round and
cried, "Henry, the
carriage is breaking."
"No, master, it is
not the carriage. It
is a band from my
heart, that was put
there in my great
pain when you were
a frog and impris-
oned in the well."
Again and once again
while they were on
their way something
cracked, and each
time the king's son
thought the carriage
was breaking; but it
was only the bands
that were spring-
ing from the heart
of Faithful Henry
because his master
was set free and was
so happy.

*Figure 6. The tale of the Frog King, typeset with quite narrow lines and with 'ragged right' margins. The breakpoints were optimally chosen under the assumption that the lines would be justified; a somewhat different criterion of optimality would have been more appropriate for unjustified setting, yet the lines did turn out to be of approximately equal width. Quite a few hyphenations were found to be desirable, since this increases the number of spaces per line and aids justification, even though the penalty for hyphenation was increased from 50 to 5000 in this example.*

forced to stretch by the enormous factor 6.833. The only way to typeset that paragraph without such wide spaces is to leave it unjustified (unless, of course, we change the problem by altering the text or the line width or the minimum size of spaces).

## FURTHER APPLICATIONS

Before we discuss the details of an optimizing algorithm, it is worthwhile to consider more fully how the basic primitives of boxes, glue, and penalties allow us to solve a



```
and were harnessed 3.250
with golden chains; 3.250
and behind stood 6.083
the young king's ser-  .778
vant Faithful Henry. 1.667
Faithful Henry had 3.500
been so unhappy 6.833
when his master was 1.056
changed into a frog, 1.556
```

*Figure 7. This portion of the story in Figure 6 is the most difficult to handle, when we try to justify the second-last paragraph using such narrow columns; even the optimum breakpoints result in wide spaces.*

wide variety of typesetting problems. Some of these applications are straightforward extensions of the simple ideas used in Figures 1 to 4, while others seem at first to be quite unrelated to the ordinary task of line breaking.

## Combining paragraphs

If the desired line widths $l_i$ are not all the same, we might want to typeset two paragraphs with the second one starting in the list of line lengths where the first one leaves off. This can be done simply by treating the two paragraphs as one, i.e., appending the second to the first, assuming that each paragraph begins with an indentation and ends with finishing glue and a forced break as mentioned above.

## Patching

Suppose that a paragraph starts on page 100 of some book and continues on to the next page, and suppose that we want to make a change to the first part of that paragraph. We want to be sure that the last line of the new page 100 will end at the right-hand margin just before the word that appears at the beginning of page 101, so that page 101 doesn't have to be redone. It is easy to specify this condition in terms of our conventions, simply by forcing a line break (with penalty $-\infty$) at the desired place, and discarding the subsequent text. The ability of the optimum-fit algorithm to 'look ahead' means that it will find a suitable way to patch page 100 whenever it is possible to do so.

We can also force the altered part of the paragraph to have a certain number of lines, $k$, by using the following trick: Set the desired length $l_{k+1}$ of the $(k+1)$st line equal to some number $\theta$ that is different from the length of any other line. Then an empty box of width $\theta$ that occurs between two forced-break penalty items will have to be placed on line $k+1$.

## Punctuation in the margins

Some people prefer to have the right edge of their text look 'solid', by setting periods, commas, and other punctuation marks (including inserted hyphens) in the right-hand margin. For example, this practice is occasionally used in contemporary advertising. It is easy to get inserted hyphens into the margin: We simply let the width of the corresponding penalty item be zero. And it is almost as easy to do the same for periods and other symbols, by putting every such character in a box of width zero and adding the actual symbol width to the glue that follows. If no break occurs at this glue, the accumulated width is the same as before; and if a break does occur, the line will be justified as if the period or other symbol were not present.

## Avoiding 'psychologically bad' breaks

Since computers don't know how to think, at least not yet, it is reasonable to wonder if there aren't some line breaks that a computer would choose but a human operator might not, if they somehow don't seem right. This problem does not arise very often when straight text is being set, as in newspapers or novels, but it is quite common in technical material. For example, it is psychologically bad to break before '$x$' or '$y$' in the sentence

A function of $x$ is a rule that assigns a value $y$ to every value of $x$.

A computer will have no qualms about breaking anywhere unless it is told not to; but a human operator might well avoid bad breaks, perhaps even unconsciously.

Psychologically bad breaks are not easy to define; we just know they are bad. When the eye journeys from the end of one line to the beginning of another, in the presence of a bad break, the second word often seems like an anticlimax, or isolated from its context. Imagine turning the page between the words 'Chapter' and '8' in some sentence; you might well think that the compositor of the book you are reading should not have broken the text at such an illogical place.

During the first year of experience with TEX, the authors began to notice occasional breaks that didn't feel quite right, although the problem wasn't felt to be severe enough to warrant corrective action. Finally, however, it became difficult to justify our claim that TEX has the world's best line-breaking algorithm, when it would occasionally make breaks that were semantically annoying; for example, the preliminary TEX manual[6] has quite a few of these, and the first drafts of that manual were even worse.

As time went on, the authors grew more and more sensitive to psychologically bad breaks, not only in the copy produced by TEX but also in other published literature, and it became desirable to test the hypothesis that computers were really to blame. Therefore a systematic investigation was made of the first 1000 line breaks in the *ACM Journal* of 1960 (which was composed manually by a Monotype operator), compared to the first 1000 line breaks in the *ACM Journal* of 1980 (which was typeset by one of the best commercially available systems for mathematics, developed by Penta Systems International). The final lines of paragraphs, and the lines preceding displays, were not considered to be line breaks, since they are forced; only the texts of articles were considered, not the bibliographies. A reader who wishes to try the same experiment should find that the 1000th break in 1960 occurred on page 67, while in 1980 it occurred on page 64. The results of this admittedly subjective procedure were a total of

13 bad breaks in 1960,
55 bad breaks in 1980.

In other words, there was more than a four-fold increase, from about 1% to a quite noticeable 5·5%! Of course, this test is not absolutely conclusive, because the style of articles in the *ACM Journal* has not remained constant, but it strongly suggests that computer typesetting causes semantic degradation when it chooses breaks solely on the basis of visual criteria.

Once this problem was identified, a systematic effort was made to purge all such breaks from the second edition of Knuth's book *Seminumerical Algorithms*[9], which was the first large book to be typeset with TEX. It is quite easy to get the line-breaking algorithm to avoid certain breaks by simply prefixing the glue item by a penalty with $p_i = 999$, say; then the bad break is chosen only in an emergency, when there is no other good way to set the paragraph. It is possible to make the typist's job reasonably easy by reserving a special symbol (e.g., &) to be used instead of a normal space between words whenever breaking is undesirable. Although this problem has rarely been discussed in the literature, the authors subsequently discovered that some typographers have a word for it: they call such spaces 'auxiliary'. Thus there is a growing awareness of the problem.

It may be useful to list the main kinds of contexts in which auxiliary spaces were used in *Seminumerical Algorithms*, since that book ranges over a wide variety of technical subjects. The following rules should prove to be helpful to compositors who are keyboarding technical manuscripts into a computer.

1. Use auxiliary spaces in cross-references:

   Theorem&A  Algorithm&B  Chapter&3  Table&4  Programs E and&F

   Note that no & appears after 'Programs' in the last example, since it would be quite all right to have 'E and F' at the beginning of a line.

2. Use auxiliary spaces between a person's forenames and between multiple surnames:

   Dr.&I.&J. Matrix  Luis&I. Trabb&Pardo  Peter Van&Emde&Boas

   A recent trend to avoid spaces altogether between initials may be largely a reaction against typical computer line-breaking algorithms! Note that it seems better to hyphenate a name than to break it between words; e.g., 'Don-' and 'ald E. Knuth' is more tolerable than 'Donald' and 'E. Knuth'. In a sense, rule 1 is a special case of rule 2, since we may regard 'Theorem A' as a name; another example is 'register&X'.

3. Use auxiliary spaces for symbols in apposition with nouns:

   base&$b$  dimension&$d$  function&$f(x)$  string&$s$ of length&$l$

   However, compare the last example with 'string&$s$ of length $l$&or more'.

4. Use auxiliary spaces for symbols in series:

   1,&2, or&3  $a$,&$b$, and&$c$  1,&2, . . . ,&$n$

5. Use auxiliary spaces for symbols as tightly-bound objects of prepositions:

   of&$x$  from 0 to&1  increase $z$ by&1  in common with&$m$

   This does not apply with compound objects: For example, type 'of $u$&and&$v$'.

6. Use auxiliary spaces to avoid breaking up mathematical phrases that are rendered in words:

   equals&$n$  less than&$\varepsilon$  mod&2  modulo&$p^e$  (given&$X$)

   Also type 'If $t$&is . . .', 'when $x$&grows'. Compare 'is&15', with 'is 15&times the height'; and compare 'for all large&$n$' with 'for all $n$&greater than&$n_0$'.

7. Use auxiliary spaces when enumerating cases:

   (b)&Show that $f(x)$ is (1)&continuous; (2)&bounded.

It would be nice to boil these seven rules down into one or two, and it would be even nicer if the rules could be automated so that keyboarding could be done without them; but subtle semantic considerations seem to be involved in many of these instances. Most examples of psychologically bad breaks seem to occur when a single symbol or a short group of symbols appears just before or after the break; one could do reasonably well with an automatic scheme if it would associate large penalties with a break just before a short non-word, and medium penalties with a break just after a short non-word. Here 'short non-word' means a sequence of symbols that is not very long, yet long enough to include instances like 'exercise&15(b)', 'length&$2^{35}$', 'order&$n/2$' followed by punctuation marks; one should not simply consider patterns that have only one or two symbols. On the other hand it is not so offensive to break before or after fairly long sequences of symbols; e.g., 'exercise 4.3.2-15' needs no auxiliary space.

Many books on composition recommend against breaking just before the final word of a paragraph, especially if that word is short; this can, of course, be done by using an auxiliary space just before that last word, and the computer could insert this automatically. Some books also give recommendations analogous to rule 2 above, saying that compositors should try not to break lines in the middle of a person's name. But there is apparently only one book that addresses the other issues of psychologically bad breaks, namely a nineteenth-century French manual by A. Frey[10], where the following examples of undesirable breaks are mentioned (vol. 1, p. 110):

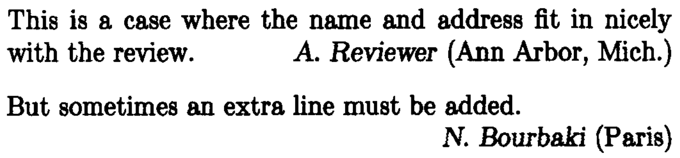
Henri&IV M.&Colin $1^{er}$&sept. art.&25 20&fr.

It seems to be time to resurrect such old traditions of fine printing.

Recent experience of the authors indicates that it is not a substantial additional burden to insert auxiliary spaces when entering a manuscript into a computer. The careful use of such spaces may in fact lead to greater job satisfaction on the part of the keyboard operator, since the quality of the output can be noticeably improved with comparatively little work. It is comforting at times to know that the machine needs your help.

## Author lines

Most of the review notices published in *Mathematical Reviews* are signed with the reviewer's name and address, and this information is typeset flush right, i.e., at the right-hand margin. If there is sufficient space to put such a name and address at the right of the final line of the paragraph, the publishers can save space, and at the same time the results look better because there are no strange gaps on the page. During recent years the composition software used by the American Mathematical Society was unable to do this operation, but the amount of money saved on paper made it economical for them to pay someone to move the reviewer-name lines up by hand wherever possible, applying scissors and (real) glue to the computer output.



This is a case where the name and address fit in nicely
with the review. *A. Reviewer* (Ann Arbor, Mich.)

But sometimes an extra line must be added.
*N. Bourbaki* (Paris)

*Figure 8. The MR problem.*

Let us say that the 'MR problem' is to typeset the contents of a given box flush right at the end of a given paragraph, with a space of at least $w$ between the paragraph and the box if they occur on the same line. This problem can be solved entirely in terms of the box/glue/penalty primitives, as follows:

$\langle$text of the given paragraph$\rangle$
penalty$(0, \infty, 0)$
glue$(0, 100000, 0)$
penalty$(0, 50, 0)$
glue$(w, 0, 0)$
box$(0)$
penalty$(0, \infty, 0)$
glue$(0, 100000, 0)$
$\langle$the given box$\rangle$
penalty$(0, -\infty, 0)$

The final penalty of $-\infty$ forces the final line break with the given box flush right; the two penalties of $+\infty$ are used to inhibit breaking at the following glue items. Thus, the above sequence reduces to two cases: whether or not to break at the penalty of 50. If a break is taken there, the 'glue$(w, 0, 0)$' disappears, according to our rule that each line begins with a box; the text of the paragraph preceding the penalty of 50 will be followed by 'glue$(0, 100000, 0)$', which will stretch to fill the line as if the paragraph had ended normally, and the given box on the final line will similarly be preceded by 'glue$(0, 100000, 0)$' to fill the gap at the left. On the other hand if no break occurs at the penalty of 50, the net effect is to have the glues added all together, producing

$\langle$text of the given paragraph$\rangle$
glue$(w, 200000, 0)$
$\langle$the given box$\rangle$

so that the space between the paragraph and the box is $w$ or more. Whether the break is chosen or not, the badness of the two final lines or the final line will be essentially zero, because so much stretchability is present. Thus the relative cost differential separating the two alternatives is almost entirely due to the penalty of 50. The optimum-fit algorithm will choose the better alternative, based on the various possibilities it has for setting the given paragraph; it might even make the given paragraph a little bit tighter than its usual setting, if this words out best.

## Ragged right margins

We observed in Figure 6 that an optimum line-breaking algorithm intended for justified text does a fairly good job at making lines of nearly equal length even when the lines aren't justified afterwards. However, it is not hard to construct examples in which the justification-oriented method makes bad decisions, since the amount of deviation in line width is weighted by the amount of stretchability or shrinkability that is present. A line containing many words, and therefore containing many spaces between words, will not be considered problematical by the justification criteria even if it is rather short or rather long, because there is enough glue present to stretch or shrink gracefully to the correct size. Conversely, when there are few words in a line, the algorithm will take pains to avoid comparatively small deviations. This is illustrated in Figure 5, which actually reads better than the corresponding paragraph in Figure 6 (except for the word that sticks out on the first line); hyphens were inserted into the paragraph of Figure 6 in order to create more interword space for justification.

Although the box/glue/penalty model appears at first glance to be oriented solely to the problem of justified text, we shall now see that it is powerful enough to be adapted to the analogous problem of unjustified typesetting: If the spaces between words are handled in the right way, we can make things work out so that each line has the same amount of stretchability, no matter how many words are on that line. The idea is to let spaces between words be represented by the sequence

$$\begin{array}{l}\text{glue}(0, 18, 0)\\ \text{penalty}(0, 0, 0)\\ \text{glue}(6, -18, 0)\end{array}$$

instead of the 'glue(6, 3, 2)' we used for justified typesetting. We may assume that there is no break at the 'glue(0, 18, 0)' in the sequence, because it will always be at least as good for the algorithm to break at the 'penalty(0, 0, 0)', when 18 units of stretchability are present. If a break occurs at the penalty, there will be a stretchability of 18 units on the line, and the 'glue(6, −18, 0)' will be discarded after the break so that the next line will begin flush left. On the other hand if no break occurs, the net effect is to have glue(6, 0, 0), representing a normal space with no stretching or shrinking.

Note that the stretchability of −18 in the second glue item has no physical significance, but it nicely cancels out the stretchability of +18 in the first glue item. Negative stretchability has several interesting applications, so the reader should study this example carefully before proceeding to the more elaborate constructions below.

Optional hyphenations in unjustified text can be specified in a similar way; instead of using 'penalty(6, 50, 1)' for an optional 6-unit hyphen having a penalty of 50, we can use the sequence

$$\begin{array}{l}\text{penalty}(0, \infty, 0)\\ \text{glue}(0, 18, 0)\\ \text{penalty}(6, 500, 1)\\ \text{glue}(0, -18, 0).\end{array}$$

The penalty has been increased here from 50 to 500, since hyphenations are not as desirable in unjustified text. After the breakpoints have been chosen using the above sequences for spaces and for optional hyphens, the individual lines should not actually be justified, since a hyphen inserted by the 'penalty(6, 500, 1)' would otherwise appear at the right margin.

It is not difficult to prove that this approach to ragged-right typesetting will never lead to words that 'stick out' in the sense mentioned above; the total demerits are reduced whenever a word that sticks out is moved to the following line.

### Centered text

Occasionally we want to take some text that is too long to fit on one line and break it into approximately equal-size parts, centering the parts on individual lines. This is most often done when setting titles or captions, but it can also be applied to the text of a paragraph, as shown in Figure 9.

**In olden times when wishing still helped one, there lived a king**
**whose daughters were all beautiful; and 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.**

*Figure 9. 'Ragged-centered' text: The optimum-fit algorithm will produce special effects like this, when appropriate combinations of box/glue/penalty items are used for the spaces between words.*

Boxes, glue, and penalties can perform this operation, in the following way: (a) At the beginning of the paragraph, use 'glue(0, 18, 0)' instead of an indentation. (b) For each space between words in the paragraph, use the sequence

$$\begin{array}{l}\text{glue}(0, 18, 0)\\ \text{penalty}(0, 0, 0)\\ \text{glue}(6, -36, 0)\\ \text{box}(0)\\ \text{penalty}(0, \infty, 0)\\ \text{glue}(0, 18, 0).\end{array}$$

(c) End the paragraph with the sequence

$$\begin{array}{l}\text{glue}(0, 18, 0)\\ \text{penalty}(0, -\infty, 0).\end{array}$$

The tricky part of this method is part (b), which ensures that an optional break at the 'penalty(0, 0, 0)' puts stretchability of 18 units at the end of one line and at the beginning of the next. If no break occurs, the net effect will be glue(0, 18, 0) + glue(6, −36, 0) + glue(0, 18, 0) = glue(6, 0, 0), a fixed space of 6 units. The 'box(0)' contains no text and occupies no space; its function is to keep the 'glue(0, 18, 0)' from disappearing at the beginning of a line. The 'penalty(0, 0, 0)' item could be replaced by other penalties, to represent breakpoints that are more or less desirable. However, this technique cannot be used together with optional hyphenation, since our box/glue/penalty model is incapable of inserting optional hyphens anywhere except at the right margin when lines are justified.

The construction used here essentially minimizes the maximum gap between the margins and the text on any line; and subject to that minimum it essentially minimizes the maximum gap on the remaining lines; and so forth. The reason is that our definitions of 'badness' and 'demerits' reduce in this case so that the sum of demerits for any choice of breakpoints is approximately proportional to the sum of the sixth powers of the individual gaps.

## ALGOL-like languages

One of the most difficult tasks in technical typesetting is to get computer programs to look right. In addition to the complications of mathematical formulas and a variety

```
const n = 10000;
var sieve, primes :
      set of 2..n;
  next, j : integer;
begin { initialize }
sieve := [2..n];
primes := [];
next := 2;
repeat { find next
      prime }
  while not (next in
        sieve) do
    next :=
        succ(next);
  primes :=
      primes + [next];
  j := next;
  while j <= n do
         { eliminate }
    begin sieve :=
        sieve − [j];
    j := j + next
    end
until sieve = []
end.
```

```
const n = 10000;
var sieve, primes : set of 2..n;
  next, j : integer;
begin { initialize }
sieve := [2..n]; primes := []; next := 2;
repeat { find next prime }
  while not (next in sieve) do next := succ(next);
  primes := primes + [next]; j := next;
  while j <= n do { eliminate }
    begin sieve := sieve − [j]; j := j + next
    end
until sieve = []
end.
```

*Figure 10. These two settings of a sample PASCAL program were made from identical input specifications in the box/glue/penalty model; in the first case the lines were set 100 points wide, and in the second case the width was 250 points. All of the line-breaking and identation was produced automatically by the optimum-fit algorithm, which has no specific knowledge of PASCAL. Compilation of the PASCAL source code into boxes, glue, and penalties was done mechanically.*

of typestyles and spacing conventions, it is important to indent the lines suitably in order to display the program structure. Sometimes a single statement must be broken across several lines; sometimes a number of short statements should be grouped together on a single line. Authors who attempt to publish programs in journals that are not accustomed to computer science material soon discover that very few printing establishments have the expertise necessary to handle ALGOL-like languages in a satisfactory way.

Once again, the concepts of boxes, glue, and penalties come to the rescue: It turns out that our line-breaking methods developed for ordinary text can be used without change to do the typesetting of programs in ALGOL-like languages. For example, Figure 10 shows a typical program taken from the PASCAL manual[11] that has been typeset assuming two different column widths. Although these two settings of the program do not look very much alike, they both were made from exactly the same input, specified in terms of boxes, glue, and penalties; the only difference was the specification of line width. (The input text in this example was prepared by a computer program called BLAISE[12], which will translate any PASCAL source text into a TEX file that can be incorporated within other documents.)

The box/glue/penalty specifications that lead to Figure 10 involve constructions similar to those we have seen above, but with some new twists; it will be sufficient for our purposes merely to sketch the ideas instead of dwelling on the details. One key point is that the breaks are chosen by the minimum-demerits criteria we have been discussing, but the lines are not justified afterwards (i.e., the glue does not actually stretch or shrink). The reason is that relations and assignment statements are processed by TEX's normal 'math mode', which allows line breaks to occur in various places but without any special constructions particular to this application, so that justification would have the undesirable effect of putting all such breaks at the right margin. The fact that justification is suppressed actually turns out to be an advantage in this case, since it means that we can insert glue stretching wherever we like, within a line, if it affects the 'badness' formula in a desirable way.

Each line in the wider setting of Figure 10 is actually a 'paragraph' by itself, so it is only the narrower setting that shows the line-breaking mechanism at work. Every 'paragraph' has a specified amount of indentation for its first line, corresponding to its position in the program, as a given number $t$ of 'tab' units; the paragraph is also given a *hanging indentation* of $t+2$ tab units. This means that all lines after the first are required to be two tabs narrower than the first line, and they are shifted two tabs to the right with respect to that line. In some cases (e.g., those lines beginning with '**var**' or '**while**') the offset is three tabs instead of two.

The paragraph begins with 'glue(0, 100000, 0)', which has the effect of providing enough stretchability that the line-breaking algorithm will not wince too much at breaks that do not square perfectly with the right margin, at least not on the first line. Special breaks are inserted at places where TEX would not normally break in math mode; e.g., the sequence

$$\begin{array}{l} \text{penalty}(0, \infty, 0) \\ \text{glue}(0, 100000, 0) \\ \text{penalty}(0, 50, 0) \\ \text{glue}(0, -100000, 0) \\ \text{box}(0) \\ \text{penalty}(0, \infty, 0) \\ \text{glue}(0, 100000, 0) \end{array}$$

has been inserted just before '*primes*' in the **var** declaration. This sequence allows a break with penalty 50 to the next line, which begins with plenty of stretchability. A similar construction is used between assignment statements, for example between '$sieve := [2\,..\,n]$;' and '$primes := [\,]$', where the sequence is

```
penalty(0, ∞, 0)
glue(0, 100000, 0)
penalty(0, 0, 0)
glue(6 + 2w, −100000, 0)
box(0)
penalty(0, ∞, 0)
glue( −2w, 100000, 0);
```

here $w$ is the width of a tab unit. If a break occurs, the following line begins with 'glue( $-2w$, 100000, 0)', which undoes the effect of the hanging indentation and effectively restores the state at the beginning of a paragraph. If no break occurs, the net effect is 'glue(6, 100000, 0)', a normal space.

No automatic system can hope to find the best breaks in programs, since an understanding of the semantics will indicate that certain breaks make the program clearer and reveal its symmetries better. However, dozens of experiments on a wide variety of PASCAL source texts have shown that this approach is surprisingly effective; fewer than 1% of the line-breaking decisions have been overridden by authors of the programs in order to provide additional clarity.

## A complex index

The final application of line breaking that we shall study is the most difficult one that has so far been encountered by the authors; it was solved only after acquiring more than two years of experience with more straightforward line-breaking tasks, since the full power of the box/glue/penalty primitives was not immediately apparent. The task is illustrated in Figure 11, which shows excerpts from a 'Key Index' in *Mathematical Reviews*. Such an index now appears at the end of each volume, together with an 'Author Index' that has a similar format.

As in Figure 10, the examples in Figure 11 were generated by the same source input that was typeset using different line widths, in order to indicate the various possibilities of breakpoints. Each entry in the index consists of two parts, the *name part* and the *reference part*, both of which might be too long to fit on a single line. If line breaks occur in the name part, the individual lines are to be set with a ragged right margin, but breaks in the reference part are to produce lines with a ragged *left* margin. The two parts are separated by *leaders*, a row of dots that expands to fill the space between them; leaders are introduced by a slight generalization of glue that typesets copies of a given box into a given space, instead of leaving that space blank. A hanging indentation is applied to all lines but the first, so that the first line of each entry is readily identifiable. One of the goals in breaking such entries is to minimize the white space that appears in ragged-right or ragged-left lines. A subsidiary goal is to minimize the number of lines that contain the reference part; for example, if it is possible to fit all of the references on one line, the line-breaking algorithm should do so. The latter event might mean that a break occurs after the leaders, with the references starting on a new line; in such a case the leaders should stop a fixed distance $w_1$ from the right margin. Furthermore, the ragged-right lines should all be at least a fixed distance $w_2$ from the right margin, so that there is no chance of confusing part of the name with part of the reference material. The individual boxes to be replicated in the leaders are $w_3$ units wide.

ACM Symposium on Principles of Programming
Languages, Third (Atlanta, Ga., 1976), selected
papers ....................................*1858
ACM Symposium on Theory of Computing, Eighth
Annual (Hershey, Pa., 1976) ........1879, 4813,
5414, 6918, 6936, 6937, 6946, 6951, 6970, 7619,
9605, 10148, 11676, 11687, 11692, 11710, 13869
Software ................................ *See* *1858

ACM Symposium
on Principles of
Programming
Languages, Third
(Atlanta, Ga., 1976),
selected papers ....*1858
ACM Symposium on
Theory of Computing,
Eighth Annual
(Hershey, Pa., 1976)
........ 1879, 4813, 5414,
6918, 6936, 6937, 6946,
6951, 6970, 7619, 9605,
10148, 11676, 11687,
11692, 11710, 13869
Software ......... See *1858

ACM Symposium on Principles of
Programming Languages, Third
(Atlanta, Ga., 1976), selected papers
................................ *1858
ACM Symposium on Theory of
Computing, Eighth Annual
(Hershey, Pa., 1976) ..........
1879, 4813, 5414, 6918, 6936, 6937,
6946, 6951, 6970, 7619, 9605, 10148,
11676, 11687, 11692, 11710, 13869
Software ..................... *See* *1858

*Figure 11. These three extracts from a 'Key Index' were all typset from identical input, with respective column widths of 225 points, 175 points, and 125 points. Note the combination of ragged right and ragged left setting, and the 'dot leaders'.*

The ground rules are illustrated in Figure 11, where there is a hanging indentation of 27 units, and $w_1 = 45$, $w_2 = 9$, $w_3 = 7{\cdot}2$; the digits are 9 units wide, and the respective column widths are 405 units, 315 units, and 225 units. The entry for 'Theory of Computing' shows three possibilities for the leader dots: They can share a line with the end of the name part and the beginning of the reference part, or they can end a line before the reference part or begin a line after the name part.

Here is how all this can be encoded with boxes, glue, and penalties: (a) Each blank space in the name part is represented by the sequence

$$\begin{array}{l} \mathrm{penalty}(0, \infty, 0) \\ \mathrm{glue}(w_2, 18, 0) \\ \mathrm{penalty}(0, 0, 0) \\ \mathrm{glue}(6 - w_2, -18, 2) \end{array}$$

which yields ragged right margins and spaces that can shrink from 6 units to 4 units if necessary. (b) The transition between name part and reference part is represented by sequence (a) followed by

$$\begin{array}{l}\text{box}(0)\\ \text{penalty}(0,\infty,0)\\ \text{leaders}(3w_3,100000,3w_3)\\ \text{glue}(w_1,0,0)\\ \text{penalty}(0,0,0)\\ \text{glue}(-w_1,-18,0)\\ \text{box}(0)\\ \text{penalty}(0,\infty,0)\\ \text{glue}(0,18,0).\end{array}$$

(c) Each blank space in the reference part is represented by the sequence

$$\begin{array}{l}\text{penalty}(0,999,0)\\ \text{glue}(6,-18,2)\\ \text{box}(0)\\ \text{penalty}(0,\infty,0)\\ \text{glue}(0,18,0),\end{array}$$

which yields ragged left margins and 6-unit to 4-unit spaces.

Parts (a) and (c) of this construction are analogous to things we have seen before; the 999-point penalties in (c) tend to minimize the total number of lines occupied by the reference part. The most interesting aspect of this construction is the transition sequence (b), where there are four possibilities: If no line breaks occur in (b), the net result is

$$\langle\text{name part}\rangle\ \text{glue}(6,0,2)\ \langle\text{leaders}\rangle\ \langle\text{reference part}\rangle,$$

which allows leader dots to appear between the name and reference parts on the current line. If a line break occurs before the leaders, the net result is

$$\begin{array}{l}\langle\text{name part}\rangle\ \text{glue}(6,0,2)\\ \langle\text{leaders}\rangle\ \langle\text{reference part}\rangle,\end{array}$$

so that we have a break essentially like that after a blank space in the name part, and the dot leaders begin the following line. If a line break occurs after the leaders, the net result is

$$\begin{array}{l}\langle\text{name part}\rangle\ \text{glue}(6,0,2)\ \langle\text{leaders}\rangle\ \text{glue}(w_1,0,0)\\ \text{glue}(0,18,0)\ \langle\text{reference part}\rangle,\end{array}$$

so that we have a break essentially like that after a blank space in the reference part but without the penalty of 999; the leaders end $w_1$ units from the right margin. Finally, if breaks occur both before and after the leaders in (b), we have a situation that always has more demerits than the alternative of breaking only before the leaders.

When the choice of breakpoints leaves room for at least $3w_3$ units of leaders, we are sure to have at least two dots, but we might not have three dots since leader dots on different lines are aligned with each other. The glue in other blank spaces on the line with the leaders will shrink if there is less than $3w_3$ of space for the leaders, and this tends to make it more likely that the leader dots will not disappear altogether; however, in the worst case the space for leaders will shrink to zero, so there might not be any dots visible. It would be possible to ensure that all the leaders contain at least two dots, by simply setting the shrink component of the leader item in (b) to zero. This would improve the appearance of the resulting output; but unfortunately it would also increase the length of the author indexes by about 15 per cent, and such an expense would probably be prohibitive.

A preliminary version of this construction has been used with TEX to prepare the indexes of *Mathematical Reviews* since November, 1979. However, the items 'box(0) penalty(0, $\infty$, 0)' were left out of (b), for compatibility with earlier indexes prepared by other typesetting software; this means that the leaders disappear completely whenever a break occurs just before them, and the resulting indexes have unfortunate gaps of white space that spoil their appearance.

## AN ALGEBRAIC APPROACH

The examples we have just seen show that boxes, glue, and penalties are quite versatile primitives that allow a user to obtain a wide variety of effects without extending the basic operations needed for ordinary typesetting. However, some of the constructions may have seemed like 'magic'; they work, but it isn't clear how they were ever conceived in the first place. We shall now study a fairly systematic way to deal with these primitives in order to assess their full potentiality; this brief discussion is independent of the remainder of the paper and can be omitted.

In the first place it is clear that

$$\mathrm{box}(w)\,\mathrm{box}(w') = \mathrm{box}(w+w'),$$

if we ignore the contents of the boxes and consider only the widths; only the widths enter into the line-breaking criteria. This formula says that any two consecutive boxes can be replaced by a single box without affecting the choice of breakpoints, since breaks do not occur at box items. Similarly it is easy to verify that

$$\mathrm{glue}(w,y,z)\,\mathrm{glue}(w',y',z') = \mathrm{glue}(w+w',y+y',z+z'),$$

since there will be no break at glue($w',y',z'$), and since a break at glue($w,y,z$) is equivalent to a break at glue($w+w',y+y',z+z'$).

Under certain circumstances we can also combine two adjacent penalty items into a single one; for example, if $-\infty < p, p' < +\infty$ we have

$$\mathrm{penalty}(w,p,f)\,\mathrm{penalty}(w,p',f) = \mathrm{penalty}(w,\min(p,p'),f)$$

with respect to any optimal choice of breakpoints, since there are fewer demerits associated with the smaller penalty. However, it is not always possible to replace the general sequence 'penalty($w,p,f$) penalty($w',p',f'$)' by a single penalty item.

We can assume without loss of generality that all box items are immediately followed by a sequence of the form 'penalty(0, $\infty$, 0) glue($w,y,z$)'. For if the box is followed by another box, we can combine the two; if it is followed by a penalty item with $p<\infty$, we can insert 'penalty(0, $\infty$, 0) glue(0, 0, 0)'; if it is followed by 'penalty($w$, $\infty$, $f$)' we can assume that $w = f = 0$ and that the following item is glue; and if the box is followed by glue, we can insert 'penalty$(0, \infty, 0)$ glue$(0, 0, 0)$ penalty$(0, 0, 0)$'. Furthermore we can delete any penalty item with $p = \infty$ if it is not immediately preceded by a box item.

Thus, any sequence of box/glue/penalty items can be converted into a 'normal form', where each box is followed by a penalty of $\infty$, each penalty is followed by glue, and each glue is either followed by a penalty $<\infty$ or by a box. We assume that there is only one penalty $-\infty$, and that it is the final item, since a forced line break effectively separates a longer sequence into independent parts. It follows that the normal-form sequences can be written

$$X_1 X_2 \ldots X_n \,\text{penalty}(w, -\infty, f)$$

where each $X_i$ is a sequence of items having the form

$$\text{box}(w)\,\text{penalty}(0, \infty, 0)\,\text{glue}(w', y, z)$$

or the form

$$\text{penalty}(v, p, f)\ \text{glue}(w, y, z).$$

Let us use the notation bpg$(w+w', y, z)$ for the first of these two forms, noting that it is a function of $w+w'$ rather than of $w$ and $w'$ separately; and let us write pg$(v, p, f, w, y, z)$ for $X$'s of the second form. We can assume that the sequence of $X$'s contains no two bpg's in a row, since

$$\text{bpg}(w, y, z)\,\text{bpg}(w', y', z') = \text{bpg}(w+w', y+y', z+z').$$

Familiarity with this algebra of boxes, glue, and penalties makes it a fairly simple matter to invent constructions for special applications like those listed above, whenever such constructions are possible. For example, let us consider a generalization of the problems arising in ragged-right, ragged-left, and ragged-centered text: We wish to specify on optional break between words such that if no break occurs we will have the sequence

$$\langle\text{end of text}_1\rangle\,\text{glue}(w_1, y_1, z_1)\,\langle\text{beginning of text}_2\rangle$$

on one line, while if a break does occur we will have

$$\langle\text{end of text}_1\rangle\ \text{glue}(w_2, y_2, z_2)\ \text{penalty}(w_0, p, f)$$
$$\text{glue}(w_3, y_3, z_3)\,\langle\text{beginning of text}_2\rangle$$

on two lines. A consideration of normal forms shows that the most general thing we can do is to insert the sequence

$$\text{bpg}(w, y, z)\ \text{pg}(w_0, p, f, w', y', z')\,\text{bpg}(w'', y'', z'')$$

between text$_1$ and text$_2$, where no additional text is associated with the two inserted bpg's. Our job reduces therefore to determining appropriate values of $w, y, z, w', y', z'$, $w''$, $y''$, $z''$, and these can be obtained immediately by solving the equations

$$\begin{aligned} w+w'+w'' &= w_1, & y+y'+y'' &= y_1, & z+z'+x'' &= z_1;\\ w &= w_2, & y &= y_2, & z &= z_2;\\ w'' &= w_3, & y'' &= y_3, & z'' &= z_3. \end{aligned}$$

Once a construction has been found in this way, it can be simplified by undoing the process we have used to derive normal forms and by using other properties of box/glue/penalty algebra. For example, we can always delete the penalty $\infty$ item in a sequence like

$$\mathrm{penalty}(0,\infty,0)\,\mathrm{glue}(0,y,z)\,\mathrm{penalty}(0,p,0),$$

if $y \geqslant 0$ and $z \geqslant 0$ and $p < 0$, since a break at the glue is always worse than a break at the penalty $p$.

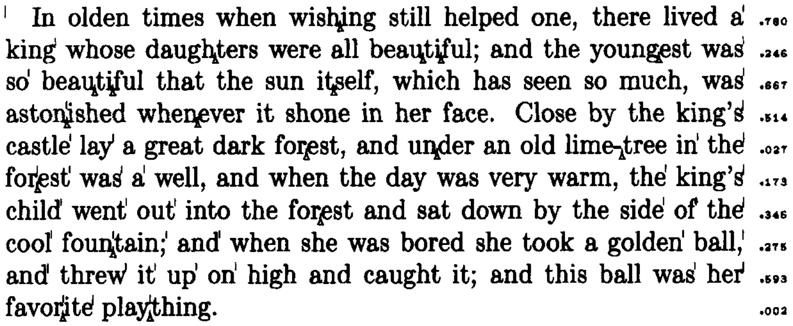
## INTRODUCTION TO THE ALGORITHM

The general ideas underlying the optimum-fit algorithm for line breaking can probably be understood best by considering an example. Figure 12 repeats the paragraph of Figure 4(c) and includes little vertical marks to indicate 'feasible breakpoints' found by the algorithm. A *feasible breakpoint* is a place where the text of the paragraph from the beginning to this point can be broken into lines whose adjustment ratio does not exceed a given tolerance; in the case of Figure 12, this tolerance was taken to be unity. Thus, for example, there is a tiny mark after 'fountain;' since there is a way to set the paragraph up to this point with 'fountain;' at the end of the 7th line and with none of lines 1 to 7 having a badness exceeding 100 (cf. Figure 4(a)).

The algorithm proceeds by locating all of the feasible breakpoints and remembering the best way to get to each one, in the sense of fewest total demerits. This is done by keeping a list of 'active' breakpoints, representing all of the feasible breakpoints that might be a candidate for future breaks. Whenever a potential breakpoint $b$ is encountered, the algorithm tests to see if there is any active breakpoint $a$ such that the line from $a$ to $b$ has an acceptable adjustment ratio. If so, $b$ is a feasible breakpoint and it is appended to the active list. The algorithm also remembers the identity of the breakpoint $a$ that minimizes the total demerits, when the total is computed from the beginning of the paragraph, through $a$, to $b$. When an active breakpoint $a$ is encountered for which the line from $a$ to $b$ has an adjustment ratio less than $-1$ (i.e., when the line can't be shrunk to fit the desired length), breakpoint $a$ is removed from the active list. Since the size of the active list is essentially bounded by the maximum number of words per line, the running time of the algorithm is bounded by this quantity (which usually is small) times the number of potential breakpoints.

For example, when the algorithm begins to work on the paragraph in Figure 12, there is only one active breakpoint, representing the beginning of the first line. It is infeasible to have a line starting there and ending at 'In', or 'olden', . . . , or 'lived', since the glue between words does not accumulate enough stretchability in such short segments of the text; but after the next word 'a' is encountered, a feasible breakpoint is found. Now there are two active breakpoints, the original one and the new one. After the next word 'king', there are three active breakpoints; but after the next word 'whose', the algorithm sees that it is impossible to squeeze all of the text from the beginning up to 'whose' on one line, so the initial breakpoint becomes inactive and only two active ones remain.

Skipping ahead, let us consider what happens when the algorithm considers the potential break after 'fountain;'. At this stage there are eight active breakpoints, following the respective text boxes for 'child', 'went', 'out', 'side', 'of', 'the', 'cool',



In olden times when wishing still helped one, there lived a .780
king whose daughters were all beautiful; and the youngest was .246
so beautiful that the sun itself, which has seen so much, was .667
astonished whenever it shone in her face. Close by the king's .514
castle lay a great dark forest, and under an old lime-tree in the .027
forest was a well, and when the day was very warm, the king's .173
child went out into the forest and sat down by the side of the .346
cool fountain; and when she was bored she took a golden ball, .275
and threw it up on high and caught it; and this ball was her .593
favorite plaything. .002

*Figure 12. Tiny vertical marks show 'feasible breakpoints' where it is possible to break in such a way that no spaces need to stretch more than their given stretchability.*

and 'foun-'. The line starting after 'child' and ending with 'fountain;' would be too long to fit, so 'child' becomes inactive. Feasible lines are found from 'went' or 'out' to 'fountain;' and the demerits of those lines are 276 and 182, respectively; however, the line from 'went' actually turns out to be preferable, since there are substantially fewer total demerits from the beginning of the paragraph to 'went' than to 'out'. Thus, 'fountain;' becomes a new active breakpoint. The algorithm stores a pointer back from 'fountain;' to 'went', meaning that the best way to get to a break after 'fountain;' is to start with the best way to get to a break after 'went'.

The computation of this algorithm can be represented pictorially by means of the network in Figure 13, which shows all of the feasible breakpoints together with the number of demerits charged for each feasible line between them. The object of the algorithm is to compute the *shortest path* from the top of Figure 13 to the bottom, using the demerit numbers as the 'distances' corresponding to individual parts of the path. In this sense, the job of optimal line breaking is essentially a special case of the problem of finding shortest paths in an acyclic network; the line-breaking algorithm is slightly more complex only because it must construct the network at the same time as it is finding the shortest path.

Notice that the best-fit algorithm can be described very easily in terms of a network like Figure 13: it is the algorithm that simply chooses the shortest continuation at every step. And the first-fit algorithm can be characterized as the method of always taking the leftmost branch having a negative adjustment ratio (unless it leads to a hyphen, in which case the rightmost non-hyphenated branch is chosen whenever there is a feasible one). From these considerations we can readily understand why the optimum-fit algorithm tends to do a much better job.

Sometimes there is no way to continue from one feasible breakpoint to any other. This situation doesn't occur in Figure 13, but it would be present below the word 'so' if we had not permitted hyphenation of 'astonished'. In such cases the first-fit and best-fit algorithms must resort to infeasible lines, while the optimum-fit algorithm can usually find another way through the maze.

On the other hand, some paragraphs are inherently difficult, and there is no way to break them into feasible lines. In such cases the algorithm we have described will find that its active list dwindles until eventually there is no activity left; what should be done in such a case? It would be possible to start over with a more tolerant attitude

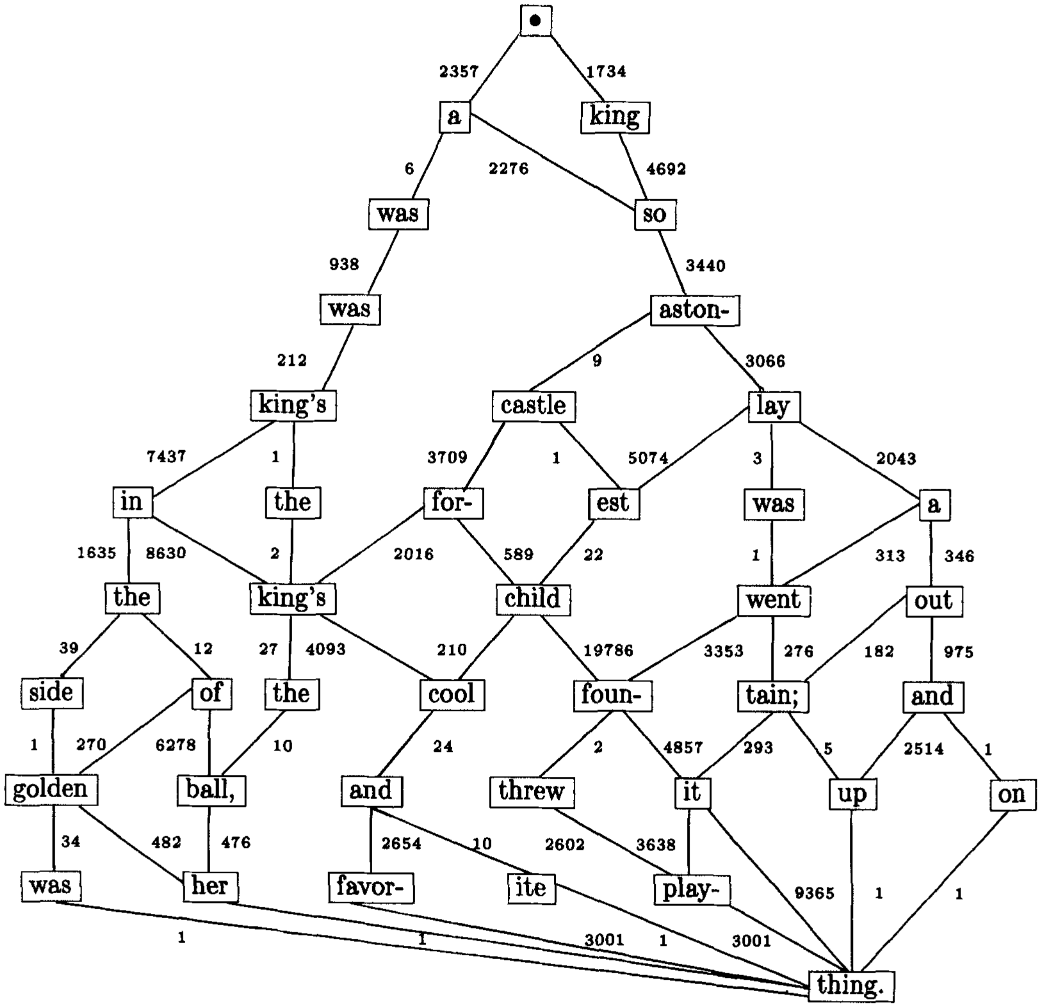
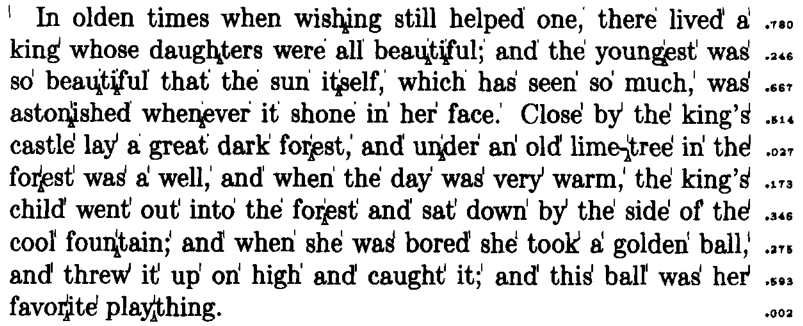


*Figure 13. This network shows the feasible breakpoints and the number of demerits charged when going from one breakpoint to another. The 'shortest path' from the top to the bottom corresponds to the best way to typeset the paragraph, if we regard the demerits as distances.*

toward infeasibility (a higher threshold value for the adjustment ratios). Alternatively, TEX takes the attitude that the user wants to make some manual adjustment when there is no way to meet the specified criteria, so the active list is forcibly prevented from becoming empty by simply declaring a breakpoint to be feasible if it would otherwise leave the active list empty. This results in an overset line and an error message that encourages the user to take corrective action.

Figure 14 shows what happens when the algorithm allows quite loose lines to be feasible; in this case a line is considered to be infeasible only if its adjustment ratio exceeds 10 (so that there would be more than two ems of space between words). Such a setting of the tolerances would be used by people who don't want to make manual adjustments to paragraphs that cannot be set well. The tiny marks that indicate feasible breakpoints have varying lengths in this illustration, with longer marks



In olden times when wishing still helped one, there lived a .780
king whose daughters were all beautiful; and the youngest was .246
so beautiful that the sun itself, which has seen so much, was .667
astonished whenever it shone in her face. Close by the king's .514
castle lay a great dark forest, and under an old lime-tree in the .027
forest was a well, and when the day was very warm, the king's .173
child went out into the forest and sat down by the side of the .346
cool fountain; and when she was bored she took a golden ball, .275
and threw it up on high and caught it; and this ball was her .593
favorite plaything. .002

*Figure 14. When the tolerance is raised to 10 times the stretchability, more breakpoints become feasible, and there are many more possibilities to explore.*

indicating places that can be reached via better paths; the tiny dots are for breakpoints that are just barely feasible. Notice that all of the potential breakpoints in Figure 14 are marked, except for a few in the first two lines; so there are considerably more feasible breakpoints here than there were in Figure 12, and the network corresponding to Figure 13 will be much larger. There are 836,272,858 feasible ways to set the paragraph when such wide spaces are tolerated, compared to only 81 ways in Figure 12. However, the number of active nodes will not be significantly bigger in this case than it was in Figure 12, because it is limited by the length of a line, so the algorithm will not run too much more slowly even though its tolerance has been raised and the number of possible settings has increased enormously. For example, after 'fountain;' there are now 17 active breakpoints instead of the 8 present before, so the processing takes only about twice as long although huge numbers of additional possibilities are being taken into account.

When the threshold allows wide spacing, the algorithm is almost certain to find a feasible solution, and it will report no errors to the user even though some rather loose lines may have been necessary. The user who wants such error messages should set the tolerance lower; this not only gives warnings when corrective action is needed, it also improves the algorithm's efficiency.

One of the important things to note about Figure 14 is that breakpoints can become feasible in completely different ways, leading up to different numbers of lines before the breakpoint. For example, the word 'seen' is feasible both at the end of line 3:

'In olden . . . lived/a . . . young-/est . . . seen'

and at the end of line 4:

'In olden . . . helped/one, . . . were/all . . . beau-/tiful . . . seen',

although 'seen' was not a feasible break at all in Figure 12. The breaks that put 'seen' at the end of line 3 have substantially fewer demerits than those putting it on line 4 (approximately $1{\cdot}68 \times 10^6$ versus $1{\cdot}28 \times 10^{10}$), so the algorithm will remember only the former possibility. This is an application of the dynamic-programming 'principle of optimality', which is responsible for the efficiency of our algorithm[4]: the optimum breakpoints of a paragraph are always optimum for the subparagraphs they create.

The area of a
circle is a mean propor-
tional between any two regular
and similar polygons of which one
circumscribes it and the other is iso-
perimetric with it. In addition, the area
of the circle is less than that of any cir-
cumscribed polygon and greater than that
of any isoperimetric polygon. And further,
of these circumscribed polygons, the one
that has the greater number of sides has
a smaller area than the one that has
a lesser number; but, on the other
hand, the isoperimetric polygon
that has the greater num-
ber of sides is the
larger.

— Galileo Galilei (1638)

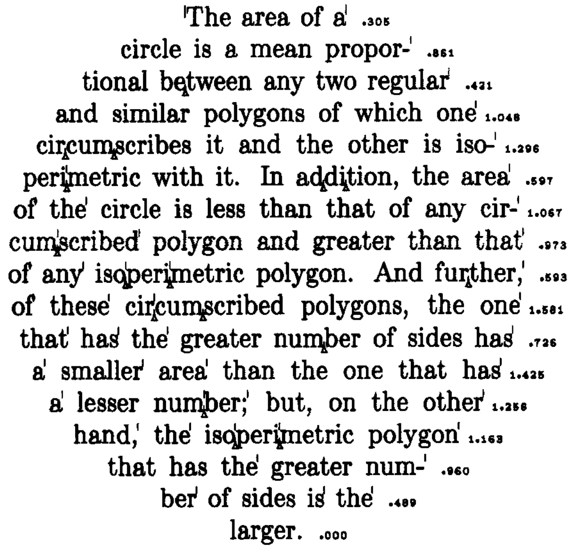
I
turn, in the
following treatises, to
various uses of those triangles
whose generator is unity. But I leave out
many more than I include; it is extraordinary how
fertile in properties this triangle is. Everyone can try his hand.

— Blaise Pascal (1654)

*Figure 15. Examples of line breaking with lines of different sizes.*

But the interesting thing is that this economy of storage would not be possible if the future lines were not all of the same length, since differing line lengths might well mean that it would be much better to put 'seen' on line 4 after all; for example, we have mentioned a trick for forcing the algorithm to produce a given number of lines. In the presence of varying line lengths, therefore, the algorithm would need to have two separate list entries for an active breakpoint after the word 'seen'. The computer cannot simply remember the one with fewest total demerits, because the optimality principle of dynamic programming would not be valid in such a case.

Figure 15 is an example of line breaking when the individual lengths are all different. In such cases, the need to attach line numbers to breakpoints might mean that the number of active breakpoints substantially exceeds the maximum number of words per line, if the feasibility tolerance is set high; so it is desirable to set the tolerance low. On the other hand, if the tolerance is set too low, there may be no way to break the paragraph into lines having a desired shape. Fortunately, there is usually a happy medium in which the algorithm has enough flexibility to find a good solution without needing too much time and space. The data in Figure 16 shows, for example, that the

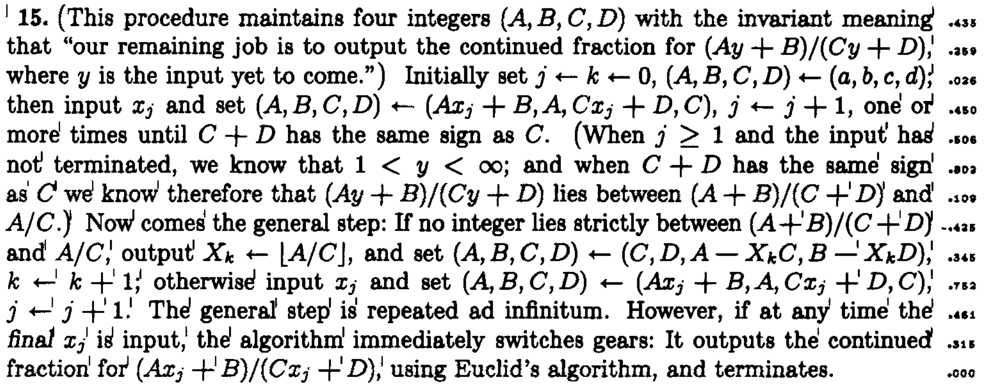


The area of a .305
circle is a mean propor- .861
tional between any two regular .421
and similar polygons of which one 1.048
circumscribes it and the other is iso- 1.296
perimetric with it. In addition, the area .597
of the circle is less than that of any cir- 1.067
cumscribed polygon and greater than that .973
of any isoperimetric polygon. And further, .593
of these circumscribed polygons, the one 1.581
that has the greater number of sides has .726
a smaller area than the one that has 1.425
a lesser number; but, on the other 1.256
hand, the isoperimetric polygon 1.163
that has the greater num- .960
ber of sides is the .489
larger. .000

*Figure 16. Details of the feasible breakpoints in the first example of Figure 15, showing how the optimum solution was found.*

algorithm did not have to do very much work to find an optimal solution for Galileo's remarks on circles, when the adjustment ratio on each feasible line was required to be 2 or less; yet there was sufficient flexibility to make feasible solutions possible.

A good line-breaking method is especially important for technical typesetting, since it is undesirable to break up mathematical formulas that appear in the text. Some of the most difficult copy of thiš kind appears in *Mathematical Reviews* or in the answer pages of *The Art of Computer Programming*, since the material in those publications is often densely packed with formulas. Figure 17 shows a typical example from the answer pages of *Seminumerical Algorithms*[9], together with indications of the feasible breaks when the adjustment ratios are constrained to be at most 1. Although some feasible breakpoints occur in the middle of formulas, they are associated with penalties that make them comparatively undesirable, so the algorithm was able to keep all of the mathematics of this paragraph intact.



**15.** (This procedure maintains four integers $(A, B, C, D)$ with the invariant meaning .435
that "our remaining job is to output the continued fraction for $(Ay + B)/(Cy + D)$, .259
where $y$ is the input yet to come.") Initially set $j \leftarrow k \leftarrow 0$, $(A, B, C, D) \leftarrow (a, b, c, d)$; .026
then input $x_j$ and set $(A, B, C, D) \leftarrow (Ax_j + B, A, Cx_j + D, C)$, $j \leftarrow j + 1$, one or .450
more times until $C + D$ has the same sign as $C$. (When $j \geq 1$ and the input has .506
not terminated, we know that $1 < y < \infty$; and when $C + D$ has the same sign .803
as $C$ we know therefore that $(Ay + B)/(Cy + D)$ lies between $(A + B)/(C + D)$ and .109
$A/C$.) Now comes the general step: If no integer lies strictly between $(A+B)/(C+D)$ -.425
and $A/C$, output $X_k \leftarrow \lfloor A/C \rfloor$, and set $(A, B, C, D) \leftarrow (C, D, A - X_kC, B - X_kD)$, .345
$k \leftarrow k + 1$; otherwise input $x_j$ and set $(A, B, C, D) \leftarrow (Ax_j + B, A, Cx_j + D, C)$, .752
$j \leftarrow j + 1$. The general step is repeated ad infinitum. However, if at any time the .461
*final* $x_j$ is input, the algorithm immediately switches gears: It outputs the continued .315
fraction for $(Ax_j + B)/(Cx_j + D)$, using Euclid's algorithm, and terminates. .000

*Figure 17. An example of the feasible breakpoints found by the algorithm in a paragraph containing numerous mathematical formulas.*

In olden times when wishing still helped one, there lived a .780
king whose daughters were all beautiful; and the youngest was so -.776
beautiful that the sun itself, which has seen so much, was aston- -.425
ished whenever it shone in her face. Close by the king's castle lay -.816
a great dark forest, and under an old lime-tree in the forest was -.191
a well, and when the day was very warm, the king's child went .107
out into the forest and sat down by the side of the cool fountain; -.538
and when she was bored she took a golden ball, and threw it up -.234
on high and caught it; and this ball was her favorite plaything. -.868

In olden times when wishing still helped one, there lived a .780
king whose daughters were all beautiful; and the youngest was .246
so beautiful that the sun itself, which has seen so much, was .667
astonished whenever it shone in her face. Close by the king's .514
castle lay a great dark forest, and under an old lime-tree in the .027
forest was a well, and when the day was very warm, the king's .173
child went out into the forest and sat down by the side of the .346
cool fountain; and when she was bored she took a golden ball, .275
and threw it up on high and caught it; and this ball was her .593
favorite plaything. .561

In olden times when wishing still helped one, there lived 1.405
a king whose daughters were all beautiful; and the young- 1.491
est was so beautiful that the sun itself, which has seen so 1.437
much, was astonished whenever it shone in her face. Close 1.156
by the king's castle lay a great dark forest, and under an 1.467
old lime-tree in the forest was a well, and when the day 1.807
was very warm, the king's child went out into the forest 1.886
and sat down by the side of the cool fountain; and when 1.551
she was bored she took a golden ball, and threw it up on 1.388
high and caught it; and this ball was her favorite play- 2.175
thing. .862

In olden times when wishing still helped one, there 3.313
lived a king whose daughters were all beautiful; and 3.510
the youngest was so beautiful that the sun itself, which 2.288
has seen so much, was astonished whenever it shone 3.636
in her face. Close by the king's castle lay a great 3.153
dark forest, and under an old lime-tree in the for- 3.858
est was a well, and when the day was very warm, 3.626
the king's child went out into the forest and sat down 2.500
by the side of the cool fountain; and when she was 3.159
bored she took a golden ball, and threw it up on 3.769
high and caught it; and this ball was her favorite play- 2.175
thing. .862

*Figure 18. Paragraphs obtained when the 'looseness' parameter has been set to* $-1$, 0, $+1$, *and* $+2$. *As in Figure 14, the spaces have been allowed to stretch up to two ems before being considered infeasible. Loose settings like this are sometimes necessary to balance a page, but of course the effects are not beautiful when one goes to extremes.*

## MORE BELLS AND WHISTLES

The optimization problem we have formulated is to find breakpoints that minimize the total number of demerits, where the demerits of a particular line depend on its badness (i.e., on how much its glue must stretch or shrink) and on a possible penalty associated with its final breakpoint; additional demerits are also added when two consecutive lines end with hyphens (i.e., end at penalty items with $f = 1$). Two years of experience with such a model of the problem gave excellent results, except that a few paragraphs showed up where further improvement was possible.

The first two lines of Figures 4(a) and 4(b) illustrate a potential source of visual disturbance that is not accounted for in the model we have been discussing: These paragraphs begin with a tight line (having $r = -\cdot 741$) immediately followed by a loose line (having $r = +\cdot 877$). Although the two lines are not offensive in themselves the contrast between tight and loose makes them appear worse. Therefore TEX's new algorithm for line breaking recognizes four kinds of lines:

Class 0 (tight lines), where $-1 \leqslant r < -\cdot 5$;
Class 1 (normal lines), where $-\cdot 5 \leqslant r < +\cdot 5$;
Class 2 (loose lines), where $+\cdot 5 \leqslant r < +1$;
Class 3 (very loose lines), where $r \geqslant +\cdot 1$.

Additional demerits are added when adjacent lines are not of the same or adjacent classes, i.e., when a Class 0 line is preceded or followed by Class 2 or Class 3, or when Class 1 is preceded or followed by Class 3.

This seemingly simple extension actually forces the algorithm to work harder, because a feasible breakpoint may now have to be entered into the active list up to four times in order to preserve the dynamic-programming principle of optimality. For example, if it is feasible to end at some point with both a Class 0 line and a Class 2 line, we must remember both possibilities even though the Class 0 choice has more demerits, because it might be desirable to follow this breakpoint with a tight line. On the other hand, we need not remember the Class 0 possibility if its total demerits exceed those of the Class 2 break plus the demerits for contrasting lines, since the Class 0 breakpoint will never be optimum in such a case.

More experience is needed to determine whether or not the additional computation required by this extension is worthwhile. It is comforting for the user to know that the line-breaking algorithm takes such refinements into account, but there is no point in doing the extra work if the output is hardly ever improved.

Another extension to the algorithm is needed to raise it to the highest standards of quality for hand composition: Sometimes it is desirable to set a paragraph so that it comes out one line longer or shorter than its optimum length, because this will avoid an isolated 'widow line' at the top or bottom of a page, or because it will make the total number of lines even so that the material can be divided into two equal columns. Although the paragraph itself will not be in its optimum form, the entire page will look better, and the paragraph will be set as well as possible subject to the given constraints. For example, one of the paragraphs in the story of Figure 6 has been set a line shorter than its optimum length, so that all six columns come out equal.

The line-breaking algorithm we shall describe therefore has a 'looseness' parameter, illustrated in Figure 18. The 'looseness' is an integer $q$ such that the total number of lines produced for the paragraph is as close as possible to $q$ plus the optimum number, without violating the conditions of feasibility. Figure 18 shows what happens to the example paragraph of Figure 14 when $q = -1$, 0, $+1$, and $+2$, respectively. Values of $q < -1$ would be the same as $q = -1$ since this paragraph cannot be squeezed any further, and values of $q > 5$ would be the same as $q = 5$ since the paragraph can't be stretched to more than 15 lines without having at least one line whose adjustment ratio exceeds 10. The user can get the optimum solution having fewest possible lines by setting $q$ to an extremely negative value like $-100$. When $q \neq 0$, the feasible breakpoints corresponding to different line numbers must all be remembered, even when every line has the same length.

When the lines of a paragraph are fairly loose, we don't want the last line to be noticeably different, so it is undesirable to use a 'finishing glue' with almost infinite stretchability as in our earlier remarks. The penalty for adjacent lines of contrasting classes seems to work best in connection with looseness if the finishing glue at the paragraph end is set to have a normal space equal to about half the total line width, stretching to nearly the full width and shrinking to zero.

## THE ALGORITHM ITSELF

Now let us get down to brass tacks and discuss the details of an optimum line-breaking algorithm. We are given a paragraph $x_1 \ldots x_m$ described by items $x_i = (t_i, w_i, y_i, z_i, p_i, f_i)$ as explained earlier, where $x_1$ is a box item and $x_m$ is a penalty item specifying a forced break ($p_m = -\infty$). We are also given a potentially infinite sequence of positive line lengths $l_1$, $l_2, \ldots$. There is a parameter $\alpha$ that gets added to the demerits whenever there are two consecutive breakpoints with $f_i = 1$, and a parameter $\gamma$ that gets added to the demerits whenever two consecutive lines belong to incompatible fitness classes. There is a threshold parameter $\rho$ that is an upper bound on the adjustment ratios. And there is a looseness parameter $q$.

A feasible sequence of breakpoints $(b_1, \ldots, b_k)$ is a legal choice of breakpoints such that each of the $k$ resulting lines has an adjustment ratio $r_j \leqslant \rho$. If $q = 0$, the job of the algorithm is to find a feasible sequence of breakpoints having the fewest total demerits. If $q \neq 0$, the job of the algorithm is somewhat more difficult to describe precisely; it can be formulated as follows: Let $k$ be the number of lines that the algorithm would produce when $q = 0$. Then the algorithm finds a feasible sequence of $k+q$ breakpoints having fewest total demerits. However, if this is impossible, the value of $q$ is decreased by 1 (if $q > 0$) or increased by 1 (if $q > 0$) until a feasible solution is found. Sometimes no feasible solution is possible even with $q = 0$; we will discuss this situation later after seeing how the algorithm behaves in the normal case.

We have seen that it is occasionally useful to permit boxes, glue, and penalties to have negative widths and even negative stretchability; but a completely unrestricted use of negative values leads to unpleasant complications. For reasons of efficiency, it is desirable to place two limitations on the paragraphs that will be treated:

- *Restriction 1.* Let $M_b$ be the length of the minimum-length line from the beginning of the paragraph to breakpoint $b$, namely the sum of all $w_i - z_i$ taken over all box and glue items $x_i$ for $1 \leqslant i < b$, plus $w_b$ if $x_b$ is a penalty item. The paragraph must have $M_a \leqslant M_b$ whenever $a$ and $b$ are legal breakpoints with $a < b$.
- *Restriction 2.* Let $a$ and $b$ be legal breakpoints with $a < b$, and assume that no $x_i$ in the range $a < i < b$ is a box item or a forced break (penalty $p_i = -\infty$). Then either $b = m$, or $x_{b+1}$ is a box item or a penalty with $p_{b+1} < \infty$.

Both of these restrictions are quite reasonable, as they are met by all known practical applications. Restriction 2 seems peculiar at first glance, but we will see in a moment why it is helpful.

Our algorithm has the following general outline, viewed from the top down:

```
⟨create an active node representing the beginning of the paragraph⟩;
for $b := 1$ to $m$ do ⟨if $b$ is a legal breakpoint⟩ then
  begin ⟨initialize the feasible breaks at $b$ to the empty set⟩;
  ⟨for each active node $a$⟩ do
    begin ⟨compute the adjustment ratio $r$ from $a$ to $b$⟩;
    if $r < -1$ or ⟨$b$ is a forced break⟩ then ⟨deactivate node $a$⟩;
    if $-1 \leqslant r < \rho$ then ⟨record a feasible break from $a$ to $b$⟩;
    end;
  ⟨if there is a feasible break at $b$⟩ then
    ⟨append the best such breaks as active nodes⟩;
  end;
⟨choose the active node with fewest total demerits⟩;
if $q \neq 0$ then ⟨choose the appropriate active node⟩;
⟨use the chosen node to determine the optimum breakpoint sequence⟩.
```

The meaning of the ad hoc Algol-like language used here should be self-evident. An 'active node' in this description refers to a record that includes information about a breakpoint together with its fitness classification and the line number on which it ends.

We want to have a data structure that makes this algorithm efficient, and it is not hard to design a reasonably good one, but there are two aspects in which some subtlety pays off: The operation of computing the adjustment ratio, from a given active node $a$ to a given legal breakpoint $b$, should be made as simple as possible; and there should be an easy way to determine which of the feasible breaks at $b$ ought to be saved as active nodes.

In the first place, the adjustment ratio depends on the total width, total stretchability, and total shrinkability computed from the first box after one breakpoint to the following breakpoint, and it would take too much time to compute these sums over and over. We can avoid this by computing the sums from the beginning of the paragraph to the current place, and subtracting two such sums to obtain the total of what lies between them. Let $(\Sigma w)_b$, $(\Sigma y)_b$, and $(\Sigma z)_b$ denote the respective sums of all the $w_i$, $y_i$, and $z_i$ in the box and glue items $x_i$ for $1 \leqslant i < b$. Then if $a$ and $b$ are legal breakpoints with $a<b$, the width $L_{ab}$ of a line from $a$ to $b$ and its stretchability $Y_{ab}$ and shrinkability $Z_{ab}$ can be computed as follows:

$$\begin{aligned}L_{ab} &= (\Sigma w)_b - (\Sigma w)_{\mathrm{after}(a)} + (w_b \text{ if } t_b = \text{`penalty'});\\ Y_{ab} &= (\Sigma y)_b - (\Sigma y)_{\mathrm{after}(a)};\\ Z_{ab} &= (\Sigma z)_b - (\Sigma z)_{\mathrm{after}(a)}.\end{aligned}$$

Here 'after()' is the smallest index $i > a$ such that either $i > m$ or $x_i$ is a box item or $x_i$ is a penalty item that forces a break ($p_i = -\infty$). These formulas hold even in the degenerate case that after($a$) > $b$, because of Restriction 2; in fact, Restriction 2 essentially stipulates that the relation 'after($a$) > $b$' implies that $(\Sigma w)_b = (\Sigma w)_{\mathrm{after}(a)}$, $(\Sigma y)_b = (\Sigma y)_{\mathrm{after}(a)}$, and $(\Sigma z)_b = (\Sigma z)_{\mathrm{after}(a)}$.

From these considerations, we may conclude that each node $a$ in the data structure should contain the following fields:

position($a$) = index of breakpoint represented by this node (0 = start of paragraph);
line($a$) = number of the line ending at this breakpoint;
fitness($a$) = fitness class of the line ending at this breakpoint;
totalwidth($a$) = $(\Sigma w)_{\text{after}(a)}$, used to calculate adjustment ratios;
totalstretch($a$) = $(\Sigma y)_{\text{after}(a)}$, used to calculate adjustment ratios;
totalshrink($a$) = $(\Sigma z)_{\text{after}(a)}$, used to calculate adjustment ratios;
totaldemerits($a$) = minimum total demerits up to this breakpoint;
previous($a$) = pointer to the best node for the preceding breakpoint;
link($a$) = pointer to the next node in the list.

Nodes become active when they are first created, and they become passive when they are deactivated. The algorithm maintains global variables $A$ and $P$, which point respectively to the first node in the active list and the first node in the passive list. The first step can therefore be fleshed out as follows:

```
⟨create an active node representing the beginning of the paragraph⟩ =
  begin $A$ := new node(position = 0, line = 0, fitness = 1,
                        totalwidth = 0, totalstretch = 0, totalshrink = 0,
                        totaldemerits = 0, previous = $\Lambda$, link = $\Lambda$);
  $P := \Lambda$;
  end.
```

We also introduce global variables $\Sigma W$, $\Sigma Y$, and $\Sigma Z$ to represent $(\Sigma w)_b$, $(\Sigma y)_b$, and $(\Sigma z)_b$ in the main loop of the algorithm, so that the operation '**for** $b := 1$ **to** $m$ **do** ⟨**if** $b$ is a legal breakpoint⟩ **then** ⟨main loop⟩' takes the following form:

```
$\Sigma W := \Sigma Y := \Sigma Z := 0$;
for $b := 1$ to $m$ do
  if $\mathbf{t}_b$ = 'box' then $\Sigma W := \Sigma W + w_b$
  else if $t_b$ = 'glue' then
    begin if $t_{b-1}$ = 'box' then ⟨main loop⟩;
    $\Sigma W := \Sigma W + w_b$; $\Sigma Y := \Sigma Y + y_b$; $\Sigma Z := \Sigma Z + z_b$;
    end
  else if $p_b \neq +\infty$ then ⟨main loop⟩.
```

In the main loop itself, the operation 'compute the adjustment ratio $r$ from $a$ to $b$' can now be implemented simply as follows:

```
$L := \Sigma W -$ totalwidth($a$);
if $t_b$ = 'penalty' then $L := L + w_b$;
$j :=$ line($a$) + 1;
if $L < l_j$ then
  begin $Y := \Sigma Y -$ totalstretch($a$);
  if $Y > 0$ then $r := (l_j - L)/Y$ else $r := \infty$;
  end
else if $L > l_j$ then
  begin $Z := \Sigma Z -$ totalshrink($a$);
  if $Z > 0$ then $r := (l_j - L)/Z$ else $r := \infty$;
  end
else $r := 0$.
```

The other nonobvious problem we have to deal with is caused by the fact that several nodes might correspond to a single breakpoint. We will never create two nodes having the same values of (position, line, fitness), since the whole point of our dynamic programming approach is that we need only remember the best possible way to get to each feasible break position having a given line number and a given fitness class. But it is not immediately clear how to keep track of the best ways that lead to a given position, when that position can occur with different line numbers; we could, for example, maintain a hash table with (line, fitness) as the key, but that would be unnecessarily complicated. The solution is to keep the active list sorted by line numbers: After looking at all the active nodes for line $j$, we can insert new active nodes for line $j+1$ into the list just before any active nodes for lines $\geqslant j+1$ that we are about to look at next.

An additional complication is that we don't want to create active nodes for different line numbers when the line lengths are all identical, unless $q \neq 0$, since this would unnecessarily slow the algorithm down; the complexities of the general case should not encumber the simple situations that arise most often. Therefore we assume that an index $j_0$ is known such that all breaks at line numbers $\geqslant j_0$ can be considered equivalent. This index $j_0$ is determined as follows: If $q \neq 0$, then $j_0 = \infty$; otherwise $j_0$ is as small as possible such that $l_j = l_{j+1}$ for all $j > j_0$. For example, if $q = 0$ and $l_1 = l_2 = l_3 \neq l_4 = l_5 = \cdots$, we let $j_0 = 3$, since it is unnecessary to distinguish a breakpoint that ends line 3 from a breakpoint that ends line 4 at the same position, as far as any subsequent lines are concerned.

For each position $b$ and line number $j$, it is convenient to remember the best feasible breakpoints having fitness classifications 0, 1, 2, 3 by maintaining four values $D_0$, $D_1$, $D_2$, $D_3$, where $D_c$ is the smallest known total of demerits that leads to a breakpoint at position $b$ and line $j$ and class $c$. Another variable $D = \min(D_0, D_1, D_2, D_3)$ turns out to be convenient as well, and we let $A_c$ point to the active node $a$ that leads to the best value $D_c$. Thus the main loop takes the following slightly altered form:

```
**begin** $a := A$; $preva := \Lambda$;
  **loop**: $D_0 := D_1 := D_2 := D_3 := D := +\infty$;
    **loop**: $nexta :=$ link($a$);
    ⟨compute the adjustment ratio $r$ from $a$ to $b$⟩;
    **if** $r < -1$ **or** $p_b = -\infty$ **then** ⟨deactivate node $a$⟩ **else** $preva := a$;
    **if** $-1 \leqslant r \leqslant \rho$ **then**
      **begin** ⟨compute demerits $d$ and fitness class $c$⟩;
      **if** $d < D_c$ **then**
        **begin** $D_c := d$; $A_c := a$; **if** $d < D$ **then** $D := d$;
        **end**;
      **end**;
    $a := nexta$; **if** $a = \Lambda$ **then exit loop**;
    **if** line($a$)$\geqslant j$ **and** $j < j_0$ **then exit loop**;
    **repeat**;
  **if** $D < \infty$ **then** ⟨insert new active nodes for breaks from $A_c$ to $b$⟩;
  **if** $a = \Lambda$ **then exit loop**;
  **repeat**;
**if** $A = \Lambda$ **then** ⟨do something drastic since there is no feasible solution⟩;
**end**.
```

For a given position $b$, the inner loop of this code considers all nodes $a$ having equivalent line numbers, while the outer loop runs through all of the line numbers that are not equivalent.

It is not difficult to derive a precise encoding of the operations that have been abbreviated in these loops:

```
⟨compute demerits $d$ and fitness class $c$⟩ =
  begin if $p_b \geqslant 0$ then $d := (1+100|r|^3+p_b)^2$
  else if $p_b \neq -\infty$ then $d := (1+100|r|^3)^2-p_b^2$
  else $d := (1+100|r|^3)^2$;
  $d := d+\alpha\cdot f_b\cdot f_{\text{position}(a)}$;
  if $r < -.5$ then $c := 0$
  else if $r \leqslant .5$ then $c := 1$
  else if $r \leqslant 1$ then $c := 2$ else $c := 3$;
  if $|c-\text{fitness}(a)| > 1$ then $d := d+\gamma$;
  $d := d+\text{totaldemerits}(a)$;
  end;
⟨insert new active nodes for breaks from $A_c$ to $b$⟩=
  begin ⟨compute $tw = (\Sigma w)_{\text{after}(b)}$, $ty = (\Sigma y)_{\text{after}(b)}$, and $tz = (\Sigma z)_{\text{after}(b)}$⟩;
  for $c := 0$ to 3 do if $D_c \leqslant D+\gamma$ then
    begin $s :=$ new node(position $= b$, line $= \text{line}(A_c)+1$, fitness $= c$,
                          totalwidth $= tw$, totalstretch $= ty$, totalshrink $= tz$,
                          totaldemerits $= D_c$, previous $= A_c$, link $= a$);
    if $preva = \Lambda$ then $A = d$ else link($preva$) $:= s$;
    $preva := s$;
    end;
⟨compute $tw = (\Sigma w)_{\text{after}(b)}$, $ty = (\Sigma y)_{\text{after}(b)}$, and $tz = (\Sigma z)_{\text{after}(b)}$⟩ =
  begin $tw := \Sigma W$; $ty := \Sigma Y$; $tz := \Sigma Z$; $i := b$;
    loop: if $i > m$ then exit loop;
    if $t_i =$ 'box' then exit loop;
    if $t_i =$ 'glue' then
      begin $tw := tw+w_i$; $ty := ty+y_i$; $tz := tz+z_i$;
      end
    else if $p_i = -\infty$ and $i>b$ then exit loop;
    $i := i+1$;
    repeat;
  end;
⟨deactivate node $a$⟩ =
  begin if $preva = \Lambda$ then $A := nexta$ else link($preva$) $:= nexta$;
  link($a$) $:= P$; $P := a$;
  end;
```

After the main loop has done its job, the active list will contain only nodes with position $= m$, since $x_m$ is a forced break. Thus, we can write

```
⟨choose the active node with fewest total demerits⟩ =
  begin if $a := b := A$; $d :=$ totaldemerits(a);
    loop: $a :=$ link($a$);
    if $a = \Lambda$ then exit loop;
```

```
        **if** totaldemerits($a$) $< d$ **then**
          **begin** $d :=$ totaldemerits($a$); $b := a$;
          **end**;
        **repeat**;
      $k :=$ line($b$);
      **end**.
```

Now $b$ is the chosen node and $k$ is its line number. The subsequent processing for $q \neq 0$ is equally elementary:

```
  ⟨choose the appropriate active node⟩ =
    **begin** $a := A$; $s := 0$;
      **loop**: $\delta :=$ line($a$) $- k$;
      **if** $q \leqslant \delta < s$ **or** $s < \delta \leqslant q$ **then**
        **begin** $s := \delta$; $d :=$ totaldemerits($a$); $b := a$;
        **end**
      **else if** $\delta = s$ **and** totaldemerits($a$) $< d$ **then**
        **begin** $d :=$ totaldemerits($a$); $b := a$;
        **end**;
      $a :=$ link($a$); **if** $a = \Lambda$ **then exit loop**;
      **repeat**;
    $k :=$ line($b$);
    **end**.
```

Now the desired sequence of $k$ breakpoints is accessible from node $b$:

```
  ⟨use the chosen node to determine the optimum breakpoint sequence⟩ =
    **for** $j := k$ **down to** 1 **do**
      **begin** $b_j :=$ position($b$); $b :=$ previous($b$);
      **end**.
```

(Another way to complete the processing, getting the lines in forward order from 1 to $k$ instead of from $k$ to 1, appears in the appendix below.) If there is no garbage collection, the algorithm concludes by deallocating all nodes on lists $A$ and $P$.

Note that Restriction 1 makes it legitimate to deactivate a node when we discover that $r < -1$, since $r < -1$ is equivalent to $l_l < L_{ab} - Z_{ab}$, therefore subsequent breakpoints $b' > b$ will have $L_{ab'} - Z_{ab'} \geqslant L_{ab} - Z_{ab}$. Thus it is not difficult to verify that the algorithm does indeed find an optimal solution: Given any sequence of feasible breakpoints $b_1 < \cdots < b_k$, we can prove by induction on $j$ that the algorithm constructs a node for a feasible break at $j$, with appropriate line numbers and fitness classifications, having no more demerits than the given sequence does.

There is only one loose end remaining in the algorithm, namely the operation 'do something drastic since there is no feasible solution'. As mentioned above, the TEX system assumes that the user has chosen the tolerance threshold $\rho$ in such a way that human intervention is desirable when this tolerance cannot be met. Another alternative would be to have two thresholds and to try the algorithm first with threshold $\rho_0$, which is lower than $\rho$, so the algorithm will generate comparatively few active nodes; if there is no way to succeed at tolerance $\rho_0$, the algorithm could simply return all nodes to free storage and try again with the actual threshold $\rho$. This dual-threshold method will not always find the strictly optimum feasible solution, since it is possible in unusual circumstances for the optimum solution to include a line whose adjustment ratio exceeds $\rho_0$ while there is a non-optimum feasible solution meeting the tolerance $\rho_0$; for practical purposes, however, this difference is negligible.

TEX uses a different sort of dual-threshold method. Since the task of word division is nontrivial, TEX first tries to break a paragraph into lines without any discretionary hyphens except those already present in the given text, using a tolerance threshold $\rho_1$. If the algorithm fails to find a feasible solution, or if there is a feasible solution with $q \neq 0$ but the desired looseness could not be satisfied ($\delta \neq q$), all nodes are returned to free storage and TEX starts again using another tolerance $\rho_2$. During this second pass, all words of five letters or more are submitted to TEX's hyphenation algorithm before they are treated by the line-breaking algorithm. Thus, the user sets $\rho_1$ to the limit of tolerance for paragraphs that can be completely broken without hyphenation, and $\rho_2$ is set to the tolerance limit when hyphenation must be tried; possibly $\rho_1$ will be slightly larger than $\rho_2$, but it might also be smaller, if hyphenation is not frowned on too much. (TEX users specify two integers, 'jjpar' = $\rho_1^3$ and 'jpar' = $\rho_2^3$.) In practice $\rho_1$ and $\rho_2$ are usually equal to each other, or else $\rho_1$ is near 1 and $\rho_2 \geqslant 2$; alternatively, one can take $\rho_2 = 0$ to effectively disallow hyphenation.

When both passes fail, TEX continues by reactivating the node that was most recently deactivated and treats it as if it were a feasible break leading to $b$. This situation is actually detected in the routine 'deactivate node $a$', just after the last active node has become passive:

$$\textbf{if } A = \Lambda \textbf{ and } \textit{secondpass} \textbf{ and } D = \infty \textbf{ and } r < -1 \textbf{ then } r := -1$$

The net result is to produce an 'overfull box' that sticks out into the right margin, whenever no feasible sequence of line breaks is possible. As discussed above, some kind of error indication is necessary, since the user is assumed to have set $\rho$ to a value such that further stretching is intolerable and requires manual intervention. An overfull box is easier to provide than an underfull one, by the nature of the algorithm. The setting of the overfull box will be as tight as possible, so that the user can easily see how to devise appropriate corrective action such as a forced line break or hyphenation.

## COMPUTATIONAL EXPERIENCE

The algorithm described in the previous section is rather complex, since it is intended to apply to a wide variety of situations that arise in typesetting. A considerably simpler procedure is possible for the special cases needed for word processors and newspapers; the appendix to this paper gives details about such a stripped-down version. Contrariwise, the algorithm in TEX is even more complex than the one we have described, because TEX must deal with leaders, with footnotes or cross references or page-break marks attached to lines, and with spacing both inside and immediately outside of math formulas; the spacing that surrounds a formula is slightly different from glue because it disappears when followed by a line break, but it does not represent a legal breakpoint. (A complete description of TEX's algorithm will appear elsewhere.[13]) Experience has shown that the general algorithm is quite efficient in practice, in spite of all the things it must cope with.

So many parameters are present, it is impossible for anyone actually to experiment with a large fraction of the possibilities. A user can vary the interword spacing and the penalties for inserted hyphens, explicit hyphens, adjacent flagged lines, and adjacent lines with incompatible fitness classifications; the tolerance threshold $p$ can also be twiddled, not to mention the lengths of lines and the looseness parameter $q$. Thus one could perform computational experiments for years and not have a completely definitive idea about the behavior of this algorithm. Even with fixed parameters there is a significant variation with respect to the kind of material being typeset; for example, highly mathematical copy presents special problems. An interesting comparative study of line breaking was made by Duncan et al.[8], who considered sample texts from Gibbon's *Decline and Fall* versus excerpts from a story entitled *Salar the Salmon*; as expected, Gibbon's vocabulary forced substantially more hyphenated lines.

On the other hand, we have seen that the optimizing algorithm leads to better line breaks even in children's stories where the words are short and simple, as in Grimm's fairy tales. It would be nice to have a quantitative feeling for how much extra computation is necessary to get this improvement in quality. Roughly speaking, the computation time is proportional to the number of words of the paragraph, times the average number of words per line, since the main loop of the computation runs through the currently active nodes, and since the average number of words per line is a reasonable estimate of the number of active nodes in all but the first few lines of a paragraph (see Figures 12 and 14). On the other hand, there are comparatively few active nodes on the first lines of a paragraph, so the performance is actually faster than this rough estimate would indicate. Furthermore, the special-purpose algorithm in the appendix runs in nearly linear time, independent of the line length, since it does not need to run through all of the active nodes.

Detailed statistics were kept when TEX's first large production, *Seminumerical Algorithms*[9], was typeset using the procedure above. This 700-page book has a total of 5526 'paragraphs' in its text and answer pages, if we regard displayed formulas as separators between independent paragraphs. The 5526 paragraphs were broken into a total of 21,057 lines, of which 550 (about 2·6 per cent) ended with hyphens. The lines were usually 29 picas wide, which means 626·4 machine units in 10-point type and about 677·19 machine units in 9-point type, roughly twelve or thirteen words per line. The threshold values $\rho_1$ and $\rho_2$ were normally both set to $\sqrt[3]{2} \approx 1{\cdot}26$, so the spaces between words ranged from a minimum of 4 units to a maximum of $6+3\sqrt[3]{2} \approx 9{\cdot}78$ units. The penalty for breaking after a hyphen was 50; the consecutive-hyphens and adjacent-incompatibility demerits were $\alpha = \gamma = 3000$. The second (hyphenation) pass was needed on only 279 of the paragraphs, i.e., about 5% of the time; a feasible solution without hyphenation was found in the remaining 5247 cases. The second pass would only try to hyphenate uncapitalized words of five or more letters, containing no accents, ligatures, or hyphens, and it turned out that exactly 6700 words were submitted to the hyphenation procedure. Thus the number of attempted hyphenations per paragraph was approximately 1·2, only slightly more than needed by conventional nonoptimizing algorithms, and this was not a significant factor in the running time.

The main contribution to the running time came, of course, from the main loop of the algorithm, which was executed 274,102 times (about 50 times per paragraph, including both passes lumped together when the second pass was needed). The total number of break nodes created was 64,003 (about 12 per paragraph), including multiplicities for the comparatively rare cases that different fitness classifications or line numbers needed to be distinguished for the same breakpoint. Thus, about 23% of the legal breakpoints turned out to be feasible ones, given these comparatively low values of $\rho_1$ and $\rho_2$. The inner loop of the computation was performed 880,677 times; this is the total number

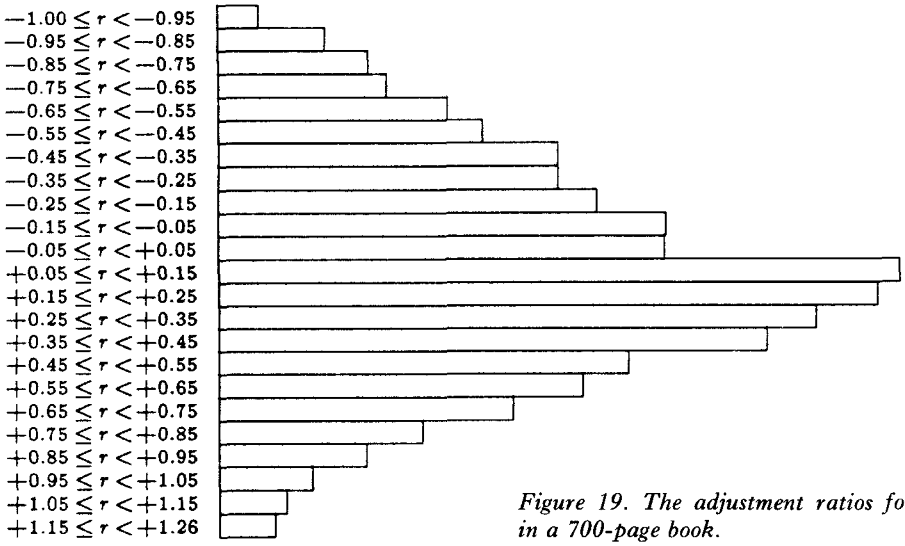


*Figure 19. The adjustment ratios for interword spaces in a 700-page book.*

of active nodes examined when each legal breakpoint was processed, summed over all legal breakpoints. Note that this amounts to about 160 active node examinations per paragraph, and 3·2 per breakpoint, so the inner loop definitely dominates the running time. If we assume that words are about five letters long, so that a legal break occurs for every six characters of input text including the spaces between words, the algorithm costs about half of an inner-loop step per character of input, plus the time to pass over that character in the outermost loop.

This source data was also used to establish the importance of the optional dominance test ‘**if** $D_c \leqslant D+\gamma$’ preceding the creation of a new node; without that test, the algorithm was found to need about 25% more executions of the inner loop, because so many unnecessary nodes were created.

And how about the output? Figure 19 shows the actual distribution of adjustment ratios $r$ in the 15,531 typeset lines of *Seminumerical Algorithms*, not counting the 5526 lines at the ends of paragraphs, for which $r \approx 0$. There was also one line with $r \approx 1{\cdot}8$ and one with $r \approx 2{\cdot}2$ (i.e., a disgraceful spacing of 12·6 units); perhaps some reader will be able to spot one or both of these anomalies some day. The average value of $r$ over all 21,057 lines was 0·08, and the standard deviation was only 0·403; about 67% of the lines had word spaces varying between 5 and 7 units. Furthermore the author believes that virtually none of the 15,531 line breaks are ‘psychologically bad’ in the sense mentioned above.

Anyone who has experience with typical English text knows that these statistics are not only excellent, they are in fact too good to be true; no line-breaking algorithm can achieve such stellar behavior without occasional assists from the author, who notices that a slight change in wording will permit nicer breaks. Indeed, this is another source of improved quality when an author is given composition tools like TEX to work with, because a professional compositor does not dare mess around with the given wording when setting a paragraph, while an author is happy to make changes that look better, especially when such changes are negligible by comparison with changes that are found to be necessary for other reasons when a draft is being proofread. An author knows that there are many ways to say what he or she wants to say, so it is no trick at all to make an occasional change of wording.

Theodore L. De Vinne, one of America's foremost typographers at the turn of the century, wrote[14] that 'when the author objects to [a hyphenation] he should be asked to add or cancel or substitute a word or words that will prevent the breakage... Authors who insist on even spacing always, with sightly divisions always, do not clearly understand the rigidity of types.' Another interesting comment was made by G. B. Shaw[15]: 'In his own works, whenever [William Morris] found a line that justified awkwardly, he altered the wording solely for the sake of making it look well in print. When a proof has been sent to me with two or three lines so widely spaced as to make a grey band across the page, I have often rewritten the passage so as to fill up the lines better; but I am sorry to say that my object has generally been so little understood that the compositor has spoilt all the rest of the paragraph instead of mending his former bad work.'

The bias caused by Knuth's tuning his manuscript to a particular line width makes the statistics in Figure 19 inapplicable to the printer's situation where a given text must be typeset as it is. So another experiment was conducted in which the material of Section 3.5 of *Seminumerical Algorithms* was set with lines 25 picas wide instead of 29 picas. Section 3.5, which deals with the question 'What is a random sequence?', was chosen because this section most closely resembles typical mathematics papers containing theorems, proofs, lemmas, etc. In this experiment the optimum-fit algorithm had to work harder than it did when the material was set to 29 picas, primarily because the second pass was needed about thrice as often (49 times out of 273 paragraphs, instead of 16 times); furthermore the second pass was much more tolerant of wide spaces ($\rho_2 = 10$ instead of $\sqrt[3]{2}$), in order to guarantee that every paragraph could be typeset without manual intervention. There were about 6 examinations of active nodes per legal breakpoint encountered, instead of about 3, so the net effect of this change in parameters was to nearly double the running time for line breaking. The reason for such a discrepancy was primarily the combination of difficult mathematical copy and a narrower column measure, rather than the 'author tuning', because when the same text was set 35 picas wide the second pass was needed only 8 times.

It is interesting to observe the quality of the spacing obtained in this 25-pica experiment, since it indicates how well the optimum-fit method can do without any human intervention. Figure 20 shows what was obtained, together with the corresponding statistics for the best-fit method when it was applied to the same data. About 800 line breaks were involved in each case, not counting the final lines of paragraphs. The main difference was that optimum-fit tended to put more lines into the range $\cdot 5 \leqslant r \leqslant 1$, while best-fit produced considerably more lines that were extremely spaced out. The standard deviation of spacing was 0·53 (optimum-fit) versus 0·65 (best-fit); 24 of the lines typeset by best-fit had spaces exceeding 12 units, while only 7 such bad lines were produced by the optimum-fit method. An examination of these seven problematical cases showed that three of them were due to long unbreakable formulas embedded in the text, three were due to the rule that TEX does not try to hyphenate capitalized words, and the other one was due to TEX's inability to hyphenate the word 'reasonable'. Cursory inspection of the output indicated that the main difference between best-fit and optimum-fit, in the eyes of a casual reader, would be that the best-fit method not only resorted to occasional wide spacing, it also tended to end substantially more lines with hyphens: 119 by comparison with 80. An author who cares about spacing, and

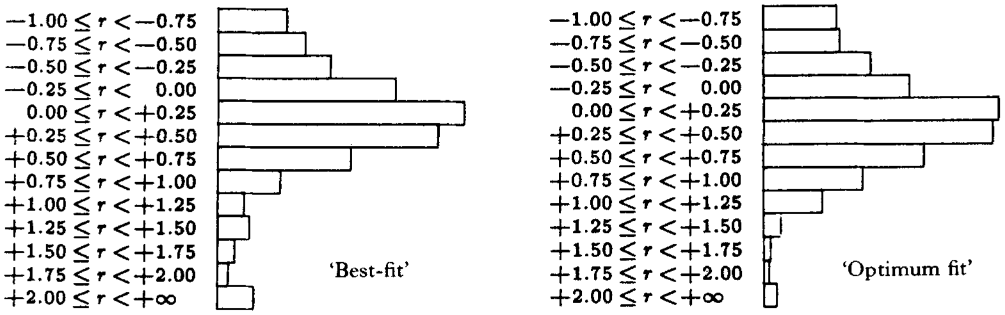


*Figure 20. The distribution of interword spaces found by the best line-at-a-time method, compared to the distribution found by the best paragraph-at-a-time method, when difficult mathematical copy is typeset without human intervention.*

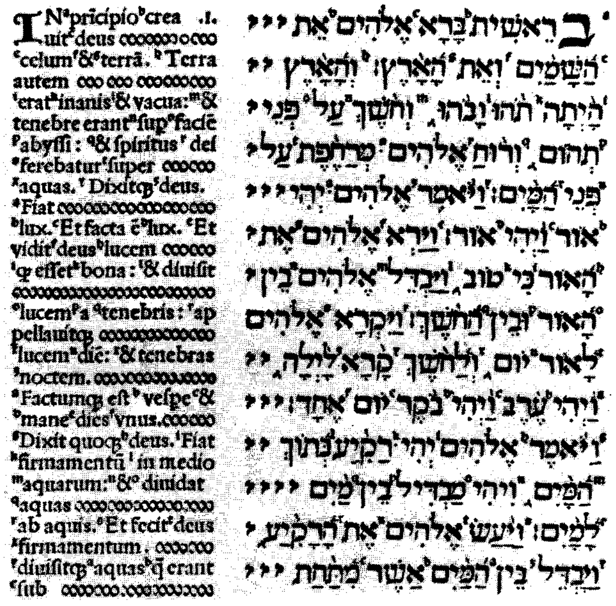
who therefore will edit a manuscript until it can be typeset satisfactorily, would have to do a significant amount of extra work in order to get the best-fit method to produce decent results with such difficult copy, but the output of the optimum-fit method could be made suitable with only a few author's alterations.

## A HISTORICAL SUMMARY

We have now discussed most of the issues that arise in line breaking, and it is interesting to compare the newfangled approaches to what printers have actually been doing through the years. Medieval scribes, who prepared beautiful manuscripts by hand before the days of printing, were generally careful to break lines so that the right-hand margins would be nearly straight, and this practice was continued by the early printers. Indeed, printers had to fill up each line of type with spaces anyway, so that the individual letters wouldn't fall out of position while making impressions, and it wasn't too much more difficult for a compositor to distribute the spaces between words instead of putting them at the ends of lines.

One of the most difficult challenges faced by printers over the years has been the typesetting of 'polyglot Bibles'—editions of the Bible in which the original languages are set side by side with various translations—since special care is needed to keep the versions of various languages synchronized with each other. Furthermore the fact that several languages appear on each page means that the texts tend to be set with narrower columns than usual; this, together with the fact that one dare not alter the sacred words, makes the line-breaking problem especially difficult. We can get a good idea of the early printers' approaches to line breaking by examining their polyglot Bibles carefully.

The first polyglot Bible[16,17,18] was produced in Spain by the eminent Cardinal Jiménez de Cisneros, who reportedly spent 50,000 gold ducats to support the project. It is generally called the *Complutensian Polyglot*, because it was prepared in Alcalá de Henares, a city near Madrid whose old Roman name was Complutus. The printer, Arnao Gúillen de Brocar, devoted the years 1514–1517 to the production of this six-volume set, and it is said that the Hebrew and Greek fonts he made for the occasion are among the finest ever cut. His approach to justification was quite interesting and unusual, as shown in Figure 21: Instead of justifying the lines by increasing the word spaces, he inserted visible leaders to obtain solid blocks of copy with straight margins.



IN principio crea .i.
uit deus
celum & terrã. Terra
autem
erat inanis & vacua: &
tenebre erant sup faciẽ
abyssi: & spiritus dei
ferebatur super
aquas. Dixitq̃ deus.
Fiat
lux. Et facta ẽ lux. Et
vidit deus lucem
q̃ esset bona: & diuisit

lucem a tenebris: ap
pellauitq̃
lucem diẽ: & tenebras
noctem.
Factumq̃ est vespe &
mane dies vnus.
Dixit quoq̃ deus. Fiat
firmamentũ in medio
aquarum: & diuidat
aquas
ab aquis. Et fecit deus
firmamentum.
diuisitq̃ aquas q̃ erant
sub

בראשית ברא אלהים את
השמים ואת הארץ: והארץ
היתה תהו ובהו וחשך על פני
תהום ורוח אלהים מרחפת על
פני המים: ויאמר אלהים יהי
אור ויהי אור: וירא אלהים את
האור כי טוב ויבדל אלהים בין
האור ובין החשך: ויקרא אלהים
לאור יום ולחשך קרא לילה
ויהי ערב ויהי בקר יום אחד:
ויאמר אלהים יהי רקיע בתוך
המים ויהי מבדיל בין מים
למים: ויעש אלהים את הרקיע
ויבדל בין המים אשר מתחת

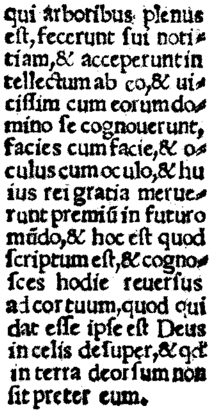
*Figure 21. The opening verses of Genesis as typeset in the Complutensian Polyglot Bible; the Latin words are keyed to the Hebrew, and leaders are used to fill out lines that would otherwise be ragged right and ragged left. Greek and Chaldee (Aramaic) versions of the text also appeared on the same page.*

These leaders appear at the right of the Latin lines and at the left of the Hebrew lines. He changed this style somewhat after gaining more experience: Starting at about the 46th chapter of Genesis, the Hebrew text was justified by word spaces, although the leaders continued to appear in the Latin column. It is clear that straight margins were considered strongly desirable at the time.

Brocar's method of line breaking seems to be essentially a first-fit approach to the Hebrew text; the corresponding Latin translation could then be set up rather easily, since there were two lines of Latin for each line of Hebrew, and this gave plenty of room for the Latin. In some cases when the Greek text was abnormally long by comparison with the corresponding Hebrew (e.g., Exodus 38), Brocar set the Hebrew quite loosely, so it is evident that he gave considerable attention to line breaking.

At about the same time, a polyglot version of the book of Psalms was being prepared as a labor of love by Agostino Giustiniani of Genoa.[19] This was the first polyglot book actually to appear in print with each language in its own characters, although Origen's third-century *Hexapla* manuscript is generally considered to be the inspiration for all of the later polyglot volumes. Giustiniani's Psalter had eight columns: (1) The Hebrew original; (2) A literal Latin rendition of (1); (3) The common Latin (Vulgate) version; (4) The Greek (Septuagint) version; (5) The Arabic version; (6) The Chaldee version; (7) A literal Latin translation of (6); (8) Notes. Since the Psalms are poems, all of the columns except the last were set with ragged margins, and an interesting convention was used to deal with the occasional line that was too wide to fit: A left parenthesis was placed at the very end of the broken line, and the remainder of that line (preceded by another left parenthesis) was placed flush with the margin of the preceding or following line, wherever it would fit.

Only column (8) was justified, and it had a rather narrow measure of about 21 characters per line. By studying this column we can conclude that Giustiniani did not take great pains to make equal spacing by fiddling with the words. For example, Figure 22,

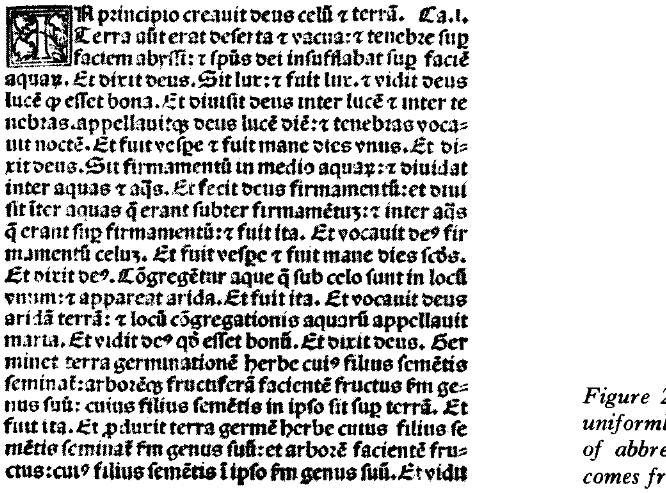


qui arboribus plenus
eſt, fecerunt ſui noti-
tiam,& acceperunt in
tellectum ab eo,& ui-
ciſſim cum eorum do-
mino ſe cognouerunt,
facies cum facie,& o-
culus cum oculo,& hu
ius rei gratia merue-
runt premiũ in futuro
mũdo,& hoc eſt quod
ſcriptum eſt,& cogno-
ſces hodie reuerſus
ad cor tuum,quod qui
dat eſſe ipſe eſt Deus
in celis deſuper,& qđ
in terra deorſum non
ſit preter eum.

*Figure 22. Part of Giustiniani's commentary on the Psalms. The presence of a loose line surrounded by two very tight lines indicates that the compositor did not go back to reset previous lines when a problem arose.*

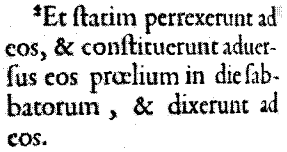
which comes from the notes on Psalm 6, shows two very tight lines enclosing a very loose one in the passage 'scriptum est . . . quod qui'. If Giustiniani had been extremely concerned about spacing he would have used the hyphenation 'cog-nosces'; the other potential solution, to move 'ad' up a line, would not have worked since there isn't quite room for 'ad' on the loose line. Notice that another aid to line breaking in Latin at that time was to replace an *m* or *n* by a tilde on the previous vowel (e.g., 'premiũ' for premium and 'mũdo' for mundo); an extension to the box/glue/penalty algebra would be needed to include such options in TEX's line-breaking algorithm! It is not clear why Giustiniani didn't set 'acceperũt' on the third line, to save space, since he had no room for the hyphen of 'in-tellectum'; perhaps he didn't have enough ũ's left in his type case.

Figure 23 shows some justified text from the Complutensian polyglot, taken from the Latin translation of an early Aramaic translation of the original Hebrew. The compositor was somewhat miraculously able to maintain this uniformly tight spacing throughout the entire volume, by making use of abbreviations and frequent hyphenations. Note that, as in Figure 22, the hyphen was omitted from a broken word when there was no room for it; e.g., 'diuisit' has been divided without a hyphen.



In principio creauit deus celũ ⁊ terrã. Ca.i.
Terra aũt erat deſerta ⁊ vacua: ⁊ tenebre ſuꝑ
faciem abyſſi: ⁊ ſpũs dei inſufflabat ſuꝑ faciẽ
aquaꝝ. Et dixit deus. Sit lux: ⁊ fuit lux. ⁊ vidit deus
lucẽ ꝙ eſſet bona. Et diuiſit deus inter lucẽ ⁊ inter te
nebras. appellauitq; deus lucẽ diẽ: ⁊ tenebras voca-
uit noctẽ. Et fuit veſpe ⁊ fuit mane dies vnus. Et di-
xit deus. Sit firmamentũ in medio aquaꝝ: ⁊ diuidat
inter aquas ⁊ aq̃s. Et fecit deus firmamentũ: et diui
ſit īter aquas q̃ erant ſubter firmamētuꝫ: ⁊ inter aq̃s
q̃ erant ſuꝑ firmamentũ: ⁊ fuit ita. Et vocauit deꝰ fir
mamentũ celuꝫ. Et fuit veſpe ⁊ fuit mane dies ſcđs.
Et dixit deꝰ. Cōgregētur aque q̃ ſub celo ſunt in locũ
vnum: ⁊ appareat arida. Et fuit ita. Et vocauit deus
aridã terrã: ⁊ locũ cōgregationis aquarũ appellauit
maria. Et vidit deꝰ qđ eſſet bonũ. Et dixit deus. Ger
minet terra germinationẽ herbe cuiꝰ filius ſemẽtis
ſeminat: arboreq; fructiferã facientẽ fructus ſm ge-
nus ſuũ: cuius filius ſemẽtis in ipſo ſit ſuꝑ terrã. Et
fuit ita. Et ꝓduxit terra germẽ herbe cuius filius ſe
mẽtis ſeminat ſm genus ſuũ: et arborẽ facientẽ fru-
ctus: cuiꝰ filius ſemẽtis ĩ ipſo ſm genus ſuũ. Et vidit

*Figure 23. Early printing of Latin texts featured uniformly tight spacing, obtained by frequent use of abbreviations and word division. This sample comes from the same page as Figure 21.*



32 *Et ſtatim perrexerunt ad
eos, & conſtituerunt aduer-
ſus eos prœlium in die ſab-
batorum , & dixerunt ad
eos.

*Figure 24. The Latin version of 1 Maccabees 2:32 from Plantin's Royal Polygot of Antwerp, showing how the second-last line of a paragraph was spaced out in order to add a line. (The copy is distorted at the right of this illustration, because the pages of this rare book cannot be laid flat without harming its binding.)*

The next great polyglot Bible was the *Royal Polyglot of Antwerp*,[20] produced during 1568–1572 by the outstanding printer Christophe Plantin. Numerous copies of the Complutensian Polyglot had unfortunately been lost at sea, so King Phillip II commissioned a new edition that would also take advantage of recent scholarship. Plantin was a pious man who was active in pacifist religious circles and anxious to undertake the job; but when he had completed the work he described it as an 'indescribable toil, labor, and expense.' On June 9, 1572, Plantin sent a letter to one of his friends, saying 'I am astonished at what I undertook, a task I would not do again even if I received 12,000 crowns as a gift.' But at least his work was widely appreciated: Lucas of Bruges, writing in 1577, said that 'the art of the printer has never produced anything nobler, nor anything more splendid.'

Most of Plantin's polyglot Bible was justified with fairly wide columns having about 42 characters per line, so it did not present especially difficult problems of line breaking. But we can get some idea of his methods by studying the texts of the Apocrypha, which were set with a narrower measure of about 27 characters per line. He arranged things so that each column on a page would have about the same number of lines, even though the individual columns were in different languages. Figure 24 shows an example of a passage excerpted from a page where the Latin text was comparatively sparse, so the paragraphs on that page needed to be rather loose. It appears that the entire page was set first, then adjustments were made after the Latin column was found to be too short; in this case the word 'eos' was brought down to make a new line and the previous line was spaced out. Plantin's compositor did not take the trouble to move 'sab-' down to that line, although such a transposition would have avoided a hyphen without making the spacing any worse. The optimum solution would have been to avoid this hyphenation and to hyphenate the previous line after 'ad-', thus achieving fairly uniform spacing throughout.

The most accurate and complete of all polyglot Bibles was the *London Polyglot*,[21] printed by Thomas Roycroft and others during the Cromwellian years 1653–1657. This massive 8-volume work included texts in Hebrew, Greek, Latin, Aramaic, Syriac, Arabic, Ethiopic, Samaritan, and Persian, all with accompanying Latin translations, and it has been acclaimed as 'the typographical achievement of the seventeenth century.' As in Plantin's work shown in Figure 24, a paragraph that has been loosened will often end with an unnecessarily tight hyphenated line followed by a loose line followed by a one-word line; so it is clear that Roycroft's compositors did not have time to do complex adjustments of line breaks.

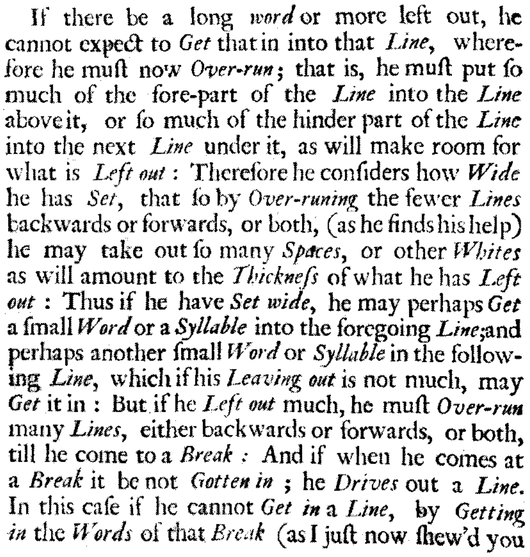
Hyphenations were clearly not frowned upon at the time, since about 40% of all lines in the London Polyglot end with a hyphen, regardless of the column width. It is not difficult to find pages on which hyphenated lines outnumber the others; and in the Latin translation of the Aramaic version of Genesis 4:15, even the two-letter word 'e-o' was hyphenated! Such practice was not uncommon: for example, the Hamburg Polyglot Bible[22] of 1596 had more than 50% hyphens at the right margin. Both Plantin's polyglot and the notes of Giustiniani's Psalter had hyphenation percentages of about 40%, and the same was true of many medieval manuscripts. Thus it was considered better to have the margins straight and to keep the spacing tight, rather than to avoid word splits.

One of the first things that strikes a modern eye when looking at these old Bibles is the treatment of punctuation. Note, for example, that no space appears after the commas in Figure 22, and a space appears *before* as well as after one of the commas in Figure 24. One can find all four possibilities of 'space before/no space before' and 'space after/no space after' in each of the Bibles mentioned so far, with respect to commas, periods, colons, semicolons, and question marks, and with no apparent preference between the four choices except that it was comparatively rare to put a space before a period. Giustiniani and Plantin occasionally would insert spaces before periods, but Roycroft apparently never did. Commas began to be treated like periods in this respect about 1700, but colons and semicolons were generally both preceded and followed by spaces until the 19th century. Such extra spaces were helpful in justifying, of course, and it was also helpful to have the option of leaving out all of the space next to a punctuation mark. Roycroft would in fact eliminate the space between words when necessary, if the following word was capitalized (e.g., 'dixitDeus'); apparently a printer's main goal was to keep the text unambiguously decipherable, while ease of readability was only of secondary importance.

Knowledge about how to carry out the work of a trade like printing was originally passed from masters to apprentices and not explained to the general public, so we can only guess at what the early printers did by looking at their finished products. A trend to put trade secrets into print was developing during the 17th century, however,[23] and a book about how to make books was finally written: Joseph Moxon's *Mechanick Exercises*,[24] published in 1683, was by forty years the earliest manual of printing in any language. Although Moxon did not discuss rules for hyphenation and punctuation, he gave interesting information about line breaking and justification.

'If the *Compositor* is not firmly resolv'd to keep himself strictly to the Rules of good Workmanship, he is now tempted to make *Botches*...', namely bad line breaks, according to Moxon. The normal 'thick space' between words, when beginning to make up a line, was one-fourth of what Moxon called the body size (one em), and he also spoke of 'thin spaces' that were one-seventh of the body size; thus, a printer who followed this practice would deal mostly with spaces of 4·5 units and 2·57 units, although these measurements were only approximate because of the primitive tools used at the time. Moxon's procedure for justifying a line whose natural width was too narrow was to insert thin spaces between one or more words to 'fill up the Measure pretty stiff,' and if necessary to go back through the line and do this again. 'Strictly, good Workmanship will not allow more [than the original space plus two thin spaces], unless the *Measure* be so short, that by reason of few *Words* in a *Line*, necessity compells him to put more *Spaces* between the the *Words*... These wide *Whites* are by *Compositers* (in way of Scandal) call'd *Pidgeon-holes*.... And as *Lines* may be too much *Spaced-out*, so may they be too close Set.'

Notice that Moxon's justification procedure would normally leave uneven spacing between words on the same line, since he inserts the thin spaces one by one. In fact, such discrepancies were the norm in early printed books, which look something like present-day attempts at justification on a typewriter or computer terminal with fixed-width spacing. For example, the relative proportions in the spaces of the third line of Plantin's text in Figure 24 are approximately 8 : 12 : 5 : 9 : 4, and in the fifth line of Giustiniani's Figure 22 they are approximately 3 : 2 : 1. Moxon's book itself (see



If there be a long *word* or more left out, he
cannot expect to *Get* that in into that *Line*, wherefore
he muſt now *Over-run*; that is, he muſt put ſo
much of the fore-part of the *Line* into the *Line*
above it, or ſo much of the hinder part of the *Line*
into the next *Line* under it, as will make room for
what is *Left out* : Therefore he conſiders how *Wide*
he has *Set*, that ſo by *Over-runing* the fewer *Lines*
backwards or forwards, or both, (as he finds his help)
he may take out ſo many *Spaces*, or other *Whites*
as will amount to the *Thickneſs* of what he has *Left
out* : Thus if he have *Set wide*, he may perhaps *Get*
a ſmall *Word* or a *Syllable* into the foregoing *Line*;and
perhaps another ſmall *Word* or *Syllable* in the following
*Line*, which if his *Leaving out* is not much, may
*Get* it in : But if he *Left out* much, he muſt *Over-run*
many *Lines*, either backwards or forwards, or both,
till he come to a *Break* : And if when he comes at
a *Break* it be not *Gotten in* ; he *Drives* out a *Line*.
In this caſe if he cannot *Get in* a *Line*, by *Getting
in* the *Words* of that *Break* (as I juſt now ſhew'd you

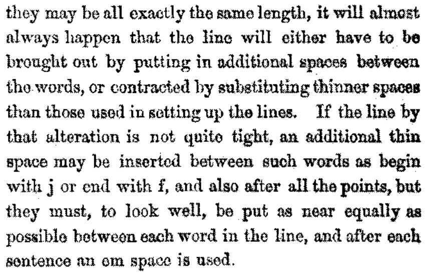
*Figure 25. An excerpt from page 245 of Joseph Moxon's 'Mechanick Exercises,' vol. 2, the first book about how printing is done. Moxon is describing the process of making corrections to pages that have already been typeset; the irregular spacing found throughout his book is probably due in part to the fact that such corrections are necessary.*

Figure 25) shows extreme variations, frequently breaking the rules he had stated for maximum and minimum spaces between words.

It would be nice to report that Moxon described a particular line-breaking algorithm, like the first-fit or best-fit method, but in fact he never suggested any particular procedure, nor did any of his successors until the computer age; this is not surprising, since people were just expected to use their common sense instead of to obey some rigid rules. Many of the breaks in Figure 25 can, however, be accounted for by assuming an underlying first-fit algorithm. For example, the looseness on lines 1, 4, and 8 is probably due to the long words at the beginning of lines 2, 5, and 9, since these long words would not fit on the previous line unless they were hyphenated. On the other hand, the extremely tight spacing on line 13 can best be explained by assuming that one or more words had to be inserted to correct an error after the page had been set. Thus we cannot satisfactorily infer the compositor's procedure from the final copy, we really need to see the first trial proofs. All we can conclude for certain is that there was very little attempt to go back and reconsider the already-set lines unless it was absolutely necessary to do so; for example, this paragraph would have been better if the first line had ended with 'can-' and the second with 'wherefore'.

Moxon's compositor was, however, supposed to look ahead: 'When in *Composing* he comes near a *Break* [i.e., the end of a paragraph], he for some *Lines* before he comes to it considers whether that *Break* will end with some reasonable *White*; If he finds it will, he is pleas'd, but if he finds he shall have but a single *Word* in his *Break*, he either *Sets* wide to drive a Word or two more into the *Break-line*, or else he *Sets* close to get in that little Word, because a *Line* with only a little Word in it, shews almost like a *White-line*, which unless it be properly plac'd, is not pleasing to a curious Eye.'

Another extract from a London printing manual[25] is shown in Figure 26; this one is from 1864 instead of 1683. Although the author says that the justifying spaces are to be made as nearly equal as possible, whoever did the composition of his book did not follow the instructions it contains! Only one of the fine books considered above has spaces that look the same, namely the Complutensian Polyglot. In fact, printers only rarely achieved truly uniform spacing until machines like the Monotype



they may be all exactly the same length, it will almost always happen that the line will either have to be brought out by putting in additional spaces between the words, or contracted by substituting thinner spaces than those used in setting up the lines. If the line by that alteration is not quite tight, an additional thin space may be inserted between such words as begin with j or end with f, and also after all the points, but they must, to look well, be put as near equally as possible between each word in the line, and after each sentence an em space is used.

*Figure 26. Printers do not always practice what they preach.*

and Linotype made the task easier towards the end of the nineteenth century; and these new machines, with their emphasis on speed, changed the philosophy of justification so much that the quality of line breaking decreased when the spacing became uniform: It became too inconvenient for the compositor to go back and reconsider any of the earlier line breaks of a paragraph, when he was expected to turn out so many more ems of type per hour.

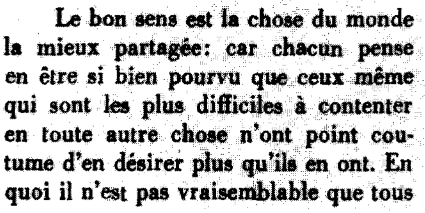
The line breaks in Figure 26 are fairly well done in spite of the uneven spacing, given that the compositor wished to avoid hyphenations and the psychologically bad break in the phrase 'with j'; it would have been slightly better, however, to move the word 'but' down to the third-last line.

Probably the most beautiful spacing ever achieved in any typeset book appeared in *The Art of Spacing*[26] by Samuel A. Bartels (1926). This book was hand set by the author, and it contains about 50 characters per line. There are no loose lines, and no hyphenated words; the final line of each paragraph always fills at least 65% of the column width, yet ends at least one em from the right margin. Bartels must have changed his original wording many times in order to make this happen; the author as compositor is clearly able to enhance the appearance of a book.

General-purpose computers were first applied to typesetting by Georges P. Bafour, André R. Blanchard, and François H. Raymond in France, who applied for patents on their invention in 1954. (They received French and British patents in 1955, and a U.S. patent in 1956.[27, 28]) This system gave special attention to hyphenation, and its authors were probably the first to formulate the method of breaking one line at a time in a systematic fashion. Figure 27 shows a specimen of their output, as demonstrated at the Imprimerie Nationale in 1958. In this example the word 'en' was not included in the second line because their scheme tended to favor somewhat loose lines: Each line would contain as few characters as possible subject to the condition that the line was feasible but the addition of the next $K$ characters would not be feasible; here $K$ was a constant, and their method was based on a $K$-stage lookahead.

Michael P. Barnett began to experiment with computer typesetting at M.I.T. in 1961, and the work of his group at the Cooperative Computing Laboratory was destined to become quite influential in the U.S.A. For example, the TROFF system[29] that is now in use at many computer centers is a descendant of Barnett's PC6 system[1], via other systems called RUNOFF and NROFF. Another line of descent is represented by the PAGE-1, PAGE-2, and PAGE-3 systems, which have been used extensively in the typesetting industry.[30, 31, 32] All of these programs use the first-fit method of line breaking that is described above.

At about the same time that Barnett began his M.I.T. studies of computer typesetting, another important university research project with similar goals was started by John Duncan at the University of Newcastle-Upon-Tyne Computing Laboratory.



Le bon sens est la chose du monde
la mieux partagée: car chacun pense
en être si bien pourvu que ceux même
qui sont les plus difficiles à contenter
en toute autre chose n'ont point cou-
tume d'en désirer plus qu'ils en ont. En
quoi il n'est pas vraisemblable que tous

*Figure 27. This is a specimen of the output produced in 1958 by the first computer-controlled typesetting system in which all of the line breaks were chosen automatically.*

Line breaking was one of the first subjects studied intensively by this group, and they developed a program that would find a feasible way to typeset a paragraph without hyphenations, if any sequence of feasible breaks exists, given minimum and maximum values for interword spaces. This program essentially worked by backtracking through all possibilities, treating them in reverse lexicographic order (i.e., starting with the first breakpoint $b_1$ as large as possible and using the same method recursively to find feasible breaks $(b_2, b_3, \ldots)$ in the rest of the paragraph, then decreasing $b_1$ and repeating the process if necessary). Thus it would either find the lexicographically largest feasible sequence of breakpoints or it would conclude that none are feasible; in the latter case hyphenation was attempted. This was the first systematic sequence of experiments to deal with the line-breaking problem by considering a paragraph as a whole instead of working line by line.

No distinction was made in these early experiments between one sequence of feasible breakpoints and another; the only criterion was whether or not all interword spacing could be confined to a certain range without requiring hyphenation. Duncan found that when lines were 603 units wide, it was possible to avoid virtually all hyphenations if spaces were allowed to vary between 3 and 12 units; with 405-unit lines, however, hyphens were necessary about 3% of the time in order to keep within these fairly generous limits, and when the line width decreased to 288 units the hyphenation percentage rose to 12% or 16% depending on the difficulty of the copy being typeset. More stringent intervals, such as the requirement of 4- to 9-unit spaces used in most of the examples we have been considering above, were found to need more than 4% hyphenations on 603-unit lines and 30% to 40% on 288-unit lines. However, these numbers are higher than necessary because the Newcastle program did not search for the best places to insert hyphens: Whenever it was unfeasible to set more than $k$ lines, the $(k+1)$st line was simply hyphenated and the process was restarted. One hyphen generated by this method tends to spawn more in the same paragraph, since the first line of a paragraph or of an artificially resumed paragraph is the most likely to require hyphenation. Examples of the performance can be seen in the article where the method was introduced[8] (using spaces of 4 to 15 units for the first six pages and 4 to 12 units for the rest), as well as in Duncan's survey paper.[2] These articles also discuss possible refinements to the method, one idea being to try to avoid loose lines next to tight lines in some unspecified manner, another being to try the method first with strict spacing intervals and then to increase the tolerance before resorting to hyphenation.

Such refinements were carried considerably further by P. I. Cooper[33] at Elliott Automation, who developed a sophisticated experimental system for dealing with entire paragraphs. Cooper's system worked not only with minimum and maximum spacing parameters, it also divided the permissible interword spaces into different sectors that yielded different so-called 'penalty scores'. Besides the penalties associated with the spaces on individual lines, there were additional penalty scores based on the respective spacing sectors of two consecutive lines, and the goal was to minimize the total penalty needed to typeset a given paragraph. Thus, his model was rather similar to the TEX model that we have been discussing, except that all spaces were equivalent to each other and special problems like hyphenation were not treated.

Cooper said that his program 'employs a mathematical technique known as "dynamic programming"' to select the optimum setting. However, he gave no details, and from the stated computer memory requirements it appears that his algorithm was only an approximation to true dynamic programming in that it would retain just one optimum sum-of-penalties for each breakpoint, not for each (breakpoint, sector) pair. Thus, his algorithm was probably similar to the method given in the appendix below.

Unfortunately, Cooper's method was ahead of its time; the consensus in 1966 was that such additional computer time and memory space were prohibitively expensive. Furthermore his method was evaluated only on the basis of how many hyphens it would save, not on the better spacing it provided on non-hyphenated lines. For example, J. L. Dolby's notes on this paper[34] compared Cooper's procedure unfavorably to Duncan's since the Newcastle method removed the same number of hyphens with what appeared to be a less complex program. In fact, Cooper himself undersold his scheme with unusual modesty and caution when he spoke about it: He said 'this investigation does not support the view that [my approach] should be given a general and enthusiastic recommendation.... It has to be admitted that an aesthetic improvement is neither predictable nor measurable.' His method was soon forgotten.

In retrospect we can see that the defect in Cooper's otherwise admirable approach was the way it dealt with hyphenation: No proper tradeoff between hyphenated lines and feasible unhyphenated lines was made, and the method would be restarted after every hyphen had to be inserted. Thus, the hyphens tended to cluster as in the Newcastle experiments.

Another approach to line breaking has recently been investigated by A. M. Pringle of Cambridge University, who devised a procedure called *Juggle*.[35] This algorithm uses the best-fit method without hyphenation until reaching a line that cannot be accommodated; then it calls a recursive procedure *pushback* that attempts to move a word from the offending line up into the previous text. If *pushback* fails to solve the problem, another recursive routine *pullon* tries to move a word forward from the previous text; hyphenation is attempted only if *pullon* fails too. Thus, *Juggle* attempts to simulate the performance of a methodical super-conscientious workman in the good olde days of hand composition. The recursive backtracking can, however, consume a lot of time by comparison with a dynamic programming approach, and an optimum sequence of line breaks is not generally achieved; for example, Figure 2 would be obtained instead of Figure 3. Furthermore there are unusual cases in which feasible solutions exist but *Juggle* will not find them; for example, it may be feasible to push back two words but not one.

Hanan Samet has suggested another measure of optimality in his recent work on line breaking.[36] Since all methods for setting a paragraph in a given number of lines involve the same total amount of blank space, he points out that the average interword space in a paragraph is essentially independent of the breakpoints (if we ignore the fact that the final line is different). Therefore he suggests that the *variance* of the interword spaces should be minimized, and he proposes a 'downhill' algorithm that shifts words between lines until no such local transformation further reduces the variance.

The first magazine publisher to develop computer aids to typesetting was Time Inc. of New York City, whose line-breaking decisions went largely on-line in 1967. According to comments made by H. D. Parks[37] at the time, line breaks were determined one by one using a variation of the first-fit algorithm that we might call 'tight-fit'; this gives the most words per line except that hyphenation is done only when necessary, and it is equivalent to the first-fit method if the normal interword spacing is the same as the minimum. The tight-fit method had previously been used on the IBM 1620 Type Composition System demonstrated in 1963 (see Duncan,[2] pages 159–160), and it is reasonable to suppose that essentially the same method was carried over to the Time group when they dedicated two IBM 360/40 computers to the typesetting task.[38]

Since the final copy in *Time* magazine has been edited and re-edited, and since manual intervention and last-minute corrections will change line-breaking decisions, it is impossible to deduce what algorithm is presently used for *Time* articles merely by examining the printed pages; but it is tempting to speculate about how the optimum-fit algorithm might improve the appearance of such publications. Figure 28 on the next page shows an interesting example based on page 22 of *Time* magazine dated June 23, 1980; Version A shows the published spacing and Version B shows what the new algorithm would produce in the same circumstances. All letters of the text have been replaced by n's of the corresponding width, so that it is possible to concentrate solely on the spacing; however, it should be pointed out that this device makes bad spacing look more innocuous, since a reader isn't so annoyingly distracted when no semantic meaning is present anyway.

The most interesting thing about Figure 28 is that the final line of the first paragraph was brought flush right in order to balance the inserted photograph properly; this photograph actually carried over into the right-hand column. Version A shows how the desired effect was achieved by stretching the final three lines, leaving large gaps that surely caught the curious eye of many a reader; Version B shows how the optimizing algorithm is magically able to look ahead and make things come out perfectly. Perhaps even more important is the fact that Version B avoids the need for letterspacing that spoiled the appearance of lines 6, 9, 10, 23, and 32 in Version A.

Letterspacing—the insertion of tiny spaces between the letters of a word so as to make large interword spaces less prominent—could readily be incorporated into the box/glue/penalty model, but it is almost universally denounced by typographers. For example, De Vinne[14] said that letterspacing is improper even when the columns are so narrow that some lines must contain only a single word; Bruce Rogers[39] said 'it is preferable to put all the extra space between the words even though the resultant "holes" are distressing to the eye.' Even one-fourth of a unit of space between letters makes the word look noticeably different. According to the style rules of the U.S. *Congressional Record*[40], 'In general, operators should avoid wide spacing. However, no letterspacing is permitted.' The optimum-fit algorithm therefore makes it possible to comply more easily with existing laws.

The idea of applying dynamic programming to line breaking occurred to D. E. Knuth in 1976, when Professor Leland Smith of Stanford's music department raised a related question that arises in connection with the layout of music on a page (see Clancy and Knuth[41]). During a subsequent discussion with students in a problem-solving seminar, someone pointed out that essentially the same idea would apply to the texts of paragraphs as well as to music. The box/glue/penalty model was developed by Knuth in April 1977 when the initial design of TEX was made, although it wasn't clear at that time whether a general optimizing algorithm could be implemented with enough efficiency for practical use. Knuth was blissfully unaware of Cooper's supposedly

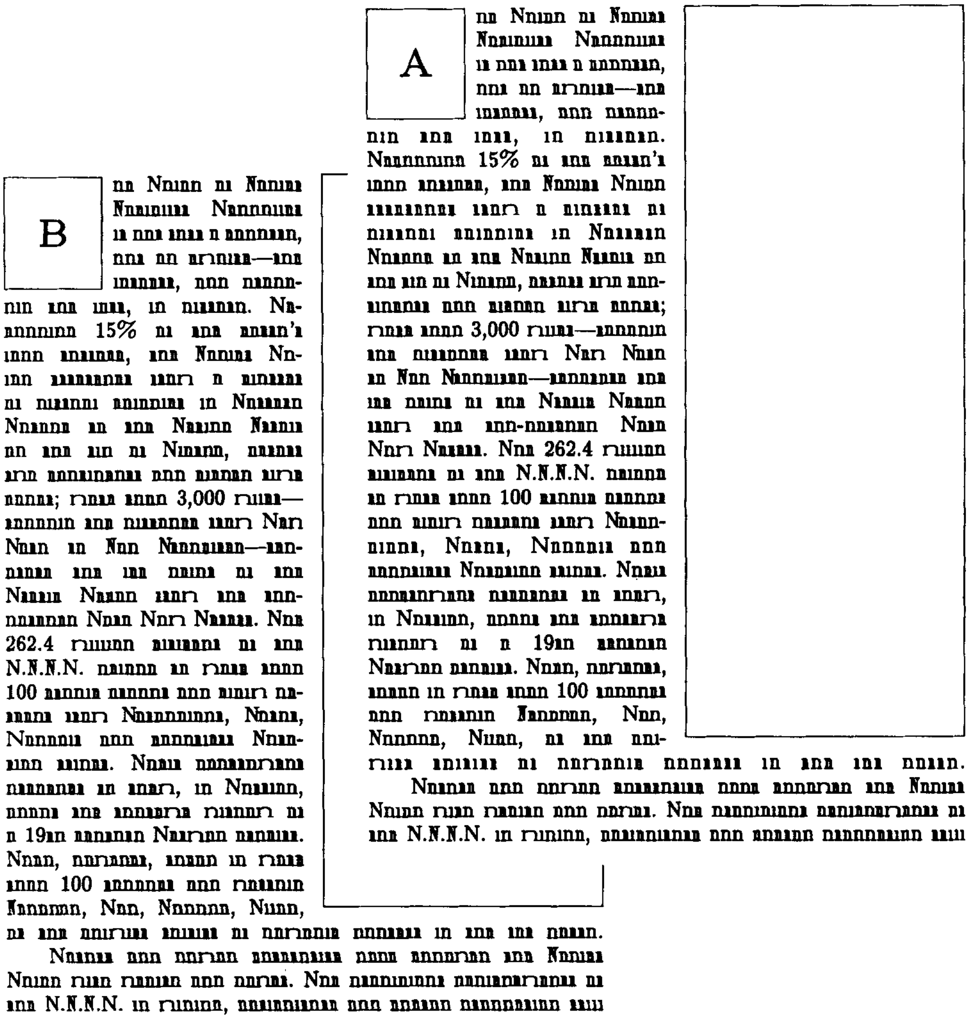


*Figure 28. This example is based on the spacing in a recent issue of Time magazine, but all of the letters have been replaced by n's of various widths. If the text were readable, the line breaks in Version B would be less distracting than those in Version A.*

unsuccessful experiments with dynamic programming, otherwise he might have rejected the whole idea subconsciously before pursuing it at all.

During the summer of 1977, M. F. Plass introduced the idea of feasible breakpoints into Knuth's original algorithm in order to limit the number of active possibilities and still find the optimum solution, unless the optimum was intolerably bad anyway. This algorithm was implemented in the first complete version of TEX (March 1978), and it appeared to work well. The unexpected power of the box/glue/penalty primitives gradually became clear during the next two years of experience with TEX; and when somewhat wild uses of negative parameters were discovered (as in the PASCAL and *Math Reviews* examples discussed above), it was necessary to ferret out subtle bugs in the original implementation.

Finally it became desirable to add more features to TEX's line-breaking procedure, especially an ability to vary the line widths with more flexibility than simple hanging indentation. At this point a more fundamental defect in the 1978 implementation became apparent, namely that it maintained at most one active node for each breakpoint regardless of the fact that a single breakpoint might feasibly occur on different lines; this meant that the algorithm could miss feasible ways to set a paragraph, in the presence of sufficiently long hanging indentation. A new algorithm was therefore developed in the spring of 1980 to replace TEX's previous method; at that time the refinements about looseness and adjacent-line mismatches were also introduced, so that TEX now uses essentially the optimum-fit algorithm that we have discussed in detail above.

## PROBLEMS AND REFINEMENTS

One unfortunate restriction remains in TEX although it is not inherent in the box/glue/penalty model: When a break occurs in the middle of a ligature (e.g., if 'efficient' becomes 'ef-ficient'), the computation of character widths is more complicated than usual. We must take into account not only the fact that a hyphen has some width, but also the fact that 'f' followed by 'fi' is wider than 'ffi'. The same problem occurs when setting German text, where some compound words change their spelling when they are hyphenated (e.g., 'backen' becomes 'bak-ken' and 'Bettuch' becomes 'Bett-tuch'). TEX does not permit such optional spelling variants; it will only insert an optional hyphen character among other unchangeable characters. Manual intervention is necessary in the rare cases when a more complicated break cannot be avoided.

It is interesting to consider what extension would be needed to make the optimum-fit algorithm handle cases like the dropping of m's and n's in Figure 22. The badness function of a line would then depend not only on its natural width, stretchability, and shrinkability; it would also depend on the number of m's and the number of n's on that line. A similar technique could be used to typeset biblical Hebrew, which is never hyphenated: Hebrew fonts intended for sacred texts usually include wide variants of several letters, so that individual characters on a line can be replaced by their wider counterparts in order to avoid wide spaces between words. For example, there is a super-extended aleph in addition to the normal one. An appropriate badness function for the lines of such paragraphs would take account of the number of dual-width characters present.

The most serious unanticipated problem that has arisen with respect to TEX's line-breaking procedure is the fact that *floating-point arithmetic* was used for all the calculations of badness, demerits, etc., in the original implementations. This leads to different results on different computers, since there is so much diversity in existing floating-point hardware, and since there are often two choices of breakpoints having almost the same total demerits. It is important to be able to guarantee that all versions of TEX will set paragraphs identically, because the ability to proofread, edit, and print a document at different sites is becoming significant. Therefore the 'standard' version of TEX, planned for release in 1982, will use fixed-point arithmetic for all of its calculations.

Books on typography frequently discuss a problem that may be the most serious consequence of loose typesetting, the occasional gaps of white space that are called 'houndsteeth' or 'lizards' or 'rivers'. Such ugly patterns, which run up through a sequence of lines and distract the reader's eye, cannot be eliminated by a simple efficient technique like dynamic programming. Fortunately, however, the problem almost never arises when the optimum-fit algorithm is used, because the computer is generally able to find a way to set the lines with suitably tight spacing. Rivers begin to be prevalent only when the tolerance threshold $\rho$ has been set high for some reason, for example in Figure 7 where an unusually narrow column is being justified, or in Figure 18(d) where the paragraph is two lines longer than optimum. Another case that sometimes leads to rivers arises when the text of a paragraph falls into a strictly mechanical pattern, as when a newspaper lists all of the guests at a large dinner party. Extensive experience with TEX has shown, however, that manual removal of rivers is almost never necessary after the optimum-fit algorithm has been used.

The box/glue/penalty model applies in the vertical dimension as well as in the horizontal, so TEX is able to make fairly intelligent decisions about where to start each new page. The tricks we have discussed for such things as ragged-right setting correspond to analogous vertical tricks for such things as 'ragged-bottom' setting. However, the current implementation of TEX keeps each page in memory until it has been output, so TEX cannot store an entire document and find strictly optimum page breaks using the algorithm we have presented for line breaks. The 'best-fit' method is therefore used to output one page at a time.

Experiments are now in progress with a two-pass version of TEX that does find globally optimum page breaks. This experimental system will also help with the positioning of illustrations as near as possible to where they are cited in the accompanying text, taking proper account of the fact that certain pages face each other. Many of these issues can be resolved by extending the dynamic programming technique and the box/glue/penalty model of this paper, but some closely related problems can be shown to be NP complete.[42]

## APPENDIX: A STRIPPED-DOWN ALGORITHM

Many applications of line breaking (e.g., in word processors) do not need all of the machinery of the general optimizing algorithm described in the text above, and it is possible to simplify the general procedure considerably while at the same time decreasing its space and time requirements, provided that we are willing to simplify the problem specifications and to tolerate less than optimal performance when hyphenation is necessary. The 'suboptimum-fit' program below is good enough to discover the line breaks of Figure 3 or Figure 4(c), but it will not handle some of the more complicated examples. More precisely, the stripped-down program assumes that

a) Instead of the general box/glue/penalty model, the input is specified by a sequence $w_1 \ldots w_n$ of nonnegative box widths representing the words of the paragraph and the attached punctuation, together with a sequence of small integers $g_1 \ldots g_n$ that specifies the type of space to be used between words. For example, we might have $g_k = 1$ when a normal interword space follows the box of width $w_k$, while $g_k = 2$ when there is to be no space since box $k$ ends with an explicit hyphen, and $g_k = 3$ when box $k$ is the end of the paragraph. Other type codes might be used after punctuation. Each type $g$ corresponds to three nonnegative numbers $(x_g, y_g, z_g)$ representing respectively the normal spacing, the stretchability, and the shrinkability of the corresponding type of space. For example, if types 1, 2, and 3 are used with the meanings just suggested, we might have

$$\begin{array}{ll}(x_1,y_1,z_1)=(6,3,2) & \text{between words}\\ (x_2,y_2,z_2)=(0,0,0) & \text{after explicit hyphens or dashes}\\ (x_3,y_3,z_3)=(0,\infty,0) & \text{to fill the final line}\end{array}$$

in terms of $\frac{1}{18}$em units, where $\infty$ stands for some large number. The width $w_1$ of the first box should include the blank space needed for paragraph indentation; thus, the Grimm fairy tale example of Figure 1 would be represented by

$$\begin{array}{l}w_1,\ldots,w_n=34,42,42,\ldots,24,39,30,\ldots,60,79\\ g_1,\ldots,g_n=\ \ 1,\ \ 1,\ \ 1,\ldots,\ \ 1,\ \ 2,\ \ 1,\ldots,\ \ 1,\ \ 3\end{array}$$

corresponding to

'⌴In', 'olden', 'times', ..., 'old', 'lime-', 'tree', ..., 'favorite', 'plaything.'

respectively, using widths from a typical roman font of type. The general input sequences $w_1\ldots w_n$ and $g_1\ldots g_n$ can be expressed in the box/glue/penalty model by the equivalent specification

$$\text{box}(w_1)\,\text{glue}(x_{g_1},y_{g_1},z_{g_1})\mathit{box}(w_2)\,\text{glue}(x_{g_2},y_{g_2},z_{g_2})\ldots\mathit{box}(w_n)\,\text{glue}(x_{g_n},y_{g_n},z_{g_n})$$

followed by 'penalty(0, $-\infty$, 0)' to finish the paragraph.

b) All lines must have the same width $l$, and each $w_k$ is less than $l$.

c) No word will be hyphenated unless there is no way to set the paragraph without violating minimum or maximum constraints on spacing. The minimum for type $g$ spaces is

$$z'_g=x_g-z_g$$

and the maximum is

$$y'_g=x_g+\rho y_g,$$

where $\rho$ is a positive tolerance that can be varied by the user. For example, if $\rho=2$ the maximum type $g$ space is $x_g+2y_g$, the normal amount plus twice the stretchability.

d) Hyphenation is performed only at the point where feasible line breaking becomes impossible, even though it may be better to hyphenate an earlier word. Thus, the general optimum-fit algorithm of the text will give substantially better results when high-quality output is desired and hyphenation is frequently necessary.

e) No penalty is assessed for a tight line next to a loose line, or for consecutive hyphenated lines, and the algorithm does not produce paragraphs that are longer or shorter than the optimum length. (In other words, $\alpha=\gamma=q=0$ in the general algorithm.)

Under these restrictions, optimum breakpoints can be found with extra efficiency.

The suboptimum-fit algorithm manipulates two arrays:

$$s_0 s_1 \ldots s_{n+1},$$

where $s_k$ denotes the minimum sum of demerits leading to a break after box $k$, or $s_k = \infty$ if there is no feasible way to break there; and

$$p_1 \ldots p_{n+1},$$

where $p_k$ is meaningful only if $s_k < \infty$, in which case the best case to end a line at box $k$ is to begin it with box $p_k + 1$. We also assume that

$$w_{n+1} = 0;$$

this represents an invisible box at the very end of the final line of the paragraph.

Besides the $4n+4$ storage locations for $w_1 \ldots w_{n+1}$, $g_1 \ldots g_n$, $s_0 \ldots s_{n+1}$, and $p_1 \ldots p_{n+1}$, and the memory required to hold the parameters $l$, $\rho$, and $(x_g, y'_g, z'_g)$ for each type $g$, the stripped-down algorithm needs only a few miscellaneous variables:

$a$ = the beginning of the paragraph (normally 0, changed after hyphenation);
$k$ = the current breakpoint being considered;
$j$ = the breakpoint being considered as a predecessor of $k$;
$i$ = the leftmost breakpoint that could feasibly precede $k$;
$m$ = the number of active breakpoints (i.e., subscripts $j \geqslant i$ with $s_j < \infty$);
$\Sigma$ = the normal width of a line from $i$ to $k$;
$\Sigma_{\max}$ = the maximum feasible width of a line from $i$ to $k$;
$\Sigma_{\min}$ = the minimum feasible width of a line from $i$ to $k$;
$\Sigma'$ = the normal width of a line from $j$ to $k$;
$\Sigma'_{\max}$ = the maximum feasible width of a line from $j$ to $k$;
$\Sigma'_{\min}$ = the minimum feasible width of a line from $j$ to $k$;
$r$ = adjustment ratio from $j$ to $k$;
$d$ = total demerits from $a$ to ... to $j$ to $k$;
$d'$ = minimum total demerits known from $a$ to ... to $k$;
$j'$ = predecessor of $k$ that leads to $d'$ total demerits, if $d' < \infty$.

All of these variables are integers, except $r$, which will be a fraction in the range $-1 \leqslant r \leqslant \rho$. The reader may verify the validity of the algorithm by verifying that these interpretations of the variables remain invariant in key places as the program proceeds.

Here now is the program, viewed from the 'top down':

```
$a := 0$;
loop: $i := a$; $s_i := 0$; $k := i+1$; $\Sigma := \Sigma_{\max} := \Sigma_{\min} := w_k$; $m := 1$;
  loop: while $\Sigma_{\min} > l$ do $\langle$advance $i$ by 1$\rangle$;
  $\langle$examine all feasible lines ending at $k\rangle$;
  $s_k := d'$; if $d' < \infty$ then
    begin $m := m+1$; $p_k := j'$;
    end;
    **if** $m = 0$ **or** $k > n$ **then exit loop**;
    $\Sigma := \Sigma + w_{k+1} + x_{g_k}$; $\Sigma_{\max} := \Sigma_{\max} + y'_{g_k}$; $\Sigma_{\min} := \Sigma_{\min} + z'_{g_k}$;
    $k := k + 1$;
    **repeat**;
  **if** $k > n$ **then**
    **begin** *output*$(a, n+1)$; **exit loop**;
    **end**
  **else begin** ⟨try to hyphenate box $k$, then output from $a$ to this break⟩;
    $a := k - 1$;
    **end**;
  **repeat**.
```

The operation 'advance $i$ by 1' is carried out only when $\Sigma_{\min} > l$, and this cannot happen when $k = i+1$ since $\Sigma_{\min} = w_k < l$ in such a case. Therefore the while loop terminates; we have

```
  ⟨advance $i$ by 1⟩ =
    **begin if** $s_i < \infty$ **then** $m := m - 1$;
    $i := i + 1$;
    $\Sigma := \Sigma - w_i - z_{g_i}$; $\Sigma_{\max} := \Sigma_{\max} - w_i - y'_{g_i}$; $\Sigma_{\min} := \Sigma_{\min} - w_i - z'_{g_i}$;
    **end**.
```

The inner loop of the suboptimal-fit program is simpler and faster than the corresponding loop in the general optimum-fit algorithm because it does not consider active breakpoints near $k$, only those that are approximately one line-width away:

```
  ⟨examine all feasible lines ending at $k$⟩ =
    begin $j := i$; $\Sigma' := \Sigma$; $\Sigma'_{\max} := \Sigma_{\max}$; $\Sigma'_{\min} := \Sigma_{\min}$; $d' := \infty$;
    **while** $\Sigma'_{\max} \geqslant l$ **do**
      **begin if** $s_j < \infty$ **then** ⟨consider breaking from $a$ to . . . to $j$ to $k$⟩;
      $j := j + 1$;
      $\Sigma' := \Sigma' - w_j - x_{g_j}$; $\Sigma'_{\max} := \Sigma'_{\max} - w_j - y'_{g_j}$; $\Sigma'_{\min} := \Sigma'_{\min} - w_j - z'_{g_j}$;
    **end**.
```

Again we can conclude that the while loop must terminate, since it will not be executed when $k = j+1$. The innermost code is easily fleshed out:

```
  ⟨consider breaking from $a$ to . . . to $j$ to $k$⟩ =
    **begin if** $\Sigma' < l$ **then** $r := \rho \cdot (l - \Sigma')/(\Sigma'_{\max} - \Sigma')$
    **else if** $\Sigma' > l$ **then** $r := (l - \Sigma')/(\Sigma' - \Sigma'_{\min})$
    **else** $r := 0$;
    $d := s_j + (1 + 100|r|^3)^2$;
    **if** $d < d'$ **then**
      **begin** $d' := d$; $j' := j$;
      **end**;
    **end**.
```

When hyphenation is necessary, the algorithm goes into panic mode, first searching for the last value of $i$ that was feasible, then attempting to split word $k$. At this point the line from $i$ to $k-1$ is too short, and from $i$ to $k$ it is too long, so there is hope that hyphenation will succeed.

```
⟨try to hyphenate box $k$, then output from $a$ to this break⟩ =
  begin loop: $\Sigma := \Sigma + w_i + x_{g_i}$;
  $\Sigma_{\max} := \Sigma_{\max} + w_i + y'_{g_i}$; $\Sigma_{\min} := \Sigma_{\min} + w_i + z'_{g_i}$; $i := i-1$;
    if $s_i < \infty$ then exit loop;
    repeat;
  $output(a, i)$;
  ⟨split box $k$ at the best place⟩;
  ⟨output the line up to the best split and adjust $w_k$ for continuing⟩;
  end.
```

Let us suppose that there are $h_k$ ways to split box $k$ into two pieces, where the widths of these pieces in the $j$th such split are $w'_{kj}$ and $w''_{kj}$, respectively; here $w'_{kj}$ includes the width of an inserted hyphen. An auxiliary hyphenation algorithm is supposed to be able to compute $h_k$ and these piece widths on demand; this algorithm is invoked only when we reach the routine 'split box $k$ at the best place'. If no hyphenation is desired one can simply let $h_k = 0$, and the program below becomes much simpler. There are $h_k + 1$ alternatives to be considered, including the alternative of not splitting at all, and the choice can be made as follows:

```
⟨split box $k$ at the best place⟩ =
  begin ⟨invoke hyphenation algorithm to compute $h_k$ and the piece widths⟩;
  $j' := 0$; $d' := \infty$;
  for $j := 1$ to $h_k$ do if $\Sigma_{\min} + w'_{kj} - w_k \leqslant l$ then
    begin $\Sigma' := \Sigma + w'_{kj} - w_k$;
    if $\Sigma' \leqslant l$ then $d := 10000\rho \cdot (l - \Sigma')/(100(\Sigma_{\max} - \Sigma) + 1)$
    else $d := 10000 \cdot (\Sigma' - l)/(100(\Sigma - \Sigma_{\min}) + 1)$;
    if $d < d'$ then
      begin $d' := d$; $j' := j$;
      end;
    end;
  end.
```

The final operation, 'output the line up to the best split and adjust $w_k$ for continuing', will only be sketched here since it is much easier to state it informally than to introduce still more notation. If $j' \neq 0$, so that hyphenation is to be performed, the program outputs a line from box $i+1$ to box $k$ inclusive, but with box $k$ replaced by the hyphenated piece of width $w'_{kj'}$; then $w_k$ is replaced by the width of the other fragment, namely $w''_{kj'}$. In the other case when $j' = 0$, the program simply outputs a line from box $i+1$ to box $k-1$ inclusive.

One more loose end needs to be tightened up: The procedure '$output(a, i)$' simply goes through the $p$ table determining the best line breaks from $a$ to $i$ and typesets the corresponding lines. One way to do this without requiring extra memory space is to reverse the relevant $p$-table entries so that they point to successors instead of predecessors:

```
procedure $output$(integer $a, i$) =
  begin integer $q, r, s$; $q := i$; $s := 0$;
  while $q \neq a$ do
    begin $r := p_q$; $p_q := s$; $s := q$; $q := r$;
    end;
  **while** $q \neq i$ **do**
    **begin** ⟨output the line from box $q+1$ to box $s$, inclusive⟩;
    $q := s$; $s := p_q$;
    **end**;
  **end**.
```

In practice there is only a bounded amount of memory available for implementing this algorithm, but arbitrarily long paragraphs can be handled if we make a minor change suggested by Cooper[33]: When the number of words in a given paragraph exceeds some maximum number $n_{\max}$, apply the method to the first $n_{\max}$ words; then output all but the final line and resume the method again, beginning with the copy carried over from the line that was not output.

## ACKNOWLEDGEMENTS

We wish to thank Barbara Beeton of the American Mathematical Society for numerous discussions about 'real world' applications; we also are grateful to James Eve of the University of Newcastle-Upon-Tyne and Neil Wiseman of Cambridge University for helping us obtain literature that was not readily available in California; and we thank the librarians of the rare book rooms at Columbia University and Stanford University for letting us study and photograph excerpts from polyglot Bibles. John Wiley & Sons Limited have taken unuśual care in typesetting this paper in exact accordance with the line breaks and page breaks found by TEX.

## REFERENCES

1. Michael P. Barnett, *Computer Typesetting: Experiments and Prospects*, M.I.T. Press, Cambridge, Mass., 1965.
2. C. J. Duncan, 'Look! No hands!', *The Penrose Annual* **57**, 121–168 (1964).
3. Michael R. Garey and David S. Johnson, *Computers and Intractability*, W. H. Freeman, San Francisco, 1979.
4. Richard Bellman, *Dynamic Programming*, Princeton Univ. Press, Princeton, N.J., 1957.
5. M. Held and R. M. Karp, 'The construction of discrete dynamic programming algorithms', *IBM Systems J.* **4**, 136–147 (1965).
6. Donald E. Knuth, *TEX and METAFONT: New Directions in Typesetting*, American Mathematical Society and Digital Press, Bedford, Massachusetts, 1979.
7. Jakob Ludwig Karl Grimm and Wilhelm Karl Grimm, 'Der Froschkönig (The Frog King)', in *Kinder- und Hausmärchen*, first published in Berlin, 1812. For the history of this story see Heinz Rölleke, *Die Altese Märchensammlung der Brüder Grimm*, Fondation Martin Bodmer, Cologny-Genève, 1979, pp. 144–153.
8. C. J. Duncan, J. Eve, L. Molyneux, E. S. Page, and Margaret G. Robson, 'Computer typesetting: an evaluation of the problems,' *Printing Technology* **7**, 133–151 (1963).
9. Donald E. Knuth, *Seminumerical Algorithms*, Vol. 2 of *The Art of Computer Programming*, second edition, Addison–Wesley, Reading, Massachusetts, 1981.
10. A. Frey, *Manuel Nouveau de Typographie*, Paris (1835), 2 vols.
11. Kathleen Jensen and Niklaus Wirth, *PASCAL User Manual and Report*, Heidelberg, Springer-Verlag, 1975.
12. Donald E. Knuth, 'BLAISE, a preprocessor for PASCAL,' file BLAISE.DEK[up,doc] at SU-AI on the ARPA network (March 1979). The program itself is on file BLAISE.SAI[tex,dek].
13. Donald E. Knuth, *Tau Epsilon Chi: A System for Technical Text*, book in preparation.
14. Theodore Low De Vinne, *Correct Composition*, Vol. 2 of *The Practice of Typography*, Century, New York, 1901. The cited material appears on pages 138 and 206.

15. George Bernard Shaw, 'On Modern Typography', *The Dolphin* **4**, 80–81 (1940).
16. T. H. Darlow and H. F. Moule, *Historical Catalogue of the Printed Editions of Holy Scripture in the library of The British and Foreign Bible Society,* The Bible House, London, 1911.
17. Basil Hall, *The Greatest Polyglot Bibles,* The Book Club of California, San Francisco, 1966.
18. Jiménez de Cisneros, sponsor, *Uetus testamentum multiplici lingua nunc primo impressum,* Industria Arnaldi Guillelmi de Brocario in Academia Complutensi, 1522. [The printing was completed in 1517, but papal permission to publish this book was delayed for several years.]
19. Aug. Giustiniani, *Psalterium,* Genoa, 1516.
20. Benedictus Arias Montanus, editor, *Biblia Sacra Hebraice, Chaldaice, Græce, & Latine,* Christoph. Plantinus, Antwerp, 1569–1573.
21. Brianus Waltonus, editor, *Biblia Sacra Polgyglotta,* Thomas Roycroft, London, 1657.
22. David Wolder, *Biblia Sacra Græce, Latine & Germanice,* Jacobus Lucius Juni., Hamburg, 1596.
23. Walter E. Houghton, Jr., 'The History of Trades: Its relation to seventeenth century thought,' in Philip P. Wiener and Aaron Noland, eds., *Roots of Scientific Thought,* Basic Books, New York, 1957, pp. 354–381.
24. Joseph Moxon, *Mechanick Exercises,* J. Moxon, London, 1683. Reprinted by the Typothetae of New York, 1896, with preface and notes by T. L. De Vinne; also reprinted by Oxford University Press, London, 1958; but these reprints do not capture the full feeling of the original, with its less sumptuous seventeenth-century workmanship. Quoted passages are from vol. 2, pp. 214–215, 226, 245, 248.
25. D. G. Berri, *The Art of Printing,* London, 1864.
26. Samual A. Bartels, *The Art of Spacing,* The Inland Printer, Chicago, 1926.
27. G. P. Bafour, A. R. Blanchard, and F. H. Raymond, 'Automatic Composing Machine,' U.S. Patent 2762485, September 11, 1956. (See also British patent 771551 and French patent 1103000.)
28. G. Bafour, 'A new method for text composition—The BBR System,' *Printing Technology* **5**, no. 2, 1961, 65–75.
29. Joseph F. Ossanna, 'NROFF/TROFF User's Manual,' Bell Telephone Laboratories Internal memorandum, Murray Hill, New Jersey, 1975.
30. Paul E. Justus, 'There is more to typesetting than setting type', *IEEE Trans. on Prof. Commun.* **PC-15**, 13–16, 18 (1972).
31. John Pierson, *Computer Composition using PAGE-1,* Wiley-Interscience, New York, 1972.
32. Information International, Inc., 'PAGE-3 Composition Language,' privately distributed. First edition, October 31, 1975; second edition, October 20, 1976. The language is sometimes called 'PAGE-III' because of the company that created it.
33. P. I. Cooper, 'The influence of program parameters on hyphenation frequency in a sophisticated justification program,' *Advances in Computer Typesetting* [Proceedings of the 1966 International Computer Typesetting Conference], The Institute of Printing, London, 1967, 176–178, 211–212.
34. [Untitled] Moderators' summaries of the papers presented at the International Computer Typesetting Conference at the University of Sussex, The Institute of Printing, London, 1966.
35. Alison M. Pringle, 'Justification with fewer hyphens,' Rainbow Memo 170, University of Cambridge Computer Laboratory, March 1980.
36. Hanan Samet, 'Heuristics for the line division problem in computer justified text,' preprint, University of Maryland, 1980.
37. H. D. Parks, 'Computerized processing of editorial copy', *Advances in Computer Typesetting* [Proceedings of the 1966 International Computer Typesetting Conference], The Institute of Printing, London, 1967, 176–178, 211–212.
38. Herman Parks, contributions to the discussions, *Proc. ASIS Workshop on Computer Composition,* American Society for Information Science, 1971, pp. 143–145, 151, 180–182.
39. Bruce Rogers, *Paragraphs on Printing,* William E. Rudge's Sons, New York, 1943, p. 88.
40. U.S. Government Printing Office, *Style Manual,* Washington, D.C., 1973. The quote is from rule 22 (catch?).
41. Michael J. Clancy and Donald E. Knuth, 'A programming and problem-solving seminar,' report STAN-CS-77-606, Computer Science Department, Stanford University, April 1977, 85–88.
42. Michael F. Plass, 'Optimal pagination techniques for automatic typesetting systems,' Ph.D. thesis, Stanford University, June 1981.