equivalent to the task of verifying an inequality proposition regarding the minimax value of a continuous-valued game tree [5] of identical structure, and, consequently, the former cannot be more complex than the latter. Thus, the quantity $(\xi_n/1 - \xi_n)^d$ should also lower bound the expected number of nodes examined by any algorithm searching a continuous-valued game tree. This, together with Eq. (18), establishes the asymptotic optimality of $\alpha$–$\beta$.

Received 5/80; revised 7/81; accepted 11/81

**References**

1. Baudet, G.M. On the branching factor of the alpha-beta pruning algorithm. *Artificial Intelligence 10*, 2 (April 1978), 173–199.
2. Fuller, S.H., Gaschnig, J.G., and Gillogly, J.J. An analysis of the alpha-beta pruning algorithm. Department of Computer Science Report, Carnegie-Mellon University, (1973).
3. Knuth, D.E., and Moore, R.N. An analysis of alpha-beta pruning. *Artificial Intelligence* 6 (1975), 293–326.
4. Kuczma, M. *Functional Equations in a Single Variable.* Polish Scientific Publishers, Warszawa, (1968), p. 141.
5. Pearl, J. Asymptotic properties of minimax trees and game-searching procedures. *Artificial Intelligence 14*, 2 (Sept. 1980), 113–138.
6. Pearl, J. A space-efficient on-line method of computing quantile estimates *J. of Algorithms 2*, 2 (June 1981) 24–28.
7. Roizen, I. On the average number of terminal nodes examined by alpha-beta. UCLA-ENG-CSL-8108, Cognitive Systems Laboratory, University of California, Los Angeles, (1981).
8. Slagle, J.R., and Dixon, J.K. Experiments with some programs that search game trees. *JACM 16*, 2 (April 1969) 189–207.
9. Stockman, G. A minimax algorithm better than alpha-beta? *Artificial Intelligence 12*, 2 (Aug. 1979), 179–196.
10. Tarsi, M. Optimal searching of some game trees. UCLA-ENG-CSL-8108, Cognitive Systems Laboratory, University of California, Los Angeles, (1981). (To appear in *JACM.)*

Programming Techniques And Data Structures

Ronald L. Rivest* Editor

# Heuristics for the Line Division Problem in Computer Justified Text

Hanan Samet
University of Maryland

**Measures for evaluating solutions to the line division problem in computer justified text are presented. They are based on the belief that documents tend to have a more pleasing visual appearance when the deviation between interword breaks in a paragraph is reduced. This effect is achieved by not placing the maximum number of words on each line. The measures are variations on the variance of the number of extra spaces per interword break in a paragraph. They are applicable to both fixed and variable width fonts. One of the measures is examined in greater detail. It has the property that a lower bound can be computed, thereby indicating when further rearrangement of the text is futile. Several text rearrangement algorithms are proposed that make use of this measure.**

**CR Categories and Subject Descriptors:** I.7.2 [**Text Processing**]: Document Preparation—*format and notation, photocomposition*; H.4 [**Information Systems Application**]: Office Automation—*word processing*

**General Term:** Algorithms

**Additional Key Words and Phrases:** line division, text justification, typesetting, layout, spacing, line breaking

## 1. Introduction

The dramatic rise in the use of interactive computer facilities has been coupled with a rise in the use of text editing programs. This has in turn led to the development of document processing systems whose role is to trans-

* Former editor of Programming Techniques and Data Structures, of which Ellis Horowitz is the current editor.

Author's present address: Hanan Samet, Department of Computer Science, University of Maryland, College Park MD 20742.

form the input text into output which meets a certain specification. The most notable of these new systems are SCRIBE [6] and TEX [4].

One of the functions of such document processing systems is to perform text justification. This is generally achieved by processing the text in sequence, placing as many words as can be fit within the margins in each line, and then inserting extra spaces if necessary to achieve the desired effect (termed *filling*). The main problem with such a method is its unidirectionality, i.e., documents tend to look nonuniform because of long words that cannot be fit at the end of certain lines (e.g., Fig. 1). Note that this tendency could have been reduced by moving forward some words from previously filled lines to reduce the glaring nonuniformity in the line in question. For example, Fig. 2 applies such a technique to Fig. 1. The problem of nonuniformity is termed the *line division* problem by Knuth [3].

In this paper we discuss measures for evaluating solutions to the line division problem. One of the measures is examined in greater detail and is shown to have the property that a lower bound can be computed, thereby indicating when further rearrangement of the text is futile. The measure could be used as a cost function and its value optimized through dynamic programming techniques [1]. This approach is described in [3], where a different measure is used. The measure is applicable to both fixed and variable width fonts. As a note of caution, we mention that nonuniformity is an aesthetic property and thus our results are based in part on our judgment that use of our measure leads to text that is usually more attractive in a visual sense.

## 2. Measure

Any measure that we define must satisfy the following criterion. First, it must be simple from the standpoint of computational complexity. Second, it must correlate with nonuniformity, i.e., as the nonuniformity decreases, so does the value of the measure. This section motivates the type of measure we seek and proposes two possible candidates. One of the candidate measures is shown to meet our criteria in a superior manner to the other. However, first we make the following remark.

*Remark 1.* For a given paragraph, a text justification method that processes the words in the body in increasing order, packing as many words as possible in a line, uses a minimum number of lines.

*Proof.* Denote by $B$ the result of justification in increasing order and let $n$ be the number of lines that are required. Denote by $B'$ the result of another justification method that requires $m$ lines. We say that $B(k)$ and $B'(k)$ correspond to the number of words in lines $k$ in $B$ and $B'$, respectively. Define $S(i)$ and $S'(i)$ as follows:

$$S(i) = \sum_{k=1}^{i} B(k)$$

$$S'(i) = \sum_{k=1}^{i} B'(k).$$

By definition of the text justification method used to obtain $B$ we have

$$S(i) \geq S'(i), \qquad 1 \leq i \leq \min(m, n) \qquad (*)$$

Assuming $n > m$, we have that $S(m) < S(n)$. Using (*), we have $S(n) > S(m) \geq S'(m)$ or $S(n) > S'(m)$. But $S(n) = S'(m) =$ number of words in the paragraph, and thus we have a contradiction. Therefore, we have proven that there is no text justification method that uses fewer lines than one that processes the body in increasing order, packing as many words as possible in each line.

Q.E.D.

Nonuniformity can be defined formally as a property of a document which is characterized by a wide fluctuation in the amount of space in excess of a mandatory minimum interword break. We make the following remark.

Fig. 1. Text Sample.

```
Compiler testing is a  term we use to  describe a means of  proving that
given a compiler  or for that  matter any program  translation procedure
that the  translation has been  correctly performed.  We  are especially
interested in cases where the translation involves a considerable amount
of optimization.  Some  possible approaches  to  this  problem include
program      proving[London72],      program      testing[Huang75],      and
decompilation[Hollander73].
```

Fig. 2. Result of Rearranging Fig. 1.

```
Compiler testing is a  term we use to  describe a means of  proving that
given a compiler  or for that  matter any program  translation procedure
that the  translation has been  correctly performed.  We  are especially
interested  in  cases  where  the  translation  involves a  considerable
amount  of  optimization.  Some  possible approaches  to  this  problem
include program   proving[London72],   program   testing[Huang75],   and
decompilation[Hollander73].
```

*Remark 2.* The average amount of extra space per interword break in a paragraph is a constant when the text justification method uses a minimum number of lines to contain the paragraph.

*Proof.* A direct consequence of Remark 1. Q.E.D.

In other words, the average amount of extra space per interword break cannot be increased or decreased no matter how we allocate the spaces within a line whenever the minimum number of lines is used. We ignore the last line of a paragraph since nonuniformity is not a factor here, i.e., nothing can be moved from this line to the previous lines and, similarly, moving words from previous lines into the last line will only result in increasing the average amount of extra space per interword break in the paragraph (excluding the last line). We assume that the minimum interword break is a constant. This is an oversimplification in the case of a period.

Since the average amount of extra space per interword break is constant for each paragraph, we need at least a measure of the second order variation of this quantity (the average is the first order variation). Such a measure is similar to the variance [5]. We propose two possible measures. However, first let us define some terms that are useful in describing our measures. Note that in our definitions a paragraph is defined to begin at the extreme left of the first line (i.e., no indentation) and is said to contain all subsequent lines with the exception of the last line. Also, space is defined in units appropriate to the font being used (e.g., 1 for fixed width fonts and mills for variable width fonts).

| | | |
|---|---|---|
| $n$ | = | number of lines in a paragraph |
| $w$ | = | number of words in a paragraph |
| $b$ | = | $w - n$, number of interword breaks in a paragraph |
| $s$ | = | minimum interword break length ($s > 0$) |
| $e_{ij}$ | = | extra space between words $j$ and $j + 1$ in line $i$ |
| $e_i$ | = | extra space in line $i$ |
| $w_i$ | = | number of words in line $i$ |
| $b_i$ | = | $w_i - 1$, number of interword breaks in line $i$ |
| $\mu$ | = | average extra space per interword break in a paragraph (a constant) |
| $\mu_i$ | = | $e_i/b_i$, average extra space per interword break in line $i$ |
| $\bar{\mu}$ | = | $\sum_{i=1}^{n} (\mu_i/n)$ |
| $d_i$ | = | $e_i$ div $b_i$, result of integer division of $e_i$ and $b_i$ |
| $m_i$ | = | $e_i$ mod $b_i$, remainder of integer division of $e_i$ and $b_i$. |

Note that $e_i = b_i{*}d_i + m_i$.

Two possible measures are given in (1) and (2):

$$\frac{\sum_{i=1}^{n} \frac{1}{b_i} \sum_{j=1}^{b_i} (e_{ij} - \mu)^2}{n} \tag{1}$$

$$\frac{\sum_{i=1}^{n} \sum_{j=1}^{b_i} (e_{ij} - \mu)^2}{b}. \tag{2}$$

The meaning of measure (2) is obvious: the average variation of all the extra interword space in the paragraph, without regard to its distribution. Measure (1), however, is slightly more complex in that it consists of finding the average variation per line and then finding the average of this average throughout the paragraph.

Equation (1) can be reduced to yield (1′) as follows.

$$\begin{aligned}
&\frac{1}{n} \sum_{i=1}^{n} \frac{1}{b_i} \sum_{j=1}^{b_i} (e_{ij} - \mu)^2 \\
&= \frac{1}{n} \sum_{i=1}^{n} \frac{1}{b_i} \sum_{j=1}^{b_i} (e_{ij}^2 - 2\mu e_{ij} + \mu^2) \\
&= \frac{1}{n} \left( \sum_{i=1}^{n} \left( \frac{1}{b_i} \sum_{j=1}^{b_i} e_{ij}^2 - \frac{2\mu}{b_i} \sum_{j=1}^{b_i} e_{ij} + \frac{b_i \mu^2}{b_i} \right) \right) \\
&= \frac{1}{n} \sum_{i=1}^{n} \left( \left( \frac{1}{b_i} \sum_{j=1}^{b_i} e_{ij}^2 \right) - 2\mu \frac{e_i}{b_i} + \mu^2 \right) \\
&= \frac{1}{n} \sum_{i=1}^{n} \left( \left( \frac{1}{b_i} \sum_{j=1}^{b_i} e_{ij}^2 \right) - 2\mu\mu_i + \mu^2 \right) \\
&= \left( \frac{1}{n} \sum_{i=1}^{n} \left( \frac{1}{b_i} \sum_{j=1}^{b_i} e_{ij}^2 \right) \right) - \frac{2\mu}{n} \left( \sum_{i=1}^{n} \mu_i \right) + \frac{n\mu^2}{n} \\
&= \sum_{i=1}^{n} \frac{1}{nb_i} \left( \sum_{j=1}^{b_i} e_{ij}^2 \right) - 2\mu\bar{\mu} + \mu^2.
\end{aligned} \tag{1′}$$

Equation (2) can be reduced to yield (2′) as follows

$$\begin{aligned}
&\frac{1}{b} \sum_{i=1}^{n} \sum_{j=1}^{b_i} (e_{ij} - \mu)^2 \\
&= \frac{1}{b} \sum_{i=1}^{n} \sum_{j=1}^{b_i} (e_{ij}^2 - 2\mu e_{ij} + \mu^2) \\
&= \frac{1}{b} \sum_{i=1}^{n} \sum_{j=1}^{b_i} e_{ij}^2 - \frac{2\mu}{b} \sum_{i=1}^{n} \sum_{j=1}^{b_i} e_{ij} + \frac{1}{b} \sum_{i=1}^{n} b_i \mu^2 \\
&= \frac{1}{b} \sum_{i=1}^{n} \sum_{j=1}^{b_i} e_{ij}^2 - 2\mu^2 + \mu^2 \\
&= \frac{1}{b} \sum_{i=1}^{n} \sum_{j=1}^{b_i} e_{ij}^2 - \mu^2.
\end{aligned} \tag{2′}$$

Note that we made use of the relation

$$\sum_{i=1}^{n} \sum_{j=1}^{b_i} e_{ij} = b\mu.$$

We now have sufficient information at hand to evaluate the two measures. We do not use as our basis the minimum variance of the estimator [2]; instead, we use criteria based on the application at hand. First, measure (1) is computationally more complex than measure (2). This stems from the presence of the $2{*}\mu{*}\bar{\mu}$ term which must be re-evaluated whenever $e_{ij}$ changes. Second, and most importantly, measure (1) is less desirable than measure (2) because it tends to weigh equally lines with a small number of long words and lines with a large number of short words. This means that the overall contribution per word to the nonuniformity raggedness measure from a line with a small number of long words is greater than the contribution of a line with a large number of short words. Moreover, we want to distribute the extra space evenly throughout all of the interword breaks in the paragraph rather than to distribute the extra space evenly throughout each line as measure (1) favors. In other words, moving words into lines with a small number of long words is more effective in reducing nonuniformity according to measure (1) than vice versa, whereas we want to be able to move words onto any line with an equal effect on the nonuniformity measure. This is the case with measure (2).

Taking advantage of the discreteness of the width of characters leads us to obtain the following simplifications for our measures. Observe that any space distribution algorithm will allocate the extra spaces in the following manner:

$d_i$ spaces in each of $b_i - m_i$ interword breaks in line $i$ (3a)

$d_i + 1$ spaces in each of $m_i$ interword breaks in line $i$. (3b)

Substitution of (3a) and (3b) into (2′) yields

$$\frac{1}{b}\sum_{i=1}^{n}\sum_{j=1}^{b_i} e_{ij}^2 - \mu^2$$

$$= \frac{1}{b}\sum_{i=1}^{n} (d_i^2(b_i - m_i) + (d_i + 1)^2 m_i) - \mu^2$$

$$= \frac{1}{b}\sum_{i=1}^{n} (b_i d_i^2 - d_i^2 m_i + d_i^2 m_i + m_i + 2d_i m_i) - \mu^2$$

$$= \frac{1}{b}\sum_{i=1}^{n} (b_i d_i^2 + m_i + 2d_i m_i) - \mu^2$$

$$= \frac{1}{b}\sum_{i=1}^{n} (m_i + d_i(b_i d_i + 2m_i)) - \mu^2$$

$$= \frac{1}{b}\sum_{i=1}^{n} (m_i + d_i(b_i d_i + m_i + m_i)) - \mu^2$$

$$= \frac{1}{b}\sum_{i=1}^{n} (m_i + d_i(e_i + m_i)) - \mu^2.$$

Hence we only need to attempt to reduce the sum:

$$\sigma = \sum_{i=1}^{n} (m_i + d_i*(e_i + m_i)). \qquad (4)$$

In order to simplify future discussion we define the following relationship.

$$\sigma_i = m_i + d_i*(e_i + m_i).$$

The algebraic representation of (4) has several important ramifications. First, notice that $e_i$ is measured in terms of space units; thus, (4) holds for both fixed and variable width fonts. Second, (4) yields information as to whether further text rearrangement is worthwhile. In essence, (4) implies that there exists a lower bound for the measure as given by the following remark.

*Remark 3.* $\sum_{i=1}^{n} e_i$ is a lower bound for measure (2) and is attained when $e_i \le b_i$ for every line $i$.

*Proof.* Recall that $e_i = b_i*d_i + m_i$. Clearly, for $0 \le e_i < b_i$, we have that $d_i = 0$ and $m_i = e_i$. Also, for $e_i = b_i$, we have that $d_i = 1$ and $m_i = 0$. Thus, whenever $e_i \le b_i$ for every line $i$, (4) attains a minimum value and likewise for measure (2). Q.E.D.

Therefore, when Remark 3 is satisfied, no further work can reduce the value of measure (2) and thus a one pass algorithm, as is used in most document processors, will achieve optimality. Note, also, that (4) only involves integer quantities and thus, if it were to be implemented in hardware (e.g., using a microprocessor), then there would be no need for a floating point arithmetic capability.

At this point we elaborate further on the optimality criterion set forth in the previous paragraph. The conclusion that optimality is achieved when $e_i$ is less than or equal to $b_i$ for all $i$ is intuitive. For example, consider the case of decreasing $\sigma_j$ by moving, from lines $j - 1$ to $j$, $n_j$ words occupying $s_j$ units of space. Such action causes $\sigma_{j-1}$ to increase, and as long as $s_j$ is less than or equal to $b_{j-1} - n_j - e_{j-1}$, then $\sigma$ remains constant.[1] Otherwise, it increases, or we must perform a similar operation between lines $j - 2$ and $j - 1$, and so forth. Optimality is achieved whenever we encounter a line $k$ such that the previous condition is satisfied, i.e., $s_k \le b_{k-1} - n_k - e_{k-1}$.

The close relationship between (4) and optimality is the reason for the selection of measure (2) over measure (1). Note that the simplifications (3a) and (3b) applied to (2′) to obtain (4) could also have been applied to (1′) to obtain the following:

$$\mu^2 + \sum_{i=1}^{n} ((m_i + d_i*(e_i + m_i)) - 2*\mu*e_i)/b_i).$$

Any algorithm that we would devise would attempt to reduce the sum:

$$\sum_{i=1}^{n} ((m_i + d_i*(e_i + m_i)) - 2*\mu*e_i)/b_i). \qquad (5)$$

However, unlike (4), there is no obvious way to determine how close a specific value of (5) is to the lower bound for optimality. For example, letting $\sigma_i$ denote the elements being summed, we have that when $0 \le e_i \le b_i$, $\sigma_i = (e_i - 2*\mu*e_i)/b_i$. Note that changes to $e_i$ within a line cause $b_i$ to change. Thus $\sigma_i$ is no longer constant for $0 \le e_i \le b_i$, as is the case when measure (2) [i.e., (4)] is used.

[1] The new number of interword breaks in line $j - 1$ is $b_{j-1} - n_j$. Motion of $s_j$ units of space from lines $j - 1$ to $j$ causes line $j - 1$ to have $s_j + e_{j-1}$ extra units of space. Optimality requires that this be less than or equal to the number of interword breaks which has now been reduced to $b_{j-1} - n_j$. Hence, we have that $s_j + e_{j-1} \le b_{j-1} - n_j$ or $s_j \le b_{j-1} - n_j - e_{j-1}$.

Fig. 3. Alternative Rearrangement of Fig. 1.

```
Compiler testing is a  term we use to  describe a means of  proving that
given a compiler  or for that  matter any program  translation procedure
that the  translation has been  correctly performed.  We  are especially
interested  in  cases  where  the  translation  involves a  considerable
amount    of    optimization.     Some     possible     approaches    to    this
problem include program proving[London72], program testing[Huang75], and
decompilation[Hollander73].
```

## 3. Example

As an example of the validity of our candidate measures, let us apply them to the paragraph in Fig. 1 and see whether they can distinguish between Figs. 1, 2, and 3. The difference between Figs. 2 and 3 is in the fifth and sixth lines of the paragraph where one more word has been squeezed into line 6 of Fig. 3 and consequently one less word occurs in line 5. We assume a fixed width font. The values of the measures [using (4) and (5)] for the three figures are given in Fig. 4. Notice that the value of measure (2) for Fig. 2 is very close to the lower bound for the measure which is 33. However, it should be clear that there need not exist a version of the paragraph whose measure value equals the lower bound. Hence Fig. 2 corresponds to the optimal paragraph both from an aesthetic point of view (this is somewhat arbitrary) and as a result of the application of our measures. Thus it is seen that both measures perform adequately in this case. As expected, measure (1), relatively speaking, is biased towards any rearrangement of text that reduces the extra space between words in a line with a small number of long words.

## 4. Algorithms

This section presents several heuristic algorithms for reducing the value of $\sigma$. These algorithms differ in the amount of work that they perform, although it is not guaranteed that for all of the algorithms more work implies greater closeness to optimality. Note that in the case of fixed width fonts very little computation is necessary since most often it is the case that $d_i = 0$ which means that we are already at optimality for line $i$. Rarely does one find $d_i > 1$ unless the length of the line is quite short. Thus, for fixed width fonts, our measures and algorithms may result in the avoidance of a good deal of work. In the case of variable width fonts, $d_i$ is greater than or equal to 1 for most lines due to the small unit of measure (i.e., mills). In such cases the presence of a floating point capability would indicate that we might prefer to use as our measure

$$\sum_{i=1}^{n} (e_{i_1}^2/b_i)$$

which is the varying component of (2) when $e_{ij}$ is approximately equal to $e_i/b_i$ for all $j$, as is the case with a variable width font.

Any algorithm that we devise must have two phases. The first phase scans the paragraph in the forward direction filling each line with as many words as possible. The second phase processes the paragraph in the backward direction and attempts to move words from lines $i - 1$ to $i$. The exact number of words to be moved is governed by the amount of extra space that is available. Since we want to reduce the overall nonuniformity, we will not want to fill all of the extra space in a particular line as this will decrease the contribution to nonuniformity due to line $i$ at the expense of increasing the contribution of line $i - 1$.

The worst approach from a computational complexity point of view is one that enumerates all of the possible rearrangements of words. Clearly, the optimal solution will be found. As noted earlier, dynamic programming can be employed as an alternative with our measure serving as the cost function to be minimized. Given $n$ lines and a maximum of $K$ words that can be fit on a line, such an approach requires work on the order of $nK^2$, where each line serves as a stage in the dynamic programming solution and at each stage a $K$ by $K$ matrix of values must be computed. Dynamic programming is used by Knuth [3] in conjunction with a sixth order cost function. Knuth points out that an improved algorithm can be obtained whose running time is almost always of order $K$.

We propose a pair of heuristic algorithms that make use of our measure and may require considerably less work for a reasonably sized paragraph (i.e., $n$ and $K$ having the same order of magnitude) than an exhaustive method such as dynamic programming. In particular, our algorithms rely heavily on the optimality property of the measure. One algorithm, Algorithm 2, will visit each line in the backward direction only once, while the other, Algorithm 1, will make at most $n - i + 1$ visits to line $i$. Thus Algorithm 2 will make $2*n$ visits to all of the $n$ lines while Algorithm 1 will make a maximum of order $n^2$ visits to all of the $n$ lines. Both algorithms make use of a "move heuristic" termed MOVE (see Sec. 5) to decide how many words to move between adjacent lines. As we shall see, the algorithms have the property that the value of our measure will not increase although this is no guarantee that optimality will be attained.

Fig. 4. Sample Measure Values.

| | Fig. 1 | Fig. 2 | Fig. 3 |
|---|---|---|---|
| Measure (1), i.e., (5) | 11.3 | 0.67 | 2.07 |
| Measure (2), i.e., (4) | 81 | 39 | 49 |

Prior to describing the algorithms let us define the following terms in addition to $\sigma$:

$$\tau_i = \sum_{j=1}^{i-1} \sigma_j$$

$$\gamma_i = \sum_{j=i+1}^{n} \sigma_j$$

Note that $\tau_i$, $\sigma_i$, and $\gamma_i$ denote the contribution of lines 1 to $i-1$, line $i$, and lines $i+1$ to $n$, respectively, to the nonuniformity measure. For each line $i$, the first phase of any of our algorithms records $e_i$, $b_i$, $\sigma_i$, and $\tau_i$. The second phase records the value of $\gamma_i$ for each line $i$. Procedure MOVE($i$) has the effect of moving words from lines $i-1$ to $i$ and returns as its value the number of words moved. In order to keep track of the status of each line immediately after words have been moved in and out, MOVE records the additional information specified below. In the following, assume that MOVE has determined that $v$ words of total length $t$ (not including the mandatory blanks) are to be moved from lines $i-1$ to $i$.

$$e'_{i-1} \leftarrow e_{i-1} + v*s + t$$

$$b'_{i-1} \leftarrow b_{i-1} - v$$

$$\sigma'_{i-1} \leftarrow m'_{i-1} + d'_{i-1}*(e'_{i-1} + m'_{i-1})$$

$$e''_i \leftarrow e'_i - v*s - t$$

$$b''_i \leftarrow b'_i + v$$

$$\sigma''_i \leftarrow m''_i + d''_i*(e''_i + m''_i)$$

Single primes indicate the state of a line immediately after words have been moved out of it and before any words have been moved into it from a previous line. Double primes indicate the status of a line once words have been moved out of it and into it. Clearly, for the next to the last line, i.e., line $n$, in any paragraph $e'_n$, $b'_n$, and $\sigma'_n$ must be initialized to $e_n$, $b_n$, and $\sigma_n$, respectively.

Algorithm 1 is shown in Fig. 5.[2] The basic idea is to process the paragraph starting at lines $n$ and $n-1$ and to attempt, through the use of MOVE, to move words from lines $n-1$ to $n$. This procedure is repeated for lines $n-1$ and $n-2$, etc., until a pair of lines $k$ and $k-1$ are reached such that no words can be moved from lines $k-1$ to $k$. At this point, backtrack to the last line, say $j$, which had words moved into it without having any words moved out of it (i.e., line $n$ in the initial case). Next, find the maximum numbered line $i$, $k \le i \le j$, such that the sum $\tau_i + \sigma'_i + \gamma''_i$ is a minimum. Undo the MOVE operations between lines $k+1$ and $i-1$. Set $e''_i$, $b''_i$, and $\sigma''_i$ to $e'_i$, $b'_i$, and $\sigma'_i$, respectively, and reapply the algorithm starting at line $i-1$. Notice that the algorithm may make as many as $n-i+1$ visits to line $i$. Hence the $n$ lines may be visited a maximum of order $n^2$ times in the backward direction.

[2] We only describe the second phases of the rearrangement algorithms.

Fig. 5. Algorithm 1.

```
procedure ALGORITHM1(integer n);
begin
  integer i, last, m, temp;
  m ← n;
  while m > 0 do
    begin
      last ← m;
      $e'_{last} \leftarrow e_{last}$;
      $b'_{last} \leftarrow b_{last}$;
      $\sigma'_{last} \leftarrow \sigma_{last}$;
      while MOVE(m) NEQ 0 do m ← m − 1;
      temp ← $\tau_{last} + \sigma'_{last} + \gamma''_{last}$;
      for i ← last − 1 step − 1 until m do
        begin
          if temp > $\tau_i + \sigma'_i + \gamma''_i$ then
            begin      /*find a minimum*/
              temp ← $\tau_i + \sigma'_i + \gamma''_i$;
              last ← i;
            end;
        end;
      for i ← last step − 1 until m + 1 do UNDO(MOVE(i));
      $e''_{last} \leftarrow e'_{last}$;
      $b''_{last} \leftarrow b'_{last}$;
      $\sigma''_{last} \leftarrow \sigma'_{last}$;
      m ← last − 1;
    end;
end;
```

Algorithm 2, shown in Fig. 6, is very similar to Algorithm 1. Once again we apply procedure MOVE until a line $k$ is encountered such that no words can be moved from lines $k-1$ to $k$. The only difference is that once the line $i$ with minimum $\tau_i + \sigma'_i + \gamma''_i$ is found, all lines, $j$, $k-1 < j < i$, are left alone and the algorithm is reapplied starting at line $k-1$ rather than at line $i-1$ as is done in Algorithm 1. Notice that each line is visited at most once in the backward direction.

Fig. 6. Algorithm 2.

```
procedure ALGORITHM2(integer n);
begin
  integer i, last, m, temp;
  m ← n;
  while m > 0 do
    begin
      last ← m;
      $e'_{last} \leftarrow e_{last}$;
      $b'_{last} \leftarrow b_{last}$;
      $\sigma'_{last} \leftarrow \sigma_{last}$;
      while MOVE(m) NEQ 0 do m ← m − 1;
      temp ← $\tau_{last} + \sigma'_{last} + \gamma''_{last}$;
      for i ← last − 1 step − 1 until m do
        begin
          if temp > $\tau_i + \sigma'_i + \gamma''_i$ then
            begin      /*find a minimum*/
              temp ← $\tau_i + \sigma'_i + \gamma''_i$;
              last ← i;
            end;
        end;
      for i ← last step − 1 until m + 1 do UNDO(MOVE(i));
      $e''_{last} \leftarrow e'_{last}$;
      $b''_{last} \leftarrow b'_{last}$;
      $\sigma''_{last} \leftarrow \sigma'_{last}$;
      m ← m − 1;
    end;
end;
```

In each algorithm the statement "**while** MOVE($m$) NEQ 0 **do** $m \leftarrow m - 1$;" acts as a pruning device on the search since it identifies the segments of text which can be processed as independent blocks. Also, each algorithm guarantees that the value of the nonuniformity measure will not increase as a result of its application. This follows from the choice of a line in each independent block such that the sum $\tau_i + \sigma_i' + \gamma_i''$ is a minimum. Note that the starting value of the sum (i.e., $i = n$) is simply the value obtained when the conventional one-pass text justification algorithm is used.

In the case of fixed width fonts, we can make a further simplification of Algorithm 2. Recall that an optimal solution is one where $e_i$ is less than or equal to $b_i$ for all lines $i$. Therefore, we only attempt to justify in the backward direction lines whose value of $e_i$ exceeds that of $b_i$. Thus Algorithm 2 is modified so that it is only applied to the first such line that is found. The new algorithm, termed Algorithm 3, is shown in Fig. 7. Notice that in the case of variable width fonts this algorithm can still be applied; however, it is most often identical to Algorithm 2 due to the small units in which space is measured (e.g., mills).

Fig. 7. Algorithm 3.

```
procedure ALGORITHM3(integer $n$);
begin
  integer $i$, last, $m$, temp;
  $m \leftarrow n$;
  while $m > 0$ do
    begin
      if $e_m > b_m$ then
        begin
          last $\leftarrow m$;
          $e'_{\text{last}} \leftarrow e_{\text{last}}$;
          $b'_{\text{last}} \leftarrow b_{\text{last}}$;
          $\sigma'_{\text{last}} \leftarrow \sigma_{\text{last}}$;
          while MOVE($m$) NEQ 0 do $m \leftarrow m - 1$;
          temp $\leftarrow \tau_{\text{last}} + \sigma'_{\text{last}} + \gamma''_{\text{last}}$;
          for $i \leftarrow$ last $- 1$ step $- 1$ until $m$ do
            begin
              if temp $> \tau_i + \sigma_i' + \gamma_i''$ then
                begin      /*find a minimum*/
                  temp $\leftarrow \tau_i + \sigma_i' + \gamma_i''$;
                  last $\leftarrow i$;
                end;
            end;
          for $i \leftarrow$ last $- 1$ step $- 1$ until $m + 1$ do
            UNDO(MOVE($i$));
          $e''_{\text{last}} \leftarrow e'_{\text{last}}$;
          $b''_{\text{last}} \leftarrow b'_{\text{last}}$;
          $\sigma''_{\text{last}} \leftarrow \sigma'_{\text{last}}$;
        end;
      $m \leftarrow m - 1$;
    end;
end;
```

## 5. Move

In the previous section we saw the need for a mechanism to decide how many words, if any, to move from lines $i - 1$ to $i$ in attempting to reduce the value of the nonuniformity measure. There is often a choice. A right choice means that there is no need to employ an algorithm which uses backtracking. This section describes some possible decision mechanisms.

The simplest decision mechanism is termed the "greedy mechanism." It moves as many words as possible from lines $i - 1$ to $i$, and likewise from lines $i - 2$ to $i - 1$, and so on until a line $j$ is found such that no words can be moved from lines $j - 1$ to $j$. Such an approach can be likened to a pyramid with the result that line $j$ will have a large value of $e_j'$ which will have an undesirable effect on the value of our measure. Our proposed mechanisms are aimed at reducing the "greediness" at each line. However, it should be clear that if only one word can be moved, then greediness may prevail. Hence our solutions are geared to situations where there is a choice of how many words to move. Three possible decision mechanisms are given below.

(1) For each line the number of words to be moved is chosen to be the ceiling of one-half of the maximum number of words that can be moved.
(2) Move a number of words $m$ from lines $i - 1$ to $i$, such that moving $m$ words results in $e_i'' \leq e_i/2$ and moving $m - 1$ words results in $e_i'' > e_i/2$.
(3) Move a number of words $m$ from lines $i - 1$ to $i$, such that moving $m$ words results in $e_{i-1}'' \leq 2*e_{i-1}$ and moving $m + 1$ words results in $e_{i-1}'' > 2*e_{i-1}$.

Mechanism (1) is independent of the amount of space moved. There is no way to distinguish between short words and long words. Thus when words are not of uniform length, we may tend to move too much or too little from one line to another depending on whether there is a sequence of long or short words, respectively, at the end of the line in question. Mechanisms (2) and (3) are 50 percent rules. Mechanism (2) results in a 50 percent decrease in the amount of extra space in line $i$, while mechanism (3) results in a 50 percent increase in the amount of extra space in line $i - 1$. We feel that mechanism (2) is the preferred of the alternatives proposed since its computation only depends on the current line and not on what happens in the future; i.e., mechanism (3) must also take into account the effect of moving words from lines $i - 2$ to $i - 1$, etc. Note that a variation of mechanism (3) with $e'$ substituted for $e''$ is undesirable because lines with small values of $e_{i-1}$ will have fewer words moved than lines with larger values of $e_{i-1}$. However, often this is the opposite of what should happen since frequently small values of $e_{i-1}$ mean that more words can be moved to line $i$ with less deletorious effects on the nonuniformity measure.

## 6. Concluding Remarks

A measure and related algorithms for reducing the nonuniformity of computer justified text have been presented. The measure has been shown to be computationally simple as well as to have an attainable lower bound.

Once again, we reiterate that nonuniformity is an aesthetic property and thus our results are essentially only valid if one adopts our measure. Nevertheless, visual inspection of the result of its application does not refute its reasonableness. The ultimate test lies in psychological experiments.

The presence of a hyphenation capability will probabily result in less of a need for the measure although it remains useful in applications such as newspapers and magazines where columns are relatively narrow. However, such a hyphenation capability does make our heuristic algorithms more attractive than the dynamic programming approach since their execution time is independent of the maximum number of word segments in a line (i.e., $K$). In particular, Algorithm 3 with its check for optimality for each line should yield the best results in terms of execution speed.

*Acknowledgments.* I have benefitted from discussions with P. Agre, G. Knott, and S. Peleg. I would also like to thank S. Peleg and R. Rivest for pointing out dynamic programming as a possible solution.

Received 5/79; revised 3/81; accepted 10/81

**References**
1. Hillier, F.S., and Lieberman, G.J. *Introduction to Operations Research.* Holden-Day, San Francisco, 1967.
2. Kendall, M. G., and Stuart, A. *The Advanced Theory of Statistics,* Vol. 2, 3rd ed., Hafner Press, New York, 1973.
3. Knuth, D.E. and Plass, M.F., Breaking paragraphs into lines. *Software Practice and Experience.* (Nov. 1981) 1119–1184.
4. Knuth, D.E. Tau Epsilon Chi, a system for technical text. Stanford Computer Science Rep. CS675, Stanford University, Stanford, CA, Sept. 1978.
5. Mood, A.M., Graybill, F.A., and Boes, D.C. *Introduction to the Theory of Statistics,* 3rd. ed. McGraw-Hill, New York, 1963.
6. Reid, B.K., and Walker, J.H. SCRIBE, introductory user's manual, 2nd ed. Carnegie-Mellon University, Pittsburgh, PA, 1979.

Programming Techniques and Data Structures

M. Douglas McIlroy* Editor

# An Efficient Garbage Compaction Algorithm

Johannes J. Martin
Virginia Polytechnic Institute and State University

**The garbage compaction algorithm described works in linear time and, for the most part, does not require any work space. It combines marking and compaction into a two-step algorithm that is considerably faster than, for example, Morris's method. The first step marks all nongarbage cells and, at the same time, rearranges the pointers such that the cells can be moved; the second step performs the actual compaction.**

**CR Categories and Subject Descriptors:** D.4.2 [Operating Systems]: Storage Management—*allocation/deallocation strategies.*

**General Terms:** Algorithms, Theory

**Additional Key Words and Phrases:** garbage collection, compaction, relocation, storage reclamation

## 1. Introduction

In a storage area divided into cells of possibly different sizes where some but usually not all are accessible by the user's program, a garbage compactor can move the accessible cells to one end of the storage area and update all pointers to these cells (pointers stored in cells as well as those that point to cells from the outside) such that they point to the new locations of the cells. At the other end of the storage area, this process creates a block of adjacent, unused fields as large as the sum of the fields occupied by all inaccessible (garbage) cells. Algorithms for this task consist of two phases. The first phase identifies all accessible cells by marker bits; the second phase performs the actual compaction.

* Former editor of Programming Techniques and Data Structures, of which Ellis Horowitz is the current editor.

Author's Present Address: Johannes J. Martin, Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061.