UT Libraries Electronic Document Delivery Service

Thank you for using ILLiad and UT's electronic document delivery service. Attached, please find a portable document format (pdf) version of an item you requested through Interlibrary Services (www.lib.utk.edu/~ils).

- This item will remain in your ILLiad account for thirty (30) days. After 30 days, ILLiad will automatically delete the file, removing the item from your account. You may delete this item any time prior to end of the 30-day expiry period.
- As Interlibrary Services (ILS) performs only minor editing of electronic documents prior to delivery (e.g. rotating, resorting pages), ILS delivers items 'as is.' ILS can't control the quality of the initial imaging performed by the lending library. ILS passes along what the lending library provides. If the quality of this document fails to meet your needs, let ILS know and ILS will find a remedy.
- Refer questions and comments concerning electronic delivery and ILLiad to ILS at ilsmail@aztec.lib.utk.edu or (865) 974-4240.

REQUEST FOR A COPY

Request ID: 9681300 (OCLC:TKN)

Request date: 10/28/04
Will pay fee:
Copyright Compliance: US:CCL
Deliver via: Your standard

Need before: 11/21/04
Reciprocal agreement:
Required format type:
Level of service:

Max cost: 35.00 (USD)
Payment provided:
Payment type: ~~IFM~~
Expiry date: 11/02/04

Call number:
IWCC$
Author: ACM SIGPLAN SIGOA Symposium on Text Manipulation (1981 : Portland, Or.)
Title: Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation, Portland, Oregon, June 8-10, 1981.
Edition:
Series and number: SIGPLAN notices ; v,16, no. 6, June 1981.
Published: New York, N.Y. : Association for Computing Machinery, c1981.

Author of excerpt: James O. Achugbue: On the line breaking problem in text formatting
Volume/issue: (1981)
Pagination: 117-122
ISBN:
ISSN:
Other ID numbers: System #:OCLC:7595448
Verification source: <TN:369053>OCLC [Format: Book]

Y 10. INV

Requester notes: Requester Number--(OCLC) ILLNUM:9681300; FAX/ARIEL:(865) 974-2708 //ARIEL 160.36.192.95 or libpc7.lib.utk.edu EMAIL:ilsmail@aztec.lib.utk.edu [Messages Only]

Responder name:
Responder ID: OCLC:TEU

Math

Document Delivery / Interlibrary Loan
Temple Univ./Samuel Paley Library (017-00)
1210 W. Berks Street
Philadelphia, PA 19122
USA

HF5548.115.A22X 1981

Date sent:
Sent via:
Charges:
# of pages:

Responder notes:

Requester name:
Requester ID: OCLC:TKN

Account number:

Ship to:
Interlibrary Services-John C. Hodges Library/1015 Volunteer Blvd./University of Tennessee/Knoxville,TN 37996-1000

Bill to:
same

Deliver via: Your standard

Client name: Eijkhout, Victor :STAT: Faculty :DEPT: Computer Science
Status:
ID:

Transaction ID: OCLC:TKN-9681300:TEU

ON THE LINE BREAKING PROBLEM IN TEXT FORMATTING

James O. Achugbue

Department of Mathematical and Computer Sciences
Michigan Technological University, Houghton MI

ABSTRACT

A basic problem in text formatting is that of determining the break points for separating a string of words into lines to obtain a formatted paragraph. When formatted text is required to be aligned with both the left and right margins, the choice of break points greatly affects the quality of the formatted document. This paper presents and discusses solutions to the line breaking problem. These include the usual line-by-line method, a dynamic programming approach, and a new algorithm which is optimal and runs almost as fast as the line-by-line method.

KEYWORDS

Text formatting, line breaking, text alignment, computer typesetting, dynamic programming.

1. INTRODUCTION

In this paper, we are concerned primarily with the design of algorithms to format the text of a paragraph for printing on an output device with fixed character positions, such as a line-printer. This is to be contrasted with printing on output devices with arbitrary character positions, such as graphic terminals and typesetting equipment. The former is usually referred to as text formatting while the term typesetting is applied to the latter type. Typesetting is the main concern of systems such as [3,4,5,6]. While this unquestionably yields documents of professional quality, the equipment required is nevertheless unavailable to the majority of potential users. Thus, any features which aim at improving the quality of text-formatting are most welcome. A useful feature, often supplied by designers of text formatting software, for example [7,8,9], is the ability to align or justify the formatted text with both left and right margins. Text alignment poses two main problems. First, the determination of the break points for separating the words of a paragraph into lines, and second, the distribution of the surplus spaces on each line in between the words of that line.

The first problem is usually solved by filling up each line as much as possible and then proceeding to the next. This method will be reviewed in the next section and used as a basis for the subsequent development. While it is simple to implement and works well in many situations, it does not always produce the best results when text is being justified.

A common strategy for dealing with the second problem is to distribute the surplus spaces for each line between the words starting alternately from the left and right margins. Thus, if line one has additional spaces, then the spaces between the first and second words, the second and third words and so on will be increased by one until the surplus spaces are used up. If line two also has surplus spaces, it is now used up by increasing spaces between the last and the last but one words, the last but one and the preceding word, and so on. The aim of this strategy is to avoid "rivers" of white space running down the length of the page. Another possibility would be to assign the extra spaces to the inter-word gaps in a pseudo-random manner. However, irrespective of the surplus space distribution strategy, a poor line breaking algorithm will undoubtedly frequently produce poorly formatted text. This paper therefore focusses attention on the line breaking problem.

Another important factor which should be taken into consideration in connection with this problem is that of hyphenation. No doubt, a good hyphenation algorithm will help improve the quality of formatted text. However, hyphenation in itself is a complex problem and will be left out for most of this presentation.

In sections 2 to 4 the algorithms are presented for formatting. Section 5 discusses extensions to typesetting and hyphenation.

2. THE LINE-BY-LINE METHOD

It is assumed that we have as input a paragraph consisting of a sequence of N>0 words. Here, a word is simply a string of non-blank characters. The number of characters in each word is given in the array W[I], 1<=I<=N. The paragraph will be formatted into M lines of D characters each. The line breaking problem is solved by specifying for the J-th line, 1<=J<=M, the index S[J] of tne first word of the line. (The major variables used by all the algorithms are summarized in table 1.)

```
---------------------------------------------------------------
      N - number of words in paragraph
      M - number of formatted lines
      D - maximum number of characters per line
   W[I] - number of characters in the I-th word
   S[I] - index of first word in I-th line,
           that is, I-th line starts with W[S[I]].
   L[I] - length of I-th formatted line before
           distribution of surplus spaces.
   E[I] - index of first word, line I, for earliest
           breaking
 F[I,J] - formatted length from I-th to J-th word
 C[I,J] - cost function, dynamic programming
   c[I] - cost function, line-breaker, = C[I,N]
---------------------------------------------------------------
```

TABLE 1:
Major variables referenced by the algorithms.

The line-by-line method is the one that immediately comes to mind and has been used in many text formatting programs. It is strongly appealing in its simplicity. The computation of the break points or equivalently the indices for words at the beginning of each line is given in algorithm LINE-BY-LINE. Note that arrays L, S and W need not be saved in actual implementations unless they are required for some other purposes. They are kept in this presentation to facilitate the discussion in subsequent sections. Clearly, then, the line-by-line method can be implemented so that it has O(N) worst case time complexity and requires storage mainly for the one line of output. (The algorithms are also given in an PASCAL-like fashion in the appendix).

ALGORITHM LINE-BY-LINE

```
---------------------------------------------------------------
    /D,L,M,N,S,W are as given in Table 1/
(1) /initialize/
    M <- 1,  S[1] <- 1,  L[M] <- W[1]
    I <- 2
(2) /add word to line/
    L[M] <- L[M] + 1 + W[I]
    if L[M] <= D then goto (4)
(3) /start new line/
    L[M] <- L[M] - 1 - W[I]
    M <- M + 1,  S[M] <- I,  L[M] <- W[I]
(4) /test for completion/
    I <- I + 1,  if I <= N then goto (2)
---------------------------------------------------------------
```

The effect of algorithm LINE-BY-LINE on a short sample paragraph from [6], formatted 47 characters to a line, is given below. The surplus spaces in this paragraph have been distributed according to the alternate left and right fashion. Note that of the seven lines in the paragraph, the first five have 1, 2, 6, 10, 7 surplus spaces respectively. This means that in the fourth line, 10 spaces have to be distributed between six words, resulting in triple spacing between some of them. Pathological cases similar to this and worse abound in the literature. It would seem that this formatting can be improved by transfering the last words of the first, second, third and fourth lines to the beginning of the second, third, fourth and fifth lines respectively. This has been done in Sample Paragraph #2 where the surplus spaces are obviously more evenly distributed among the lines. In fact, the first paragraph has eight occurences of triple spacing compared to two in the second.

Sample Paragraph #1:

```
We  live in a print-oriented society. Every day
we produce a huge volume of  printed  material,
ranging   from  handbills  to  heavy  reference
books.  Despite   the   mushroom   growth   of
electronic   media,   print  remains  the  most
versatile and most widely used medium for  mass
communication.
```

Sample Paragraph #2:

```
We  live  in  a  print-oriented  society. Every
day  we  produce  a  huge  volume  of   printed
material,   ranging  from  handbills  to  heavy
reference books. Despite  the  mushroom  growth
of  electronic  media,  print  remains the most
versatile and most widely used medium for  mass
communication.
```

3. A DYNAMIC PROGRAMMING SOLUTION

The improvements to the sample paragraph indicated in the preceding section demonstrate that line breaking as done by the line-by-line method does not always produce the best results. In this section, a dynamic programming solution for optimal line breaking is presented. This idea is not new. Knuth [5] indicates that he uses such an approach for line breaking in his typesetting system.

The key to the improvements in the preceding section arise from the fact that when a sequence of words has to be broken into two or more lines it should be broken in such a way that the lines are equally used up or very nearly so. The idea is to eliminate extreme variations in the amount of surplus space to be distributed among the lines. In other words, justified text will look better if the unjustified version has minimum raggedness.

A second important idea to keep in mind is that if a sequence of words will fit on one line, then there is no point in, and no attempt should be made at, splitting it into several lines.

These considerations lead to the following definitions. First, the formatted length F[I,J] of words I to J is defined as the width that the words will occupy. Thus

F[I,J] = W[I] + 1 + W[I+1] + 1 + ... + W[J].

Second, the following cost function is suggested for minimization.

$$C[I,J] = \begin{cases} 2 & \text{if } F[I,J] <= D \ \& \ J = N \\ 1+1/F[I,J] & \text{if } F[I,J] <= D \ \& \ J < N \\ 1+\min(C[I,K] * C[K+1,J],\ I <= K < J), & \text{otherwise.} \end{cases}$$

The cost function, C[I,J], will be the cost of formatting words I to J. It recognizes the fact that the last line of a paragraph need not be (and is not normally) aligned with the right margin. This case is recognized by the condition J=N. It also attempts no splitting of a sequence of words that will fit on one line. Such a sequence simply contributes a factor of 1+1/F[I,J] to the cost of the paragraph. When a split has to be made, however, the break point is chosen from all possible candidates so as to minimize the overall cost.

The discussion following presentation of the line-by-line algorithm suggests the following definition of optimally formatted text which we shall adopt. A paragraph W[1]...W[N] is optimally formatted if it is broken into the fewest number of lines and the surplus spaces on each line, not counting the last line, are as close together as possible.

We argue that minimizing C[1,N] will result in an optimally formatted paragraph. First note that if a line is split into two the cost function will increase. Hence, a paragraph with minimum C[1,N] will have the fewest number of lines. Secondly, we show that the function is minimized by having equal length lines (not counting the last one). Let the line lengths for the first m-1 lines be x[1], x[2], ... x[m-1]. The final cost is twice the product of (1+1/x[i]), 1<=i<=m-1. Now, given that a+b is constant it is straightforward to prove that (1+1/a)(1+1/b) is minimal when a=b. In the general case when m>3, assume the lengths x[i], 1<=i<=m-1 are optimal and x[j] is not equal to x[j+1] for some j. Then we can lower the overall cost by keeping the other x[i] and replacing x[j], x[j+1] by (x[j]+x[j+1])/2. This contradicts the fact that we had the minimum cost.

In order to compute the optimal breaking indices, note that if W[I]..W[J] has to be broken into W[I]...W[K] and W[K+1]...W[J], then both subsequences W[I]...W[K] and W[K+1]...W[J] must each be optimally formatted this time taking into consideration the last line of any subsequence which is not the last line of the paragraph. The computation of optimum cost C[1,N] is given in algorithm DYNAMIC. It is similar to many other dynamic programming algorithms, [1,2] for example, and the modifications required to keep track of the breaking indices is a straightforward exercise.

It is also straightforward to determine that algorithm DYNAMIC takes O(N**2) space and O(N**3) time. The algorithm is thus too costly for regular use and one would rather put up with the poorer results of line-by-line processing. However, by combining features of this algorithm and the line-by-line one, a much faster optimal solution can be devised.

As expected, application of the dynamic programming approach to the sample paragraph yields the much improved version given in Sample Paragraph #2.

ALGORITHM DYNAMIC

```
----------------------------------------------------
    /computation of C[1,N]/
    /Only upper diagonal of C computed/
    /C, D, F, N and W are explained in table 1/
(1) /initialize/
    C[I,J] <- F[I,J] <- 0,  1<=I<=N,  1<=J<=N.
    F[I,I] <- W[I],  C[I,I] <- 1+1/W[I],  1<=I<=N.
    I <- N - 1
(2) /loop on rows from last to first/
    J <- I + 1
(3) /loop on columns from I+1 to N/
    /calculate length/
    F[I,J] <- F[I,J-1] + 1 + W[J]
    if F[I,J] > D then goto (5)
(4) /words I to J fit on one line/
    if J = N then C[I,J] <- 2
            else C[I,J] <- 1 + 1/F[I,J].
    goto (6)
(5) /split words I to J/
    C[I,J] <- minimum{C[I,K] * C[K+1,J], I<=K<J}
(6) /end loop on columns/
    J <- J + 1,  if J <= N then goto (3)
(7) /end loop on rows/
    I <- I - 1,  if I > 0 then goto (2)
----------------------------------------------------
```

4. THE LINE BREAKER

We begin this section by first proving a simple but important property of the line-by-line algorithm. Let P[I], 1<=I<=M be the indices for optimal starting words in a paragraph. Then, P[I]<=S[I], 1<=I<=M. (Recall that S[I] are determined by algorithm line-by-line.) This property is easily shown by contradiction as follows. Let J be the smallest index for which P[J] > S[J]. Clearly, P[1]=S[1]=1. It is also trivial that P[2]<=S[2]. Thus, J>2. Now, words P[J-1],...,(P[J]-1) must fit on one line. Since P[J-1]<=S[J-1] and P[J]>S[J], it follows that words S[J-1],...,S[J] also fit on one line. This is not possible, since by the line-by-line method, word S[J] could not be added to the (J-1)-th line. Hence, no J exists for which P[J]>S[J].

A direct consequence of the above property is that algorithm line-by-line formats the paragraph into the minimum number of lines possible, thus satisfying part of the conditions for optimal formatting. Intuitively, this is to be expected from the greedy approach. More importantly, this means that the improvements obtainable by using algorithm dynamic are solely by breaking some lines earlier, without further changes in the actual number of lines. Thus, in computing C[1,N], we may seek the first optimal break point between words S[1] and S[2]. Another way of looking at it is that one should seek for the position to break off one line at a time. Using this approach, the computation of C[1,N] can be speeded up to O(N**2) time. However, there are further improvements to be had and we shall proceed a little differently.

Breaking with the set of indices S[1],...,S[M] provides latest breaking points for formatting in the minimum number of lines, M. Now, consider breaking with the earliest breaking indices, E[I], that result in M formatted lines. If P[I] gives the indices for an optimal set of break points, then P[M]=S[M] since clearly there is no gain in pushing any words from the (M-1)-th line to the M-th line. It is also clear that the optimal breaking index P[I] lies between E[I] and S[I] and since P[M]=S[M] one might as well choose E[M]=S[M]. The remaining indices E[1],...,E[M-1] are then the earliest breaking indices for formatting words 1 to (S[M]-1) into M-1 lines and are computed by performing the line-by-line algorithm in reverse, that is, scanning the words in reverse order and filling up the last line and then the last but one line and so on.

Thus, the optimal set of breaking indices P[I] satisfy the condition E[I]<=P[I]<=S[I] where E[1]=P[1]=S[1]=1 and E[M]=P[M]=S[M]. We have therefore significantly reduced the search regions for the optimal starting indices P[I]. The improved computation of an optimal set of starting indices is given in algorithm LINE-BREAKER. It is assumed that the latest breaking points S[I] and associated line lengths L[I] have been computed in the line-by-line fashion, and the earliest indices E[I] by the reverse process as explained above. The algorithm uses the same cost function as for DYNAMIC but computes only a small number of the matrix entries. The order of computation is C[J,N], J=E[M-1]...S[M-1], then C[J,N], J=E[M-2]...S[M-2] and so on until C[1,N]. Since the second index is always the same (only elements from the last column are computed) only the the second index J is used, giving the cost c[J]=C[J,N]. Note that in computing the cost c[J] the search is done for the next breaking index (breaking off one line at a time) which is known to lie in the range E[I+1]...S[I+1]. Note further that it has also been arranged to compute the length function only where necessary.

The correctness of algorithm line-breaker can be surmised from the preceding discussion. The algorithm clearly uses O(N) space principally for the required linear arrays.

Define the slack on the I-th line to be T[I]=S[I]-E[I]. In the LINE-BREAKER algorithm, the loop on I is performed for M-1 iterations and within this loop, the loop on J is performed for T[I] iterations and within the latter there is yet another loop on K which is performed for T[I+1] iterations. These loop operations thus require O(T[1]T[2] + T[2]T[3] + ... + T[M-1]T[M]) time. One may use as a very loose upper bound the maximum slack, v, and bound the above expression with Mv**2. Since the computation of latest and earliest breaking indices and the initial line lengths, L[I], as well as the final recovery of the optimal set of breaking indices are each linear in N, we obtain a total running time of O(N+M(V**2)).

It should be stressed that the search for optimal breaking has thus been reduced to searching among those words which can possibly be moved from their initially assigned lines to the next one without increasing the number of lines, that is, the words which constitute the slack on the I-th line. Full advantage is taken of the work done by the simple line-by-line procedure. In practice, the slack per line is a very small number so that the algorithm is almost linear in behavior.

ALGORITHM LINE-BREAKER

```
--------------------------------------------------------
    /fast computation of optimal breaking indices
            P[1],P[2],...,P[M],  M > 2.
     Assume S[I] - latest breaking indices
            E[I] - earliest breaking indices, and
            L[I] - lengths of lines from the
                   line-by-line approach
                   have been computed.
        INFINITE - any number larger than maximum
                   possible c[I]./
(0) /initialize/
    I <- M - 1,  c[S[M]] <- 2
(1) /loop on lines, backwards/
    X <- L[I] - 1 - W[S[I]]
    J <- S[I]
(2) /loop over I-th slack/
    X <- X + 1 + W[J],  Y <- X + 1 + W[S[I+1]]
    c[J] <- INFINITE
    K <- S[I+1]
(3) /loop over (I+1)-th slack/
    Y <- Y - 1 - W[K],  if Y > D then goto (5)
    Z <- (1 + 1/Y) * c[K]
    if Z >= c[J] then goto (5)
(4) /update c[j]/
    c[J] <- z,  P[J] <- K
(5) /end loop over (I+1)-th slack/
    K <- K - 1,  if K >= E[I+1] then goto (3)
(6) /end loop over I-th slack/
    J <- J - 1,  if J >= E[I] then goto (2)
(7) /end loop on lines/
    I <- I - 1,  if I > 0 then goto (1)
(8) /retrieve optimal breaking indices/
    J <- P[1],  P[1] <- 1,  P[M] <- S[M],  I <- 2
(9) /retrieve loop/
    K <- P[I],  P[I] <- J,  J <- K
    I <- I + 1,  if I < N then goto (9)
--------------------------------------------------------
```

5. EXTENSIONS

Although this investigation was originally prompted by the need for improved formatting, the resulting algorithm can be extended in a straightforward manner to deal with variable pitch fonts for typesetting. This requires redefining W[i] to be the width of word i rather than the number of characters in it. Similarly, redefine D to be the line width and replace "1" for space widths by the minimum allowed width for a space in a justified line.

Algorithm LINE-BREAKER could also be extended for use with an automatic hyphenation process in a manner that would minimize the words considered for breaking. Assuming the hyphenation points, if any, within each word are given, one possibility would be to ignore these points in the initial computation of latest and earliest breaking indices, and take them into consideration in the final stage when attempting to equalize the formatted lengths of the lines. This approach will not always produce the absolute minimum number of lines but as a trade-off fewer words will be considered for breaking.

6. CONCLUSIONS

Three algorithms for the line breaking problem have been discussed in this paper. The line-by-line method is simple but often produces undesirable results. The dynamic programming approach generally results in costly computations. By combining features of both methods, a new hybrid algorithm, LINE-BREAKER, is produced, which guarantees optimal results for the type of printing environments under consideration, uses much less space than dynamic programming and is almost as fast as the basic line-by-line processing. Line-breaker is sufficiently fast for regular use when text is to be justified.

These algorithms have so far been tested on a stand alone basis. It is expected that the LINE-BREAKER approach will be incorporated in future text formatting systems.

ACKNOWLEDGEMENTS

The author is grateful to the referees for their very helpful suggestions and to his colleagues Karl Ottenstein and John Lowther for proof-reading several versions of the paper.

REFERENCES.

[1] A. V. Aho, J. E. Hopcroft & J. D. Ullman, The Design and Analysis of Computer Algorithms, Addison Wesley, 1974, p69.

[2] K. Q. Brown, Dynamic Programming in Computer Science, Tech. Rep. CMU-CS-79-106, Dept of Computer Sc., Carnegie-Mellon Univ., Pittsburgh Pa, 1979.

[3] B. W. Kernighan, M. E. Lesk, J. F. Ossanna, Unix Time Sharing System: Document Preparation, Bell System Technical Journal 57(6), July-Aug 1978, pp2115-2134.

[4] B. W. Kernighan, L. L. Cherry, A System for Typesetting Mathematics, CACM 18, March 1975, pp151-157.

[5] D. E. Knuth, TAU EPSILON CHI, A System for Technical Text, American Mathematical Society, Providence, Rhode Island, 1979.

[6] J. Sachs, Economical Typesetting from Small Computer Text Files, Proc of the Third Symposium on Small Computers, sponsored by ACM SIGSMALL & SIGPC, Sept 1980, Palo Alto, California, pp184-188.

[7] J. Pearkins, FMT Rference Manual, University of Alberta Computing Services, April 1976.

[8] TXTFORM - A Text Formatter, Dept of Computer Science, Purdue University, West Lafayette, Ind., 1979.

[9] DOC Processor Bulletin, Academic Computing Services, Michigan Technological University, Houghton, MI.

APPENDIX

```
PROCEDURE LINE-BY-LINE;
(* D,L,M,N,S,W ARE EXPLAINED IN TABLE 1 *)

BEGIN

   (* INITIALIZE *)
   M := 1;
   S[1] := 1;
   L[M] := W[1];

   FOR I := 2 TO N DO
   BEGIN

      (* ADD NEXT WORD TO CURRENT LINE *)
      L[M] := L[M] + 1 + W[I];

      IF (L[M] > D) THEN
      BEGIN
         L[M] := L[M] - 1 - W[I];

         (* START NEW LINE *)
         M := M + 1;
         S[M] := I;
         L[M] := W[I]
      END
   END
END;
```

```
PROCEDURE DYNAMIC;

(* COMPUTATION OF OPTIMAL COST C[1,N]
   C[I,J], F[I,J] EXPLAINED IN TABLE 1. *)

BEGIN

   (* INITIALIZE VARIABLES *)
   FOR I := 1 TO N DO
   BEGIN
      FOR J := 1 TO N DO
      BEGIN
         F[I,J] := 0;
         C[I,J] := 0
      END;
      F[I,I] := W[I];
      C[I,I] := 1 + 1 / W[I]
   END

   (* COMPUTE UPPER DIAGONAL OF L AND C
      IN REVERSE ROW ORDER *)
   FOR I := N-1 DOWNTO 1 DO
   BEGIN
      FOR J := I+1 TO N DO
      BEGIN

         (* CALCULATE FORMATTED LENGTH *)
         F[I,J] := F[I,J-1] + W[J] + 1;
         IF F[I,J] <= D THEN
         BEGIN

            (* WORDS I TO J FIT ON LINE *)
            IF J = N THEN C[I,J] := 2
                     ELSE C[I,JM := 1 + 1 / F[I,J]
         END
         ELSE
         BEGIN

            (* WORDS I TO J HAVE TO BE SPLIT *)
            C[I,J] := C[I,I] * C[I+1,J];
            FOR K := I+1 TO J-1 DO
            BEGIN
               T := C[I,K] * C[K+1,J];
               IF T < C[I,J] THEN C[I,J] := T
            END
         END
      END
   END
END;
```

```
PROCEDURE LINE-BREAKER;

(* COMPUTATION OF OPTIMAL STARTING INDICES P[I]
   FOR M > 2.
   ASSUME S[I], E[I], L[I] (DEFINED IN TABLE 1)
   HAVE BEEN COMPUTED.  X,Y,Z ARE USED TO KEEP
   TRACK OF REQUIRED LENGTHS.
   INFINITE IS ANY NUMBER LARGER THAN MAXIMUM
   POSSIBLE COST c[I]   *)

BEGIN

   c[S[M]] := 2.0;

   (* LOOP ON LINES BACKWARDS *)
   FOR I := M-1 DOWNTO 1 DO
   BEGIN

      X := L[I] - 1 - W[S[I]];

      (* LOOP OVER I-TH SLACK *)
      FOR J := S[I] DOWNTO E[I] DO
      BEGIN

         X := X + 1 + W[J]];
         Y := X + 1 + W[S[I+1]];
         c[J] := INFINITE;

         (* LOOP OVER (I+1)-TH LACK *)
         FOR K := S[I+1] DOWNTO E[I+1] DO
         BEGIN
            Y := Y - 1 - W[K];
            IF Y <= D THEN
            BEGIN

               (* UPDATE c[J] *)
               Z := (1 + 1 / Y) * c[K];
               IF Z < c[J] THEN
               BEGIN
                  c[J] := Z;
                  P[J] := K
               END
            END
         END
      END
   END;

   (* RETRIEVE OPTIMAL STARTING INDICES *)
   P[M] := S[M];  J := P[1];  P[1] := 1;
   FOR I := 2 TO M-1 DO
   BEGIN
      K := P[I];  P[I] := J;  J := K
   END
END;
```