CS 157: Assignment 5

Douglas R. Lanman
24 April 2006

Problem 2: Printing Neatly

In a word processor or in LaTeX, one routinely encounters the "pretty printing" problem. That is, how does one transform text with a ragged right margin into text whose right margin is as "even" as possible. This write-up discusses the design and analysis of an efficient dynamic program for "pretty printing" and is composed of four sections. In Section 1, we define the "slack cost" (the cost associated with printing a set of words on a line) and provide an efficient algorithm for computing it. In Section 2, we apply this algorithm to the general problem of "pretty printing". The running time and storage requirements of this algorithm are presented in Section 3. Finally, we conclude with a discussion of the applications and nuances of "pretty printing".

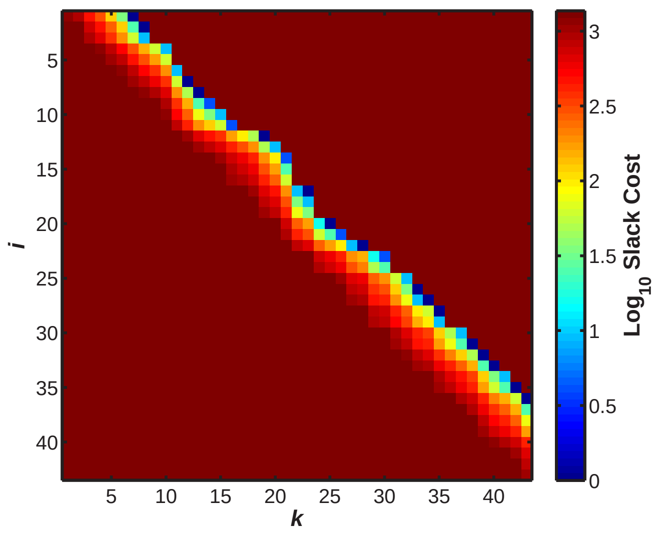
1 Pre-computing the Slack Cost Table

The "pretty printing" problem has previously been solved using dynamic programming by Donald Knuth for the TeX document preparation system [1, 2]. We follow a similar approach in this write-up – focusing on the merits of a dynamic solution. To begin our analysis, let's review the formal definition of "pretty printing".

In general, we can represent a piece of text by an array W, whose elements w_i are the length of each word in characters. In addition, we can define a general display device by a maximum line length L. To simplify the analysis, we consider the font to be fixed-width and that we are not allowed to break (i.e., hyphenate) words across lines. With these restrictions, a "formatting" of the text is defined as a partition of W into separate lines.

```
Call me Ishmael. Some years ago, never
mind how long precisely, having little
or no money in my purse, and nothing
particular to interest me on shore, I
thought I would sail about a little
and see the watery part of the world.
```

(a) Example of neatly-printed text



(b) Corresponding *slack cost table*

Figure 1: Example of a pre-computed *slack cost table* generated by SLACK-TABLE for the 43 word text excerpt shown at the left. The maximum line length $L = 38$ for this example.

For this analysis, we assume that a single space is printed after each word on a line except the last one. As a result, the total number of printed characters on a line is given by

$$N[i,k] \equiv \sum_{i \le j \le k} W[j] + (k-i) \tag{1}$$

which results from printing words i through k on a single line. We select the definition provided by Kleinberg and Tardos for "even" text; that is, we seek a partition such that the sum of the squares of the "slacks" is minimized. Here the "slack" is defined as the distance in characters from the end of the line and is computed as follows.

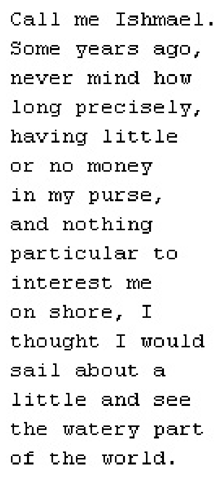
$$S[i,k]^2 \equiv \begin{cases} (L - N[i,k])^2 & L - N[i,k] \ge 0 \\ \infty & \text{otherwise} \end{cases} \tag{2}$$

Note that the slack is defined as infinite when the number of characters is greater than the line length – representing our desire to never print past the end of a line.

In the process of computing the optimal partition of words into lines, we will be required to evaluate Equation 1 repeatedly. As a result, it is best to pre-compute a *slack cost table*, such that slacks can be looked up in constant time in our main routine. As shown below, the Slack-Table procedure is a simple dynamic program for pre-computing the values of $S[i,k]\ \forall(i,k)$. The input to Slack-Table is the word length array W and the maximum line length L. The output is the slack cost table S. An example of a pre-computed slack cost table is shown in Figure 1(b).

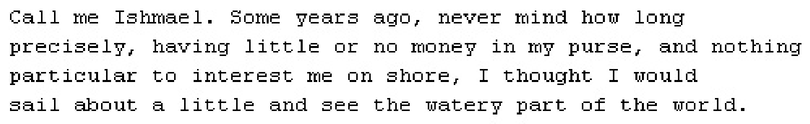
```
Slack-Table($W, L$)
 1  $n \leftarrow length[W]$
 2  $S \leftarrow \emptyset,\ K \leftarrow \emptyset$
 3  $\triangleright$ Initialize slack table $S$
 4  $S[1,1] \leftarrow W[1]$
 5  $K[1] \leftarrow n$
 6  for $k \leftarrow 2$ to $n$
 7      do $S[1,k] \leftarrow S[1,k-1] + W[k]$
 8  for $i \leftarrow 2$ to $n$
 9      do $k \leftarrow i$
10         $S[i,k] \leftarrow S[1,k] - S[1,i-1]$
11         while $k \le n-1$ and $L - S[i,k] - (k-i) > 0$
12             do $k \leftarrow k+1$
13                $S[i,k] \leftarrow S[1,k] - S[1,i-1]$
14         $K[i] \leftarrow k$
15  $\triangleright$ Evaluate sum-of-squares for valid entries
16  for $i \leftarrow 1$ to $n$
17      do for $k \leftarrow i$ to $K[i]$
18             do $S[i,k] \leftarrow L - S[i,k] - (k-i)$
19                if $S[i,k] < 0$
20                  then $S[i,k] \leftarrow \infty$
21                  else $S[i,k] \leftarrow S[i,k]^2$
22  return $S$
```

On lines 1 and 2 of Slack-Table, we initialize the S and K arrays, where S (eventually) stores the slack costs and K stores the last allowed word on a line. (Note that on lines 2-15, the value of $S[i,k]$ actually corresponds to $N[i,k] - (k-i)$ as defined in Equation 1.) On lines 5 and 6 we



```
Call me Ishmael.
Some years ago,
never mind how
long precisely,
having little
or no money
in my purse,
and nothing
particular to
interest me
on shore, I
thought I would
sail about a
little and see
the watery part
of the world.
```

(a) $L = 16$



```
Call me Ishmael. Some years ago, never mind how long
precisely, having little or no money in my purse, and nothing
particular to interest me on shore, I thought I would
sail about a little and see the watery part of the world.
```

(b) $L = 64$

Figure 2: Examples of neatly-printed text produced by PRINT-NEATLY using the example shown in Figure 1 with various values of the maximum line length L.

recursively compute the initial values of $N[1,i]-(i-1)$ for $i \in \{1,\ldots,n\}$. We exploit the following property on lines 8-15 to efficiently compute the slack costs from the initial row of the S matrix.

$$N[i,k] = N[1,k] - N[1,i-1] - 1, \text{ for } i > 1 \tag{3}$$

On lines 17-22 we compute the actual slack costs of the finite elements of S in-place using Equation 2. As shown in Figure 1(b), the output of SLACK-TABLE is a square matrix S requiring $O(n^2)$ storage. As will be discussed in detail in Section 3, the running time of SLACK-TABLE is $O(nL)$.

2 A Dynamic Program for Printing Neatly

2.1 High-level Description

The key to developing a dynamic program is in identifying the appropriate recursion to solve the problem in a "bottom-up" manner. For the "pretty printing" task we identify the following key fact: once a minimum cost partition of words to a set of lines has been discovered, the inclusion of additional words can only alter the configuration of the last line. This is best demonstrated by an example; consider the excerpt shown in Figure 1(a). In this case, the optimum partition required six lines. If we were to append another sequence of words to this paragraph, the arrangement of the first *five* lines would not be altered.

We can express this observation as a recurrence relation on the minimum cost $C[i]$ of arranging the first i words in the text.

$$C[i] \equiv \begin{cases} 0 & i = 0 \\ \min\limits_{1 \le k \le i} C[k-1] + S[k,i] & i \ge 1 \end{cases} \tag{4}$$

In this expression, k corresponds to the index of the first word on the last line. As a result, Equation 4 provides the *arrangement cost* $C[i]$ of placing the first i words, with the first word on the last line given by the argument k which achieved the minimum cost arrangement for a given i words. In general, we can use a simple dynamic program to recursively compute the minimum costs $\{C[i]\}\ \forall i$ in a "bottum-up" manner. The pseudocode for such a program is shown below. Note that the inputs are identical to SLACK-TIME: the word length array W and the maximum line width L. The output of PRINT-NEATLY is a list F of the indices of the first word on each line.

```
PRINT-NEATLY($W, L$)
 1  $n \leftarrow length[W]$
 2  $S \leftarrow$ SLACK-TABLE($W, L$)
 3  $C \leftarrow \emptyset,\ B \leftarrow \emptyset$
 4  $\triangleright$ Determine the least-cost arrangement
 5  $C[1] \leftarrow 0$
 6  for $i \leftarrow 1$ to $n$
 7      do $C[i+1] \leftarrow \infty$
 8         $k \leftarrow i$
 9         $T \leftarrow C[k] + S[k, i]$
10         if $T < C[i+1]$
11            then $C[i+1] \leftarrow T$
12                 $B[i] \leftarrow k$
13         while $k \geq 2$ and $T < \infty$
14             do $k \leftarrow k - 1$
15                $T \leftarrow C[k] + S[k, i]$
16                if $T < C[i+1]$
17                   then $C[i+1] \leftarrow T$
18                        $B[i] \leftarrow k$
19  $\triangleright$ Determine the first word on each line
20  $F \leftarrow \emptyset,\ F \leftarrow B[n] \cup F$
21  $i \leftarrow B[n] - 1$
22  while $i > 0$
23      do $F \leftarrow B[i] \cup F$
24         $i \leftarrow B[i] - i$
25  return $F$
```

On line 2 we call SLACK-TABLE and pre-compute all possible slack values. On line 3 we initialize the cost array C and an array B to store the "breaking" word k – corresponding to the first word on the last line for the best arrangement of words 1 through i. Lines 6-18 compute the values of $C[i]$ defined by Equation 4. Note that, in general, we do not have to test every possible "breaking" word k. That is, there is some value of k for which the last line exceeds the maximum line width L. Once we find that a certain word cannot fit on the last line, then there is no reason to try placing additional words on this line. (We'll see in the next section that this observation will ultimately improve the running time).

To determine the actual arrangement of words, we can use the "breaking" word array B. As implemented on lines 19-24, we initially add $B[n]$ to the output list. That is, the last line of the best arrangement must contain the "breaking" word $B[n]$ for the least cost arrangement of all n words. Next, we find the "breaking" word for the best cost arrangement of words 1 through $B[n] - 1$. We iterate this process until we find the first word in the paragraph.

2.2 Proof of Correctness

The correctness of SLACK-TABLE is immediate from Equation 3. The proof of correctness for PRINT-NEATLY follows directly from our reasoning for Equation 4. That is, if the least cost arrangement of the first i words currently requires l lines, then we cannot rearrange the position of words in lines 1 through $l-1$ and achieve a lower cost arrangement $C[i-1]$. If we could, then we could replace our current solution for $C[i]$ with the rearranged text and achieve a lower cost arrangement of the first i words. Therefore, a proof by contradiction holds, and Equation 4 must be correct.

3 Analysis of Storage Requirements and Running Times

3.1 Storage Requirements

As previously discussed, Slack-Time required $O(n^2)$ storage for the S matrix. This is shown graphically in Figure 1(b). Aside from the matrix S, Print-Neatly requires two additional arrays C and B which are both linear in the size of the input, therefore the overall storage requirements are $O(n^2)$.

3.2 Running Times

The asymptotic worst-case running time for both Slack-Time and Print-Neatly is $O(nL)$, where L is the integer value of the maximum allowed line width. We can demonstrate this by examining lines 6-18 of Print-Neatly. Initially, one might expect the running time to be quadratic in n, the length of W. This is not the case, however, due to the **while** condition on line 13. As previously mentioned, once we find that a certain word cannot fit on the current line, then there is no reason to try placing additional words on this line. As a result, if the slack cost becomes infinite, then no additional values of k can be valid in the final solution.

In the worst case, each word will be composed of a single character. As a result, the **while** loop will be required to evaluate $\lceil L/2 \rceil$ values of k before exiting. Since this value is proportional to L and the **while** loop is inside a **for** loop over n elements, the overall running time is given by $O(nL)$. (Note that a similar argument applies to lines 8-14 of Slack-Table which establishes its running time of $O(nL)$ as well).

4 Conclusion

This write-up has presented the design and analysis of an algorithm for "pretty printing". As previously discussed, this problem has been of historic interest in the design of document preparation systems including TEX and LATEX. As presented, this problem highlights several general characteristics of dynamic programs. Most importantly, we find that the running time of Print-Neatly is "pseudo-polynomial" and depends on the input parameter L [2].

It is important to note that the definition of "even" text is arbitrary. Cormen *et al.* propose a cost function proportional to the cube of the slacks and unrestricted for the last line of a paragraph [1]. In any case, given a definition of "even" text, it would be a straightforward task to modify Equation 2 and Slack-Table to achieve the desired result. In conclusion, the proposed solution possesses some degree of generality and could easily be implemented in a real-world document preparation system.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition.* The MIT Press and McGraw-Hill, 2001.

[2] Jon Kleinberg and Eva Tardos. *Algorithm Design.* Addison-Wesley, 2005.