A Practical Longest Common Subsequence Algorithm for Text Collation

Jim des Rivieres

SCS-TR-4
June 1982

School of Computer Science
Carleton University
Ottawa, K1S 5B6
Canada

A Practical Longest Common Subsequence for Text Collation

Jim des Rivieres
School of Computer Science
Carleton University

Abstract

Previously published algorithms for finding the longest common subsequence in less than quadratic space are not well suited to the task of finding the differences between long streams of words (text collation). This paper presents a practical incremental algorithm with typical space requirements small enough to allow the handling of large insertions and deletions.

Keywords and Phrases: Longest common subsequence, text collation, practical algorithms, prose collation, scholarly editing, word processing

CR Categories: 3.43, 3.63, 3.79, 5.25, 5.39

Introduction

Algorithms for finding the longest common subsequence(LCS) of two sequences have been used in several applications: line by line comparison of files [1], word by word comparison of files [2], and, more recently, updating the image displayed on a video terminal [3]. This paper addresses itself to the problems found in the word by word comparison of files.

In the preparation of definitive editions of literary works, it is necessary to compare all extant editions of a particular work published during the author's lifetime, looking for differences in wording, spelling, and punctuation. Armed with a detailed list of the different variant readings of a work, the scholarly editor may begin to restore the work to something close to what the author intended, undoing those changes introduced during the process of editing and typesetting. This process of word by word comparison, called text collation, is obviously a tedious and time-consuming task. In the late sixties several researchers developed automatic text collation techniques using computers [4,5,6,7]. A text collation with sound theoretical underpinnings was presented in [2] and shown to be theoretically better than existing algorithms. Called OPCOL, this text collation algorithm was based on the longest common subsequence algorithm of [8]. Due to the quadratic space required by the algorithm, OPCOL is limited in the size of insertions and deletions that can be handled in main memory.

There are two things that make the text collation problem different from line by line file difference detection. The first is size - a chapter is the natural unit of work, and it is not uncommon for a chapter to exceed 10,000 words in length. The longest common subsequence file difference program described in [1] is limited to about 3,500 lines. The second, and more important, difference is that most file difference programs make use of the property that the probability of two lines being the same is low [1,9]. This is definitely not the case in text collation. Most file difference programs would perform poorly on files consisting entirely of words, one per line.

The size of the texts also has a major bearing on the type of longest common subsequence algorithm that can be used. For reasons of both time and space, it is impractical to apply any LCS algorithm to the entire texts. Two streams of words can be compared for equality in linear time and fixed space, whereas an LCS determination would take quadratic time and at least linear space, given a sufficiently rich vocabulary [10]. OPCOL is a hybrid algorithm; it performs pairwise matching of words until a difference is detected; then it uses a LCS algorithm to isolate and describe the differences and to find a place where normal pairwise matching can be resumed. The LCS algorithm is used in an incremental, or online, fashion: the two sequences of words, one for each text, are extended until the results of adding successive words indicate that the section of differences has been isolated. Unfortunately, this incremental property makes it difficult to use any of the sophisticated LCS algorithms that run in better than quadratic time and/or space [11,12,13].

In this paper, we present an incremental longest common subsequence algorithm based on [8] that compares two sequences of length n and m in O(n*m) time and O(n+m+r) space, where r is the total number of matching pairs of positions within the sequences in question. Moreover, the constants of proportionality are reasonably small, making the algorithm eminently practical for use in text collation algorithms such as OPCOL.

Terminology and Notation

A = A[1]A[2]...A[n] is a sequence of symbols, or words, of length n. A[i] denotes the ith element of the sequence, 1<=i<=n. A[i:k] denotes the sequence A[i]A[i+1]...A[k] when 1<=i<=k<=n, and the the empty sequence when i>k. C = C[1]C[2]...C[p] is a subsequence of sequence A = A[1]A[2]...A[n] if and only if there is a mapping f:{1,2,...,p}->{1,2,...,n} such that f(k)=i only if C[k]=A[i] and f is strictly increasing. For example, "eat" is a subsequence of "equality" when viewed as character sequences.

Sequence C is a common subsequence of sequences A and B if and only if C is a subsequence of both A and B.

The longest common subsequence problem can be stated as follows: given sequences A = A[1]A[2]...A[n] and B = B[1]B[2]...B[m], find a sequence C = C[1]C[2]...C[p] such that C is a common subsequence of A and B, and p is maximized. We call C a longest common subsequence of A and B. For example, the longest common subsequence of "algol" and "pascal" is "al".

Throughout this paper the variables i and n always apply to the A sequence, and j and m, to the B sequence.

An Incremental LCS Algorithm

Given two sequences A and B, an incremental LCS algorithm computes the LCS for A[1:n] and B[1:m] where n and m are not greater than the true lengths of A and B, respectively. The algorithm shown in Figure 1 is essentially ALG A of [11] (which is based on an Algorithm X of [8]) expressed in an incremental form. The A and B sequences are stored in arrays that are passed as parameters to the LCS module. State is maintained in three variables: LastA and LastB keep track of how many words of the sequences have been tabulated, and L is a matrix containing lengths of least common subsequences. The following invariant is preserved by all of the operations: for all i and j, 0<=i<=LastA, 0<=j<=LastB, L[i,j] = the length of the LCS for A[1:i] and B[1:j]. For now we will assume that L is "big enough" to handle anything we want to do. The InitializeLCS function sets things up. ExtendA (ExtendB) is used to increase the length of the A (B) sequence. All three functions return the length of the longest common subsequence of the portions of A and B tabulated - this information can be used to determine which way to grow the L matrix next, and when to stop.

At any point, it is possible to traverse the L matrix and extract those pairs of positions that are part of a longest common subsequence. In fact, it is possible to extract all LCSs, but this is not important for text collation applications. From this list of pairs it is a simple process to describe how sequences A[1:LastA] and B[1:LastB] differ. The functions shown in Figure 2, which should be included in the preceding LCS module, are based on Algorithm Y in [8]. (cons is the standard LISP list processing primitive.)

To compare the first n words of A with the first m words of B and get a list of the pairs of positions in some LCS, InitializeLCS will have to be called exactly once, followed by n calls to ExtendA mixed with m calls to ExtendB, followed by a single call to GetLCS. Without loss of generality, assume that all calls to ExtendB precede those to ExtendA. The call to InitializeLCS and each call to ExtendB will take a fixed number of steps, independent of n and m. Each of the n calls to ExtendA will require O(m) steps since it will be required to fill in L[i,0:m] for some i. The loop in ExtractLCS can be executed at most n+m times, of which at most min(n,m) steps can add another pair of positions to the list. Thus the total running time will be O(n*m) steps. The portion of the L matrix that is used requires O(n*m) space, and that overshadows the O(min(n,m)) storage requirements of the list of position pairs.

The O(n*m) running time is tolerable; the O(n*m) space is not. Having to limit n and m for practical use in text collation imposes restrictions on the length of insertions and deletions that can be handled automatically - if the limits is, say, 300, it will be impossible to recover from a deletion of more than 300 successive words, which in many books is less than a single printed page. In the next sections, we will look at how to significantly decrease the space requirements by transforming the functions presented in this section.

Extracting LCS Lists on the Fly

The previous collection of functions construct the L matrix incrementally, a row or a column at a time. However, the LCS extraction is done after the fact using the entire L matrix. By transforming the functions so that LCS information is extracted on the fly, it will become possible to later eliminate most of the matrix.

Define a matrix T in the module and add to each of the functions InitializeLCS, ExtendA, and ExtendB the statement T[i,j] := ExtractLCS(i,j) just after each new element of L has been assigned a value. As a result, T[i,j] will contain a list of pairs of positions in an LCS of A[1:i] and B[1:j].

Several simplifying transformations are now possible. Obviously T[i,0] = T[0,j] = nil for all i and j, 0<=i<=LastA, 0<=j<=LastB. From the workings of ExtractLCS, it is evident that T[i,j] can be either T[i-1,j], T[i,j-1], or cons(cons(i,j),T[i-1,j-1]), depending on the relative values of L[i,j], L[i-1,j], and L[i,j-1]. (Actually, T[i,j] = reverse(ExtractLCS(i,j)), but this can be readily resolved in GetLCS.) After having expanded the calls to ExtractLCS inline and having done the aforementioned simplifications, the module appears as shown in Figure 3 (due to its close similarity to ExtendA, ExtendB has been omitted.)

Besides the obvious additional O(n*m) storage required for matrix T (which we will address momentarily), how much extra space is required for the LCS lists? Noting that each time through the loop an Extend function can create at most one new position pair, it is clear that at most n*m are created.

But we can be more precise. The notion of candidate is defined in [11] as follows: {(0,0)} is the set of 0-candidates, and for k>=1, (i,j) is a k-candidate if A[i]=B[j] and there exists i´ and j´ such that i´<i, j´<j, and (i´,j´) is a (k-1)-candidate. The number of conses is precisely the number of potential k-candidates for some k>=1. In the worst case, this is O(n*m), but in practice the number of candidates is quite small.

Getting Rid of the Matrices

What has the on-the-fly LCS extraction bought us? Inspection of the ExtendA function shows that only the last row of the L and T matrices are ever interrogated (and ExtendB uses only the last column). It is therefore possible to replace the L matrix with two vectors, HL[0:LastB] and VL[0:LastA], containing row LastA and column LastB of L, respectively. Similarly, the T matrix can be replaced by vectors HT[0:LastB] and VT[0:LastA]. The result is an O(n*m) time algorithm with O(n+m+r) space, where r is the number of candidates. As mentioned before, r is at worst O(n*m) but is typically much more reasonable.

In the slightly revised versions of ExtendA and ExtendB shown in Figure 4, the current value of L[i-1,j-1] is remembered in CornerL, L[i-1,j] in AboveL, L[i,j-1] in LeftL, and the value of L[i,j] in CurrentL. Similarly for the T values.

Estimated Storage Requirements

Letting k be the limit on the size of the sequences to be compared, the array storage requirements are:

i) A[1:k] and B[1:k], the word arrays.

ii) HL[0:k] and VL[0:k], one row and one column of the L matrix.

iii) HT[0:k] and VT[0:k], one row and one column of the T matrix.

Assume that there is a pool of cons cells from which to draw storage for the list of position pairs. If the pool is used solely for the purposes of the LCS determination, InitializeLCS can create the the free list for the pool each time it is called. Reference counting or garbage collection would help, but are not essential.

We can estimate the size of the pool of cons cells required for the collation of typical English language text as follows. Given a k word sequence of English text, we expect that the number of distinct words present, and the number of occurrences of each to be based on properties of the English language. If we know the probability of occurrence of the most frequently occurring English words, we can determine the expected number of occurrences of each in the k word sample; the remainder of the sample will be accounted for by lower probability words.

By assuming that both sequences being collated will have the same composition, the number of potential candidates can be estimated by summing the squares of the expected number of occurrences over the distinct words expected in the sample. For example, based on English word frequency statistics published in [14], we estimate the composition of a 100 word sample to be: 7 occurrences of "the", 3 occurrences each of "and" and "of", 2 occurrences each of "a", "to", and "in", and a single occurrence of each of 81 other distinct words. If two similarly constituted samples of 100 words were collated, the number of candidates would be at most 7**2 + 2(3**2) + 3(2**2) + 81(1**2) = 160, since this is the expected number of entries in the L matrix with identical row and column words (and the candidates are a subset of these). Table I gives some estimates for several values of k.

Number of words in sequence (k)	Expected number of distinct words	Estimated number of potential candidates	Contribution of the word "the"
100	87	160	49
200	160	524	225
500	352	2726	1369
1000	634	10108	5329
2000	1148	39618	21316
5000	2722	244376	133225
10000	5348	971860	534682

Table I.

As can be seen from Table I, the word "the" alone, with probability of occurrence of 0.0731228, can be expected to account for around half of the potential candidates. This fact can be used to arrives at a simple estimate of the number of candidates: 2*(k*p("the"))**2 ~ 0.01*k**2. Thus, only about 1% of the entries in the original table will be potential candidates when collating English text.

In practical terms, is it possible to use large values for k. Given that the average word can be represented in 3 machine words of a typical large main-frame, that a cons cell takes 2 machine words, and that each entry of the HL, VL, HT, and VT arrays need one word, the total storage is roughly 10*k+0.04*k**2 machine words. However, the algorithm still has O(k*k) running time, and this will undoubtably prove to be the overriding consideration.

In the context of text collation, this means that insertions and deletions in excess of 1,000 words in length can be handled with all tables in main memory. This is a great improvement over the original LCS algorithm used in OPCOL, whose practical space requirements would be around 6*k+k**2 machine words (k=545 would occupy roughly 300,000 machine words).

Further space and time savings can be realized by replacing the HL and VL vectors, which contain numbers in the range 0 to k, with Boolean vectors that encode only the change in L value while traversing the row or column. Although the space saved is small, the inner loop of the Extend functions can be made faster since most of the work is in checking for increasing LCS length along rows and columns.

Conclusions

The techniques described in this paper are being used successfully in the OPCOL-based text collation program being used by the Centre for Editing Early Canadian Texts, which is funded by grants from the Social Science and Humanities Research Council and Carleton University. Early experience shows that about half of the program's time is spent in reading lines of text and breaking them into words.

References

[1] Hunt, J.W. and McIlroy, M.D. "An Algorithm for Differential File Comparison". Computer Science Technical Report #41, Bell Laboratories, Murray Hill, N.J., August 1976.

[2] Cannon, R.L.Jr. "OPCOL: An Optimal Text Collation Algorithm". C.Hum. 10 (1976), 33-40.

[3] Gosling, J. "A Redisplay Algorithm". Proc. ACM SIGPLAN/SIGOA Conf. on Text Manipulation, SIGPLAN Notices 16, 6 (June 1981), 123-129.

[4] Froger, J. La Critique des textes et son automatisation. Dunod, Paris, 1970.

[5] Cabaniss, M.S. "Using the Computer for Text Collation". Computer Studies in the Humanities and Verbal Behaviour 3 (1970), 1-33.

[6] Petty, G.R. and Gibson, W.M. Project OCCULT. New York University Press, New York, 1970.

[7] Gilbert, P. "Automatic Collation: A Technique for Medieval Texts". C. Hum. 7 (1973), 139-147.

[8] Wagner, R.A. and Fischer, M.J. "The String-to-String Correction Problem". J. ACM 21, 1 (January 1974), 168-173.

[9] Heckel, P. "A Technique for Isolating Differences Between Files". Comm. ACM 21, 4 (April 1978), 264-268.

[10] Aho, A.V., Hirschberg, D.S., and Ullman, J.D. "Bounds on the Complexity of the Longest Common Subsequence Problem". J. ACM 23, 1 (January 1976), 1-12.

[11] Hirschberg, D.S. "A Linear Space Algorithm for Computing Maximal Common Subsequences". Comm. ACM 18, 6 (June 1975), 341-343.

[12] Hunt, J.W. and Szymanski, T.G. "A Fast Algorithm for Computing Longest Common Subsequences". Comm. ACM 20, 5 (May 1977), 350-353.

[13] Hirschberg, D.S. "Algorithms for the Longest Common Subsequence Problem". J. ACM 24, 4 (October 1977), 664-675.

[14] Carroll, J.B., Davies, P., and Richman, B. The American Heritage Word Frequency Book. Houghton Mifflin Company, Boston, 1971.

```
module LCS(A,B[0:Size] of word)
   L[0:Size,0:Size] of 0:Size
   LastA, LastB: 0:Size

   function InitializeLCS returns 0:Size
      LastA := 0      LastB := 0
      L[LastA,LastB] := 0
      return L[LastA,LastB]
   endfunction

   function ExtendA returns 0:Size
      i, j: 0:Size

      LastA := LastA + 1
      i := LastA
      L[i,0] := 0
      for j := 1 to LastB do
         if A[i] = B[j] then
            L[i,j] := L[i-1,j-1] + 1
         else
            L[i,j] := maximum(L[i-1,j],L[i,j-1])
         endif
      endfor
      return L[LastA,LastB]
   endfunction

   function ExtendB returns 0:Size
      i, j: 0:Size

      LastB := LastB + 1
      j := LastB
      L[0,j] := 0
      for i := 1 to LastA do
         if A[i] = B[j] then
            L[i,j] := L[i-1,j-1] + 1
         else
            L[i,j] := maximum(L[i-1,j],L[i,j-1])
         endif
      endfor
      return L[LastA,LastB]
   endfunction
endmodule
```

Figure 1. Incremental LCS algorithm.

```
internal function ExtractLCS(APosition,BPosition: 0:Size) returns list
   i, j: 0:Size
   LCSList: list

   i := APosition
   j := BPosition
   LCSList := nil
   while i>0 and j>0 do
      if L[i,j] = L[i-1,j] then
         i := i - 1
      else if L[i,j] = L[i,j-1] then
         j := j - 1
      else
         LCSList := cons(cons(i,j),LCSList)
         i := i - 1
         j := j - 1
      endif
   endwhile
   return LCSList
endfunction

function GetLCS returns list
   {Returns a sorted list of pairs of positions in some LCS
    of A[1:LastA] and B[1:LastB].}
   return ExtractLCSList(LastA,LastB)
endfunction
```

Figure 2. Additional functions for extracting LCS position pairs.

```
module LCS(A,B[0:Size] of word) {With the T matrix.}
   L[0:Size,0:Size] of 0:Size
   T[0:Size,0:Size] of list
   LastA, LastB: 0:Size

   function InitializeLCS returns 0:Size
      i, j: 0:Size

      LastA := 0      LastB := 0
      L[LastA,LastB] := 0
      T[LastA,LastB] := nil
      return L[LastA,LastB]
   endfunction

   function ExtendA returns 0:Size
      i, j: 0:Size

      LastA := LastA + 1
      i := LastA
      L[i,0] := 0
      T[i,0] := nil
      for j := 1 to LastB do
         if A[i] = B[j] then
            L[i,j] := L[i-1,j-1] + 1
         else
            L[i,j] := maximum(L[i-1,j],L[i,j-1])
         endif
         if L[i,j] = L[i-1,j] then
            T[i,j] := T[i-1,j]
         else if L[i,j] = L[i,j-1] then
            T[i,j] := T[i,j-1]
         else
            T[i,j] := cons(cons(i,j),T[i-1,j-1])
         endif
      endfor
      return L[LastA,LastB]
   endfunction

   function GetLCS returns list
      return reverse(T[LastA,LastB])
   endfunction

endmodule
```

Figure 3. Incremental LCS algorithm with position pairs extracted on the fly.

```
module LCS(A,B) {With only one row and column stored.}
   HL[0:Size],VL[0:Size] of 0:Size
   HT[0:Size],VT[0:Size] of list
   LastA, LastB: 0:Size

   function InitializeLCS returns 0:Size
      LastA := 0             LastB := 0
      HL[0] := 0             VL[0] := 0
      HT[LastB] := nil       VT[LastA] := nil
      return HL[LastB]
   endfunction

   function ExtendA returns 0:Size
      i, j, AboveL, LeftL, CurrentL, CornerL: 0:Size
      AboveT, LeftT, CurrentT, CornerT: list

      LastA := LastA + 1
      i := LastA
      {HL[0] is always 0; HT[0] is always nil.}
      AboveL := 0            CurrentL := 0
      AboveT := nil          CurrentT := nil
      for j := 1 to LastB do
         LeftL := CurrentL  CornerL := AboveL  AboveL := HL[j]
         LeftT := CurrentT  CornerT := AboveT  AboveT := HT[j]
         if A[i] = B[j] then
            CurrentL := CornerL + 1
         else
            CurrentL := maximum(AboveL,LeftL)
         endif
         if CurrentL = AboveL then
            CurrentT := AboveT
         else if CurrentL = LeftL then
            CurrentT := LeftT
         else
            CurrentT := cons(cons(i,j),CornerT)
         endif
         HL[j] := CurrentL
         HT[j] := CurrentT
      endfor
      VL[LastA] := HL[LastB]
      VT[LastA] := HT[LastB]
      return HL[LastB]
   endfunction
```

Figure 4(a). Final incremental LCS alogrithm.

```
   function ExtendB returns 0:Size
      i, j, AboveL, LeftL, CurrentL, CornerL: 0:Size
      AboveT, LeftT, CurrentT, CornerT: list

      LastB := LastB + 1
      j := LastB
      {VL[0] is always 0; VT[0] is always nil.}
      LeftL := 0           CurrentL := 0
      LeftT := nil         CurrentT := nil
      for i := 1 to LastA do
         AboveL := CurrentL  CornerL := LeftL  LeftL := VL[i]
         AboveT := CurrentT  CornerT := LeftT  LeftT := VT[i]
         if A[i] = B[j] then
            CurrentL := CornerL + 1
         else
            CurrentL := maximum(AboveL,LeftL)
         endif
         if CurrentL = AboveL then
            CurrentT := AboveT
         else if CurrentL = LeftL then
            CurrentT := LeftT
         else
            CurrentT := cons(cons(i,j),CornerT)
         endif
         VL[i] := CurrentL
         VT[i] := CurrentT
      endfor
      HL[LastB] := VL[LastA]
      HT[LastB] := VT[LastA]
      return HL[LastB]
   endfunction

   function GetLCS returns list
      return reverse(HT[LastB])
   endfunction
endmodule
```

Figure 4(b). Final incremental LCS algorithm.

Carleton University
School of Computer Science

Bibliography of SCS reports

SCS-TR-1 THE DESIGN OF CP-6 PASCAL
Jim des Rivieres and Wilf R. Lalonde, June 1982.

SCS-TR-2 SINGLE PRODUCTION ELIMINATION IN LR(1) PARSERS: A SYNTHESIS
Wilf R. Lalonde, June 1982.

SCS-TR-3 A FLEXIBLE COMPILER STRUCTURE THAT ALLOWS DYNAMIC PHASE ORDERING
Wilf R. Lalonde and Jim des Rivieres, June 1982.

SCS-TR-4 A PRACTICAL LONGEST COMMON SUBSEQUENCE ALGORITHM FOR TEXT COLLATION
Jim des Rivieres, June 1982.