Department of Mathematics and Computer Science January 12, 2017
University of Southern Denmark RF

The Hunt-Szymanski Algorithm for LCS

In 1977, James W. Hunt and Thomas G. Szymanski published an algorithm for the Longest Common Subsequence (LCS) problem which runs in time $O(r \log n + m \log(m))$, where m is the length of the longest string X, n is the length of the shortest string Y, and r is the number of pairs (i, j) for which $X[i] = Y[j]$. For the strings `acea` and `aeaca`, the set R of such pairs can be illustrated as follows:

	a	e	a	c	a
a	×		×		×
c				×	
e		×			
a	×		×		×

Since r is at most mn, this is never much worse than the standard dynamic programming algorithm, which runs in $O(mn)$ time. On the other hand, r may be much smaller, in particular for large alphabets and fairly random strings—for strings with characters chosen uniformly at random from an alphabet Σ, the expected value of r is $mn/|\Sigma|$.

A version of the algorithm is used in the `diff` command (for comparing files on UNIX/Linux and in many version control systems) when comparing two text-based files: the idea is to hash each line in a file to a k bit integer, and view the file as a string of such integers. To come up with a suggestion for which lines have *not* been changed between two files, the LCP of the corresponding two strings is calculated (the rest of the lines are reported as removed in one file or inserted in the other). For files with few repeated lines (and assuming a good hash function), these strings will behave like random strings from an alphabet of size 2^k, hence the Hunt-Szymanski algorithm will be very efficient.

The algorithm is based on the following observation:

Lemma 1 *A row in the* lcs*-table for the standard dynamic programming algorithm for the LCS problem has a content of the following type:*

$$0\,0\,0\,0\,0\,1\,1\,1\,1\,2\,3\,3\,3\,3\,4\,4\,4\,4$$

In other words, it is a sequence of non-empty blocks of integers of the same value, where the value is zero for the first block and increases by one for each new block.

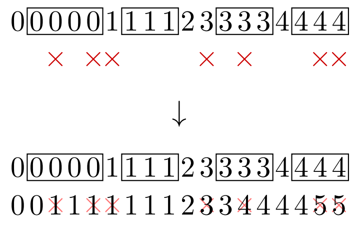
Proof: By induction on the row number. The base case is the first row, which contains only zeros (by the base case of the recursive formula for the dynamic programming algorithm), i.e., is a line of the above type containing just a single block. For the induction step, assume that the statement is true for row $i-1$. For a block starting at entry $(i-1, j_1)$ and ending at entry $(i-1, j_2)$, call j_1 its *head* index and $j_1+1, j_1+2, \ldots, j_2$ its *tail* indices. Below, the row from above is shown again with the (row entries of the) tail indices of the blocks framed:

$$0\boxed{0\,0\,0\,0}1\boxed{1\,1\,1}2\,3\boxed{3\,3\,3}4\boxed{4\,4\,4}$$

We call an index j in row $i-1$ *active* if (i, j) is in R. Consider filling out row i from left to right using the recursive formula for the dynamic programming algorithm. The first entry $\text{lcs}(i, 0)$ is always zero. When filling out using the recursive formula, the value of $\text{lcs}(i, j)$ for increasing j's will stay at zero until the first active tail index j in the first block in row $i-1$, if such an index exists. At this point $\text{lcs}(i, j)$ will turn one. For the remaining tail indices j in the first block of row $i-1$, $\text{lcs}(i, j)$ will stay at one, whether or not the indices are active. When j becomes the head index of the second block of row $i-1$, $\text{lcs}(i, j)$ will become one in all cases (whether or not an active index was met in the tail of the first block, and whether of not the head index of the second block is active).

The process thus repeats with the second block, just with all lcs values increased by one. By induction on the block number (an induction separate from the main induction), the process repeats for all the blocks in the row.

The entire step when generating row i from row $i-1$ is illustrated below. Active indices in row i are shown by red crosses.



This proves the induction step (of the main induction). $\square$

The next observation used in the algorithm is that by Lemma 1, a row can be represented simply by storing the head indices of its blocks. The row

$$0\,0\,0\,0\,0\,1\,1\,1\,1\,2\,3\,3\,3\,3\,4\,4\,4\,4$$

will thus be represented by the set $\{0, 5, 9, 10, 14\}$ (assuming that the first character of a string has index one, hence the first column index zero).

Recall from DM507 that for an ordered set S, the *predecessor* of a search key x is defined by $\text{pred}(x) = \max\{y \in S \mid y \leq x\}$ and the *successor* is defined by $\text{succ}(x) = \min\{y \in S \mid y \geq x\}$. Also recall from DM507 that storing the set in a decorated balanced binary tree allows us to perform each of these operations in time $O(\log |S|)$.

The Hunt-Szymanski algorithm generates the rows of the lcs-tabel in a top-down manner, using this representation. From the proof of the induction step in Lemma 1, we see how row i can be produced from row $i-1$: Each head index should be moved to the leftmost active tail index (if any) in the block to the left of the head index, and the leftmost active tail index (if any) in the rightmost block should generate a new head index.

Assume that we for all i have access to the i'th row $R[i] = \{j \mid (i, j) \in R\}$ of the active indices R, sorted in descending order on j. Then the following algorithm carries out the method above:

```
T.insert(0) // Initialize tree to represent the first row
FOR i=1 to m
  FOR j in R[i] // in decreasing order of j
    IF T.succ(j) exists
      T.delete(T.succ(j))
    T.insert(j)
```

The inner loop correctly generates row i from row $i-1$: Actions are only taken on active indices. If the first j from $R[i]$ is the rightmost active tail index of the rightmost block, the second to last line will not be executed for this j, and the last line will insert a new head. For the rest of the j's in $R[i]$, both of the two last lines will be executed. If j is a head index, the two lines cancel out (recall that the successor of an element present is the element itself, so the head index is deleted and then re-inserted in the tree). If j is a tail index, the lines move the closest head index right of j to position j. Due to the decreasing order of $R[i]$, the total effect is that all existing head indices are moved to the leftmost active tail index (if any) in the block to the left of the head index, and the leftmost active tail index (if any) in the rightmost block generates a new head index. The time used is $O(m + r \log n)$.

What remains to discuss is how to first find R and $R[i]$. For each string X and Y, we annotate its characters with their position, and then sort the characters of each string. This takes $O((m + n) \log(m + n))$ for general comparison-based alphabets. For alphabets linear (polynomial) in $m+n$, it can even be done in time $O(m + n)$ by countingsort (radixsort).

Passing through the two sorted sequences in a merge-like fashion, we can easily generate R in time $r = |R|$: all `a`'s of X and all `a`'s of Y will be passed at the same time and the crossproduct of their positions can be generated, the same applies to the rest of the characters present. Finally, we sort R lexicographically, ascending on first coordinate i and descending on second coordinate j. This can be done by radixsort in $O(r + (m + n))$ time.

In total, this is time $O(r \log n + m \log(m))$ for comparison based alphabets, assuming without loss of generality that $m \geq n$.

The above algorithm at the end has calculated the (represention of) the last row of the lcs-table, hence we can find the value $\text{lcs}(m, n)$. If we desire an actual longest subsequence (which clearly is needed in the `diff` application, for instance), we can try to use backtracking as in the standard dynamic programming algorithm. However, if we generate the entire table during the Hunt-Szymanski algorithm, we will spend additional time $\Theta(mn)$ just on that. Luckily, it turns out that we can restrict ourselves to only store information in the table each time a head index is moved, while still being able to backtrack correctly. The additional time for generating (and backtracking over) this information is $O(r)$.

The details of this are left as an exercise. We here give an outline (not curriculum): The observation used is that during backtracking in the lcs-table, we can always move left or up as long as the entry value stays the same: if e.g. $\mathrm{lcs}(i, j-1) = \mathrm{lcs}(i, j)$, the backtracking from $\mathrm{lcs}(i, j-1)$ will create a LCS for $X[1..i]$ and $Y[1..(j-1)]$ (which is also a LCS for $X[1..i]$ and $Y[1..j]$) of optimal length for $X[1..i]$ and $Y[1..j]$. This means that during backtracking over the table, we can always proceed left to the head index of the block we are in. It also means that from the head index of a block, we can proceed up as long the head index of the block we are in is the same. We exploit this by representing by a single node all consecutive (in terms of row number) versions of a block for which the head index does not change (these blocks are below each other in the table, and combined constitute a connected region of it, with a left side which is a straight, vertical line). The node stores the position (i, j) of the head index of the first of these versions (where the head index had just changed from the previous row). During the algorithm, the currently active nodes (the nodes representing the blocks of row $i-1$ during execution of the inner FOR-loop) are stored with the blocks (i.e., stored with the head indices kept in the tree). When a head index of a block changes from row $i-1$ to row i, it is due to an active index, and the correct backtracking from this position in the table is diagonally left-up. This is to a block which is among those represented by the currently active node v representing the block to the left (block entry value one lower). A new node u for the moved block is created (only one node for the entire move of the head index, even if it is done over several iterations of the inner FOR-loop), and an edge (u, v) is also created. The node u will be the active node for that block in the next iteration of the outer FOR-loop, at which point the node v becomes inactive. If a head index does not change, the active node of the block stays the same for the next iteration of the outer FOR-loop. Backtracking is now performed as follows: Starting with the node representing the region containing $\mathrm{lcs}(m, n)$, we move to its stored coordinates (i, j) (in terms of backtracking in the entire table, this is a move left, then up, while generating no elements of the output LCS). From there, we follow the single edge out of the node to the node representing the region containing $\mathrm{lcs}(i-1, j-1)$ (in terms of backtracking in the entire table, this is a move left-up to $(i-1, j-j)$, and should generate one character of the output LCS). Then repeat.