Heaviest Increasing/Common Subsequence Problems.

Conference Paper · April 1992

DOI: 10.1007/3-540-56024-6_5 · Source: DBLP

2 authors, including:

Guy Jacobson
AT&T
20 PUBLICATIONS **692** CITATIONS

SEE PROFILE

Heaviest Increasing/Common Subsequence Problems

Guy Jacobson and Kiem-Phong Vo

AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974

Abstract. In this paper, we define the *heaviest increasing subsequence* (HIS) and *heaviest common subsequence* (HCS) problems as natural generalizations of the well-studied *longest increasing subsequence* (LIS) and *longest common subsequence* (LCS) problems. We show how the famous Robinson-Schensted correspondence between permutations and pairs of Young tableaux can be extended to compute heaviest increasing subsequences. Then, we point out a simple weight-preserving correspondence between the HIS and HCS problems. ¿From this duality between the two problems, the Hunt-Szymanski LCS algorithm can be seen as a special case of the Robinson-Schensted algorithm. Our HIS algorithm immediately gives rise to a Hunt-Szymanski type of algorithm for HCS with the same time complexity. When weights are position-independent, we can exploit the structure inherent in the HIS-HCS correspondence to further refine the algorithm. This gives rise to a specialized HCS algorithm of the same type as the Apostolico-Guerra LCS algorithm.

1 Introduction

Given a sequence σ over some linearly ordered alphabet, the *longest increasing subsequence* (LIS) problem is to find a longest subsequence of σ that is strictly increasing. Given two sequences α and β over some general alphabet, the *longest common subsequence* (LCS) problem is to find a longest sequence γ that is a subsequence of both α and β.

Both the LIS and LCS problems have venerable histories. The LIS problem arises in the study of permutations, Young tableaux and plane partitions. These objects play central roles in the representation theory of the symmetric group initiated by Young[25] and MacMahon in the early part of the century. In 1938, Robinson[16] found an explicit correspondence between permutations and pairs of Young tableaux. This correspondence was rediscovered in 1961 by Schensted[18] who extended it to general integer sequences. There are many interesting results concerning the Robinson-Schensted correspondence. The reader is referred to other papers[20, 5, 22, 12] for more details. Schensted's main motivation was to compute an LIS from a given sequence of integers. This can be done by specializing the algorithm to compute only the left-most column of the Young tableau. Fredman[4] has shown that $O(n \log n)$ time is required to compute an LIS. Thus, the Robinson-Schensted algorithm is optimal for LIS. In a different guise, LIS can be used to compute a largest stable set for a permutation graph[6]. In this guise, LIS has a ready generalization: computing a heaviest stable set for a permutation graph whose nodes have

non-uniform weights. This is a special case of the *heaviest increasing subsequence* (HIS) problem: Given a sequence over some linearly ordered alphabet and a weight function on the symbols and their positions in the sequence, find a subsequence with the heaviest sum of weights. We shall show how to generalize the Robinson-Schensted algorithm to compute an HIS in $O(n \log n)$ time.

The LCS problem was first studied in the context of the *string-to-string correction* problem. Wagner and Fischer[23] solved this problem using dynamic programming in quadratic time and space. Since then, LCS has found many practical applications: CRT screen updates, file differential comparison, data compression, spelling correction, and genetic sequencing[17]. It was the CRT screen update problem that led one of us (Vo) in 1983 to look into a weighted extension for LCS called the *minimal distance LCS* (MDLCS) problem. Here, a typical screen update involves two screens, the current one, and the desired one. The update algorithm must match the screen lines and issue hardware line insertion/deletion to align matched lines. To reduce screen disturbance, it is desirable that matched lines that are closely aligned be given preference over other matches. Thus, the minimal distance weight function assigns higher weights to closely aligned matched lines. The MDLCS problem is to find among all LCS's one that minimizes the total distances of all matched lines. Therefore, the MDLCS weight function combines the length of the common subsequence and the distance between their matches. This is an example of a class of general weight functions that assign values from some *ordered additive monoid* to common subsequences based on both matched symbols and their positions in the original sequences. The *heaviest common subsequence* (HCS) problem is to find common subsequences that maximize such weights. The dynamic programming algorithm is easily extended to solve HCS. This algorithm was implemented in the `curses` screen update library distributed with System V UNIX systems[21].

Aho, Hirschberg and Ullman[1] showed that quadratic time is needed to find the length of an LCS when the computation model allows only equal-unequal comparisons. On the other hand, in a more general computation model, Fredman's lower bound for the LIS problem gives a lower bound of $O(n \log n)$ for computing the LCS of two sequences of length n. This can be seen as follows. Let π be a permutation of the integers from 1 to n. A LCS between π and the sequence $1, 2, \ldots, n$ is also an LIS of π. The first general subquadratic algorithm for LCS was found by Masek and Patterson[13], who employed a "Four-Russians" approach to solve the problem in $O(n^2/\log n)$ for finite alphabets and $O(n^2 \log\log n/\log n)$ for a general alphabet. The question of whether $O(n \log n)$ is a tight bound for the LCS problem is still open.

More recent work on the LCS problem focused on finding general algorithms whose efficiency is a function of certain characteristics of the problem instance.

Hunt and Szymanski[10, 11] gave an $O((r+n)\log n)$ algorithm where r is the total number of *matches*; a match is an ordered pair of positions (i, j) such that $\alpha_i = \beta_j$. This is efficient when the matches are sparse. Apostolico and Guerra[2] improved on the Hunt-Szymanski algorithm and described an $O(d \log n)$ algorithm where d is the number of *dominant matches*; a dominant match is an ordered pair (i, j) such that $\alpha_i = \beta_j$ and every LCS of the prefixes $\alpha_1 \cdots \alpha_i$ and $\beta_1 \cdots \beta_j$ has α_i as its final symbol. The quantity d is important because the number of dominant matches can be much smaller than the number of matches.

Another line of research into LCS seeks an algorithm that is fast when α and β are similar. This is practical, for example, if they are two versions of a text file. Myers[14] describes an $O(n\Delta)$ time algorithm, where Δ is the edit distance between the two strings (with unit cost insertion and deletion). Interested readers should see the work of Hirschberg[8], Nakatsu et al.[15], Hsu and Du[9], Wu et al,[24], and Chin and Poon[3] for other approaches.

The LIS and LCS problems are closely related. In fact, both the Hunt-Szymanski and Apostolico-Guerra algorithms are implicitly based on a correspondence between increasing and common subsequences. We shall state explicitly this bijective correspondence and note that it is weight-preserving. This allows us to map common subsequence problems to increasing subsequence problems and make use of the machinery in the Robinson-Schensted correspondence. It can be seen from this that the Hunt-Szymanski LCS algorithm is only a special case of the Robinson-Schensted algorithm. Then, it is easy to see how the inherent structure in the LCS problem can be exploited in the mapping to tune the Hunt-Szymanski algorithm. This refinement results in the Apostolico-Guerra LCS algorithm. In fact, this process of discovery uncovers a bug in the original description given by Apostolico and Guerra. Finally, applying our extension of the Robinson-Schensted algorithm for computing HIS, we derive fast algorithms for computing HCS.

2 Basic Definitions and Notations

To make the paper self-contained, in this section, we give all the basic definitions and specify conventions for how the algorithms will be presented.

2.1 Sequences and Subsequences

Let $\sigma = \sigma_1\sigma_2\cdots\sigma_p$ be a sequence over some alphabet A. We shall use σ_i to denote the ith symbol of σ, and $\sigma_{i\ldots j}$ to denote the contiguous subsequence consisting of symbols in positions from i to j. A sequence τ is called a subsequence of σ if there is a sequence of integers $i_1 < i_2 < \cdots < i_l$ such that τ is equal to $\sigma_{i_1}\sigma_{i_2}\cdots\sigma_{i_l}$. If the alphabet A is linearly ordered, we say that τ is an increasing subsequence if $\tau_1 < \tau_2 < \cdots < \tau_l$. Given two sequences α and β, a sequence γ is called a common subsequence of α and β if it is both a subsequence of α and a subsequence of β. That is, there are two sequences of integers $i_1 < i_2 < \cdots < i_l$ and $j_1 < j_2 < \cdots < j_l$ such that γ is equal to $\alpha_{i_1}\alpha_{i_2}\cdots\alpha_{i_l}$ and $\beta_{j_1}\beta_{j_2}\cdots\beta_{j_l}$.

2.2 Dominant Matches vs. Edit Distance

A number of modern LCS algorithms have their complexity based on either the edit distance or the number of dominant matches. The edit distance between the strings α and β is the minimum number of character insertions and deletions required to transform α to β. Let the strings α and β have a LCS of length ρ and an edit distance of Δ. We will always have $|\alpha| + |\beta| = 2\rho + \Delta$, because each symbol in α or β but not in the LCS increases the edit distance by one.

Now, define a match (i, j) of the two letters $\alpha_i = \beta_j$ to be dominant if every LCS of $\alpha_{1...i}$ and $\beta_{1...j}$ must end at positions i and j. Following standard conventions, we denote the total number of matches by r and the number of dominant matches by d. Other authors[9, 2] have observed that d can be *much* smaller than r, especially when the two strings are very similar. We will now make this observation rigorous by proving a bound on d based on the edit distance Δ.

Let $\rho(i, j)$ denote the length of the LCS of prefixes $\alpha_{1...i}$ and $\beta_{1...j}$; similarly let $\Delta(i, j)$ denote the edit distance between the prefixes $\alpha_{1...i}$ and $\beta_{1...j}$. Say that a dominant match (i, j) is k-dominant if $\rho(i, j) = k$.

Theorem 1. *The number of dominant matches* $d \leq \rho(\Delta + 1)$.

Proof. Suppose there are d_k k-dominant matches. Sort them by increasing values of i: $\{(i_1, j_1), (i_2, j_2), \ldots, (i_{d_k}, j_{d_k})\}$ where $i_1 < i_2 < \ldots < i_{d_k}$ and $j_1 > j_2 > \cdots > j_{d_k}$. Now because the i's are strictly increasing integers and the j's are strictly decreasing, $i_l - i_1 \geq l - 1$ and $j_l - j_{d_k} \geq d_k - l$.

Now consider the edit distance $\Delta(i_l, j_l)$:

$$\Delta(i_l, j_l) = i_l + j_l - 2\rho(i_l, j_l) = i_l + j_l - 2k.$$

Because (i_1, j_1) is a k-dominant match, $i_1 \geq k$ and similarly, $j_{d_k} \geq k$. So

$$\Delta(i_l, j_l) \geq i_l + j_l - i_1 - j_{d_k}$$

rearranging:

$$\Delta(i_l, j_l) \geq (i_l - i_1) + (j_l - j_{d_k}).$$

Now we can use the inequalities derived earlier to get:

$$\Delta(i_l, j_l) \geq (l - 1) + (d_k - l) = d_k - 1.$$

Now consider the particular value of k with the largest number of k-dominant matches. A LCS of α and β can be constructed using only dominant matches, and then it must use one of these k-dominant matches, say (i_l, j_l). Now if (i_l, j_l) is a match that is used in the LCS of α and β, then $\Delta \geq \Delta(i_l, j_l)$. Therefore $\Delta + 1 \geq d_k$. Now since d_k is at least as great as any other d_l, for $1 \leq l \leq \rho$ then $\rho d_k \geq d$. Combining these two inequalities, we get $\rho(\Delta + 1) \geq d$.

A corollary of this theorem is that Apostolico and Guerra's $O(d \log n)$ LCS algorithm is also bounded by $O(n \log n \Delta)$ time, and so is never more than a log factor slower than Myer's $O(n\Delta)$ algorithm.

2.3 Ordered Additive Monoids as Weight Systems

As we have seen with the minimal distance LCS problem, the weight of a matched pair of symbols may not be just a simple value but can be made up from different components. For MDLCS, the components are the length of a common subsequence and a weight based on the distance among matched symbols as measured by their positions in the given sequences. Therefore, it is necessary to talk about more general weight systems. To this end, we define a class of objects called *ordered additive monoids*. An ordered additive monoid is a triple $(M, +, \leq)$ such that:

1. M is a set with a distinguished element 0.
2. For all $x \in M$, $x + 0 = x$.
3. For all $x, y \in M, x + y = y + x \in M$.
4. For all $x, y, z \in M, (x + y) + z = x + (y + z)$.
5. M is linearly ordered with respect to $\leq$.
6. For all $x, y \in M, x \leq x + y$ and $y \leq x + y$.

A simple example of an ordered additive monoid is the non-negative real numbers. A more nontrivial example is the monoid defined on the power set 2^S of a given set S. In this case, $+$ is set union, and $\leq$ can be taken as any linear extension of the partial order on 2^S defined by inclusion. As seen with the MDLCS problem, of interest to us is the fact that given two ordered additive monoids $(M, +_M, \leq_M)$ and $(N, +_N, \leq_N)$, we can construct a new ordered additive monoid on the set $M \times N$ by defining for all (u, v) and (x, y) in $M \times N$:

1. $(u, v) + (x, y) = (u +_M x, v +_N y)$.
2. $(u, v) \leq (x, y)$ if $u \leq_M x$ or $u = x$ and $v \leq_N y$.

As MDLCS suggested, we need to consider weight functions that depend on both symbols and their positions. Let N be the non-negative integers. Let A be an alphabet and M be an ordered additive monoid. Let M^+ be the set of non-zero elements of M. For increasing subsequence problems, a weight function is a map $\omega : N \times A \mapsto M^+$. The weight of a sequence σ over A is defined as $\sum_{1 \leq k \leq l} \omega(k, \sigma_k)$.

For common subsequence problems, there are two involved sequences over the alphabet A, α and β. A weight function is a function $\omega : N \times N \times A \mapsto M^+$ where M is again some ordered additive monoid. The weight of a common subsequence γ is defined as: $\sum_{1 \leq k \leq l} \omega(i_k, j_k, \gamma_k)$. Here i_k and j_k are the indices of matched pairs as they appear in the original sequences α and β.

It is not hard to see that the dynamic programming method for LCS can be extended to solve the HCS problem. Let Ω_{ij} be the weight of an HCS of $\alpha_{1\ldots i}$ and $\beta_{1\ldots j}$. Define w_{ij} as 0 if $\alpha_i \neq \beta_j$ and $\omega(i, j, \alpha_i)$ if the two symbols are the same. Below is the recursion to compute Ω_{ij}:

$$\Omega_{ij} = \max(\Omega_{i-1,j}, \Omega_{i,j-1}, \Omega_{i-1,j-1} + w_{ij})$$

2.4 Algorithm Presentation

We shall present algorithms in a pseudo-C syntax. Each algorithm is described with line numbers which are used in subsequent discussions. Frequently, we need to deal with ordered lists. Given a list `L` of objects of certain type, we shall require the following operations on `L`:

`insert(L,o)`: insert the object `o` into the list `L`.
`delete(L,o)`: delete the object `o` from the list L.
`next(L,o)`: find the least element strictly larger than `o` in `L`.
`prev(L,o)`: find the largest element strictly smaller than `o` in `L`
`max(L)`: find the maximal element in `L`.
`min(L)`: find the minimal element in `L`.

An important note on these list operations is that, using balanced tree structures, they can all be performed in $O(\log n)$ time where n is the number of objects involved. In practice, we use splay trees[19]. They are simple to implement, use less space, and work just as well as balanced trees. In the algorithms, ϕ will stand for some undefined object. C programmers may think of this as the `NULL` pointer. The operations `next()`, `prev()`, `max()` and `min()` when not defined will return ϕ. `next()` and `prev()` do not require that the argument object be already in `L` but it has to be of the right type. For convenience, `next(L,`ϕ`)` is equivalent to `min(L)`. Similarly, `prev(L,`ϕ`)` is equivalent to `max(L)`.

3 Computing a Heaviest Increasing Subsequence

The Robinson-Schensted algorithm computes a pair of tableaux from a sequence. For the purpose of computing an LIS, we don't need the entire algorithm, only the part that computes the left-most column of the left tableau. Figure 1 shows the simplified LIS algorithm.

```
 1. lis(σ1σ2···σn)
 2. {   L = φ;
 3.     for(i = 1; i <= n; i = i+1)
 4.     {   s = prev(L,σi);
 5.         t = next(L,s);
 6.         if(t != φ)
 7.             delete(L,t);
 8.         insert(L,σi);
 9.         node[σi] = newnode(σi, node[s]);
10.     }
11. }
```

Fig. 1. The Robinson-Schensted LIS algorithm

Remarks on Figure 1

2: This line initializes the left-most column `L` of the Young tableau.

4: This line computes an element `s` in `L` where the current symbol can be appended while maintaining the invariant that `L` is strictly increasing.

5-7: These lines replace the element after `s` with σ_i. In tableau parlance, `t` is *bumped* by σ_i.

8: `node` is an auxiliary array that, for each element in `L`, contains a record of an element that precedes this element in an increasing subsequence. The function `newnode()` constructs such records and links them into a directed graph. At the end of the algorithm, we can search from the maximal element of `L` to recover an LIS of σ.

Note that in the `lis()` algorithm, at any given time, the length of an LIS of the prefix of σ considered thus far is kept implicitly as the height of the list `L`. For the weighted case, we must maintain the weight of an HIS of a prefix explicitly. Thus, the elements of `L` are pairs (s, w) with $s \in \sigma$ and w is the total weight of an HIS ending

with s. We maintain the invariant that L is strictly increasing in both coordinates. Therefore, the ordering based on their first coordinate (the alphabet ordering) can be used to order L. Figure 2 shows the HIS algorithm.

```
 1. his(σ1σ2···σn, Ω)
 2. {   L = φ;
 3.     for(i = 1; i <= n; i = i+1)
 4.     {   (s,v) = prev(L,(σi,0));
 5.         (t,w) = next(L,(s,v));
 6.         while((t,w) != φ)
 7.         {   if( v+Ω(i,σi) < w )
 8.                 break;
 9.             delete(L,(t,w));
10.             (t,w) = next(L,(t,w));
11.         }
12.         if((t,w) == φ || σi < t)
13.         {   insert(L,(σi,v+Ω(i,σi)));
14.             node[σi] = newnode(σi, node[s]);
15.         }
16.     }
17. }
```

Fig. 2. A $O(n \log n)$ HIS algorithm

Remarks on Figure 2

4: `prev()` computes the largest element `(s,v)` in L such that `s` is strictly smaller than σ_i. This means that σ_i can be appended to any increasing subsequence ending at `s` to define a new increasing subsequence. If there is no such `(s,v)`, we define `v` to be 0.

5-15: These lines replace lines 5–9 of the `lis()` algorithm. Bumping is done in the `while()` loop between lines 6–11. This ensures the invariant that the second coordinates of objects in L are strictly increasing. Line 12 tests to see if $(\sigma_i, v + \Omega(i, \sigma_i)))$ can be inserted into the list L while maintaining the invariant that the first coordinates of objects are strictly increasing. This test is needed because our weight function is also based on the indices of symbols. It can be omitted if the weight function only depends on the symbols. Line 13 does the actual insertion. Line 14 constructs a record so we can recover an actual HIS when the algorithm terminates.

To see how the algorithm runs, consider the sequence 9,2,6,1,1,2,5 in which all elements have their integral values as weights except that the first 1 has weight 2. Below is the progression of the list L as elements are processed:

9	2	6	1	1	2	5
9,9	2,2	2,2	1,2	1,2	1,2	1,2
	9,9	6,8	6,8	6,8	2,4	2,4
		9,9	9,9	9,9	6,8	5,9
					9,9	

Now, consider the list L after each iteration of the `for(;;)` loop. For convenience, we shall use L_i to denote the state of L after the ith iteration. Each element `(s,v)` on L defines an increasing subsequence ending at `s` by tracing the links created on line 14. We shall say that this sequence is defined by `s`.

We claim that for every increasing subsequence $s_1 s_2 \cdots s_k$ of $\sigma_{1\ldots i}$, there is an element $t \leq s_k$ in L_i that defines an increasing subsequence that is at least as heavy as $s_1 s_2 \cdots s_k$. From this, it follows that the maximal element of L_i defines an HIS of $\sigma_{1\ldots i}$. Thus, when the algorithm ends, the maximal element of L defines an HIS for the entire sequence σ.

We prove the claim by induction on `i`, the index variable of the `for(;;)` loop. The case `i=1` is clear. Now, assume the assertion for `i-1` and consider an increasing subsequence $s_1 s_2 \cdots s_k$ of $\sigma_1 \cdots i$. Consider the case when $s_k = \sigma_i$. By induction, there is an element $t \leq s_{k-1}$ in L_{i-1} that defines an increasing sequence that is at least as heavy as $s_1 \cdots s_{k-1}$. Since $t < \sigma_i$, t cannot be bumped off L on lines 6-11. After the ith iteration, either σ_i was inserted into L or it is already in L. Since L is strictly increasing in the weights, the sequence defined by σ_i satisfies the claim. So, assume that $s_k \neq \sigma_i$. Now there are two cases. The case $s_k < \sigma_i$ follows immediately since the part of L preceding σ_i is unchanged in the ith iteration. Assume $s_k > \sigma_i$, by the induction hypothesis, there is a sequence defined by some t in L_{i-1} that is at least as heavy as $s_1 \cdots s_k$. Now, either t is still in L_i and we are done, or t was bumped off L in the `while()` loop of lines 6-11. In this case, the `if()` statement of line 7 guarantees that the sequence defined by σ_i will be at least as heavy as the sequence defined by t (in the $i-1$st step). This complete the proof of the claim. Therefore, the algorithm `his()` is correct.

To analyze the time complexity of `his()`, we observe that all operations in each iteration of the `for(;;)` take $O(\log n)$ time. Since the loop iterates n times, the total time is $O(n \log n)$. We have proved:

Theorem 2. *Let σ be a sequence over a linearly ordered alphabet A and Ω a weight function from A to some ordered additive monoid M. Algorithm* `his(σ,Ω)` *computes a heaviest increasing subsequence of σ in time $O(n \log n)$ where n is the length of σ.*

4 Computing a Heaviest Common Subsequence

Let N be the set of natural numbers. A *biletter* is an element of the set $N \times N$. Given an instance of a common subsequence problem, i.e., two sequences over some alphabet A, $\alpha = \alpha_1 \alpha_2 \cdots \alpha_m$ and $\beta = \beta_1 \beta_2 \cdots \beta_n$, we can construct from these sequences a corresponding *biword* (sequence of biletters) as in Figure 3.

For example, given the sequences `abac` and `baba`, `biword(abac,baba)` will construct the biword:

1 1 2 2 3 3
4 2 3 1 4 2

It is not hard to see by induction on `i` that every common subsequence of α and β maps to an increasing subsequence of the lower word of the biword. On the other hand, given an increasing subsequence of the lower word of the biword, it is easy to invert the indices and retrieve a common subsequence between α and β.

```
1. biword(α,β)
2. {   B = φ;
3.     for(i = 1; i <= m; i = i+1)
4.     {   Let P be the list of positions of α_i in β;
5.         for(k = max(P); k != φ; k = prev(P,k))
6.             append (i,k) to B;
7.     }
8. }
```

Fig. 3. The HCS-HIS correspondence

Let $\Omega : N \times N \times A \mapsto M$ be a weight function. We assign the weight of a matched pair of symbols to the lower part of the corresponding biletter. Then, `biword()` is a weight preserving function that maps bijectively every common subsequence of α and β to an increasing subsequence of the corresponding biword with the same weight.

Applying the `his()` algorithm from the last section to the lower part of the biword, we immediately have an algorithm for computing HCS. Of course, in practice there is no need to ever construct the biword explicitly. It can be generated from the lists of positions in β of the given symbols. The Hunt-Szymanski algorithm is essentially `lis()` where the biword is constructed on the fly. Figure 4 shows `hcs1()`, an algorithm for HCS. In the same way that `his()` is a generalization of the Robinson-Schensted algorithm `lis()`, this algorithm is a generalization of the Hunt-Szymanski algorithm.

The correctness of `hcs1()` follows immediately from that of `his()` and the discussions on the correspondence. The run time of `hcs1()` depends on r, the total number of matches between α and β, and the size of the list `L` which is less than $\min(n, m)$. Assume that $n < m$, we have:

Theorem 3. *Let α and β be two given sequences over an alphabet A. Let $\Omega : N \times N \times A \mapsto M$ be a weight function. Algorithm* `hcs1(α,β,Ω)` *computes a heaviest common subsequence between α and β in time $O((r+m)\log n)$.*

To see the algorithm working, consider the sequences `abca` and `aabd`. Let the weight of a matched symbol at positions `i` and `j` be the ordered pair $(1, 4-|i-j|)$. For example, the weight of the match of the symbol `b` is $(1, 3)$. This is the MDLCS weight function. The corresponding biword is:

```
1 1 2 4 4
2 1 3 2 1
```

Below is the progress of the list `L` as the biletters are processed. The algorithm shows that the HCS is the sequence `ab` at positions 1,2 of `abca` and at positions 1,3 of `aabd`.

(1,2)	(1,1)	(2,3)	(4,2)	(4,1)
2,(1,3)	1,(1,4)	1,(1,4)	1,(1,4)	1,(1,4)
		3,(2,7)	3,(2,7)	3,(2,7)

```
 1. hcs1(α1α2···αm,β1β2···βn,Ω)
 2. {   for(i = 1; i <= n; i = i+1)
 3.         insert(Position[βi],i);
 4.     L = φ;
 5.     for(i = 1; i <= m; i = i+1)
 6.     {   P = Position[αi];
 7.         for(j = max(P); j != φ; j = prev(P,j))
 8.         {   (s,v) = prev(L,(j,0));
 9.             (t,w) = next(L,(s,v));
10.             while((t,w) != φ)
11.             {   if(v+Ω(i,j,αi) < w)
12.                     break;
13.                 delete(L,(t,w));
14.                 (t,w) = next(L,(t,w));
15.             }
16.             if((t,w) == φ || j < t)
17.                 insert(L,(j,v+Ω(i,j,αi)));
18.         }
19.     }
20. }
```

Fig. 4. An $O(r \log n)$ HCS algorithm

Remarks on Figure 4

2-3: For each symbol in β, an ordered list of its positions is constructed.

4: This line initializes the list `L` as in algorithm `his()`. Each object to be stored in `L` compose from the lower part of a biletter (i.e., the matched index in β) and the weight of some corresponding common subsequence defined by this symbol.

5-7: Here, the two `for(;;)` loops essentially construct the corresponding biword on the fly. Note that the decreasing order processing of indices of matched symbols on line 7 is crucial for the correctness of the algorithm. This points out a bug in the original Apostolico-Guerra algorithm which traverses the lists of matched indices in increasing order.

8-17: These lines are straightforward translation of lines 4-15 in algorithm `his()`. For clarity, we omitted the construction of the linked list to retrieve an HCS. Note that on line 8, the function call `prev(L,(j,0))` works because we are assuming that the implementation orders `L` by the matched indices only, not the weights. The `while()` loop on lines 10-15 ensures that the weights are strictly increasing on `L`.

5 Tuning the HCS Algorithm

The algorithm `hcs1`(α,β,Ω) can be tuned further if we know more about the weight function Ω. This section considers a few main cases in which weights are known to follow some regular patterns. The complexity analyses of the tuned algorithms is based on weighted *dominant matches* which are defined as follows: A match $\alpha_i = \beta_j$ is dominant if every HIS of $\alpha_{1...i}$ and $\beta_{1...j}$ must end at α_i and β_j. Again, following standard conventions, we let d be the total number of weighted dominant matches.

Recall from section 2 that if Ω_{ij} is defined as the weight of a HIS between $\alpha_{1...i}$

and $\beta_{1..j}$, then $\Omega_{ij} = max(\Omega_{i-1,j}, \Omega_{i,j-1}, w_{ij})$ where w_{ij} is $\Omega_{i-1,j-1}$ if $\alpha_i \neq \beta_j$ or $\Omega_{i-1,j-1}+\omega(i,j,\alpha_i)$ if they are equal. It can be seen easily by induction that a match is dominant whenever Ω_{ij} is defined by w_{ij}. That is, a match $\alpha_i = \beta_j$ is dominant when $w_{ij} > \Omega_{i-1,j}$ and $w_{ij} > \Omega_{i,j-1}$.

Assume an instance of the HCS problem with sequences α, β, and weight function Ω. We say that Ω is β-decreasing, if for every symbol in α, the weights of its matches are decreasing (but not necessarily strictly decreasing) as they appear from left to right in β. On the other hand, if for every symbol in α, the weights of its matches in β strictly increase from left to right, we say that Ω is β-increasing. We similarly define α-decreasing and α -increasing. The below result follows from a simple induction. It shows that algorithm `hcs1()` runs in $O(d \log n)$ for weight systems that are increasing.

Theorem 4. *If the weight function Ω is α-increasing and β-increasing, then every match is a dominant match.*

The rest of this section shows tunings of the algorithm based on whether or not the weight functions are α-decreasing, β-decreasing or both. We shall state conditions when the algorithms perform in $O(d \log n)$ time.

5.1 Computing HCS for β-Decreasing Weights

In this case, let `s` and `j` be defined as on lines 7–8 of algorithm `hcs1()`. The reverse order insertion of the sequence $s < j_1 < \cdots < j_k = j$ eventually amounts to the insertion of just j_1 since it is heaviest. This means that intermediate insertions can be avoided by directly computing j_1. Figure 5 shows the modified HCS algorithm. Line 10 is the new addition to algorithm `hcs1()`.

Theorem 5. *If the weight function Ω is β-decreasing and α-increasing, then algorithm* `hcs2()` *runs in $O(d \log n)$ time.*

5.2 Computing HCS for α-Decreasing Weights

In this case, consider lines 8–9 of algorithm `hcs1()`. If `j` is currently on the list `L`, these lines will define `t` to be `j`. If the element immediately precedes `j` in `L` has not changed since `j` was inserted into `L`, then because the weight of the new `j` is less than the one already in `L`, lines 10–17 will leave `L` unchanged. This means that when an element `j` is inserted into `L`, we can delete it from its position list to avoid duplicate processing. However, we must insert it back into the position list if it gets removed from `L` or if its predecessor in `L` ever changes. Figure 6 shows the modified algorithm.

Theorem 6. *If the weight function Ω is α-decreasing and β-increasing, then algorithm* `hcs3()` *runs in $O(d \log n)$ time.*

```
1. hcs2(α1α2···αm,β1β2···βn,Ω)
2. {   for(i = 1; i <= n; i = i+1)
3.         insert(Position[βi],i);
4.     L = φ;
5.     for(i = 1; i <= m; i = i+1)
6.     {   P = Position[αi];
7.         for(j = max(P); j != φ; j = prev(P,j))
8.         {   (s,v) = prev(L,(j,0));
9.             (t,w) = next(L,(s,v));
10.            j = next(P,s);
11.            while((t,w) != φ)
12.            {   if(v+Ω(i,j,αi) < w)
13.                    break;
14.                delete(L,(t,w));
15.                (t,w) = next(L,(t,w));
16.            }
17.            if((t,w) == φ || j < t)
18.                insert(L,(j,v+Ω(i,j,αi)));
19.        }
20.    }
21. }
```

Fig. 5. An HCS algorithm for decreasing weights in β

5.3 Computing HCS When Weights are Position-Independent

An important special case that finds many practical applications is when the weights are dependent only on the symbols in the alphabet. In this case, both conditions of algorithms `hcs2()` and `hcs3()` apply. Further, the test on line 16 of `hcs1()` is not needed as we noted in the remarks following algorithm `his()`. Putting everything together, we have algorithm `hcs4()` (Figure 7) for computing an HCS when weights are position-independent. When all weights are constant, `hcs4()` reduces to the Apostolico-Guerra LCS algorithm.

5.4 Conclusions

In this paper, we defined the heaviest increasing subsequence and heaviest common subsequence problems as natural generalizations of the longest increasing and longest common subsequence problems. These problems are intimately related by a weight-preserving correspondence. We showed how to generalize the Robinson-Schensted LIS algorithm to solve the HIS problem in $O(n \log n)$ time. Then using the weight-preserving correspondence, we applied the new HIS algorithm to solve the HCS problem in $O(r \log n)$ time, where r is the number of matches. This algorithm is a generalization of the Hunt-Szymanski LCS algorithm. We showed through a sequence of simple refinements how to tune the HCS algorithm when weights followed certain regular patterns. In particular, when weights are position-independent, our HCS algorithm can be viewed as a generalization of the Apostolico-Guerra LCS algorithm. Typically, computing an HCS may require much fewer matches than the entire set of

```
 1. hcs3(α1α2···αm,β1β2···βn,Ω)
 2. {   for(i = 1; i <= n; i = i+1)
 3.         insert(Position[βi],i);
 4.     L = φ;
 5.     for(i = 1; i <= m; i = i+1)
 6.     {   P = Position[αi];
 7.         for(j = max(P); j != φ; j = prev(P,j))
 8.         {   (s,v) = prev(L,(j,0));
 9.             (t,w) = next(L,(s,v));
10.             while((t,w) != φ)
11.             {   if(j < t)
12.                     insert(Position[βt],t);
13.                 if(v+Ω(i,j,αi) < w)
14.                     break;
15.                 delete(L,(t,w));
16.                 (t,w) = next(L,(t,w));
17.             }
18.             if((t,w) == φ || j < t)
19.             {   insert(L,(j,v+Ω(i,j,αi)));
20.                 delete(P,j);
21.             }
22.         }
23.     }
24. }
```

Fig. 6. An HCS algorithm for decreasing weights in α

Remarks on Figure 6

11-12: These lines implement the condition that a position `t` on `L` must be reinserted into its `Position` list if it gets bumped off `L` or if the element preceding it in `L` changes.

20: This line removes `j` from its `Position` list after it gets inserted into `L` so that redundant processing of `j` is avoided.

matches. We defined generalized *dominant matches*, and specified conditions under which all of our HCS algorithms would run in $O(d \log n)$ time where d is the number of dominant matches.

The Robinson-Schensted LIS algorithm is central in the combinatorial theory of tableaux and plane partitions. Our extension of the algorithm indicates that many of the interesting results in the theory may extend. Of even more interest is the weight-preserving correspondence between the HIS and HCS problems. In future work, we hope it will show new ways in using the machinery of tableau and plane partition theory to find out more about the structure of common subsequence problems.

```
 1. hcs4(α1α2···αm,β1β2···βn,Ω)
 2. {   for(i = 1; i <= n; i = i+1)
 3.         insert(Position[βi],i);
 4.     L = φ;
 5.     for(i = 1; i <= m; i = i+1)
 6.     {   P = Position[αi];
 7.         for(j = max(P); j != φ; j = prev(P,j))
 8.         {   (s,v) = prev(L,(j,0));
 9.             (t,w) = next(L,(s,v));
10.             j = next(P,s);
11.             while((t,w) != φ)
12.             {   insert(Position[βt],t);
13.                 if(v+Ω(αi) < w)
14.                     break;
15.                 delete(L,(t,w));
16.                 (t,w) = next(L,(t,w));
17.             }
18.             insert(L,(j,v+Ω(αi)));
19.             delete(P,j);
20.         }
21.     }
22. }
```

Fig. 7. An HCS algorithm for position-independent weights

References

1. A. V. Aho, D. S. Hirschberg, and J. D. Ullman. Bounds on the complexity of the longest common subsequence problem. *JACM*, 23(1):1–12, 1976.
2. A. Apostolico and C Guerra. The longest common subsequence problem revisited. *Algorithmica*, 2:315–336, 1987.
3. Francis Y. L. Chin and C. K. Poon. A fast algorithm for computing longest common subsequences of small alphabet size. *Journal of Information Processing*, 13(4):463–469, 1990.
4. Michael L. Fredman. On computing the length of longest increasing subsequences. *Discrete Mathematics*, 11:29–35, 1975.
5. E. Gansner. *Matrix Correspondences and the Enumeration of Plane Partitions.* PhD thesis, MIT, Cambridge, MA, 1978.
6. M. Golumbic. *Algorithmic Graph Theory and Perfect Graphs.* Academic Press, 1980.
7. Daniel S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *CACM*, 18(6):341–343, 1975.
8. Daniel S. Hirschberg. Algorithms for the longest common subsequence problem. *JACM*, 24(4):664–675, 1977.
9. W. J. Hsu and M. W. Du. New algorithms for the LCS problem. *JCSS*, 29:133–152, 1984.
10. J. W. Hunt and M. D. McIlroy. An algorithm for differential file comparison. Computer Science Technical Report 41, Bell Laboratories, 1975.
11. James W. Hunt and Thomas G. Szymanski. A fast algorithm for computing longest common subsequences. *CACM*, 20(5):350–353, 1977.

12. D. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, Reading, MA, 1973.
13. William J. Masek and Michael S. Patterson. A faster algorithm computing string edit distances. *JCSS*, 20:18–31, 1980.
14. Eugene W. Myers. An $O(ND)$ difference algorithm and its variations. *Algorithmica*, 1:251–266, 1986.
15. N. Nakatsu, Y. Kambayashi, and S. Yajima. A longest common subsequence algorithm suitable for similar text strings. *Acta Informatica*, 18:171–179, 1982.
16. G. De B. Robinson. On the representations of the symmetric group. *American J. Math.*, 60:745–760, 1938.
17. D. Sankoff and J.B. Kruskal. *Time Warps, String Edits and Macromolecules: The Theory and Practice of Sequence Comparisons.* Addison Wesley, Reading, MA, 1983.
18. C. Schensted. Largest increasing and decreasing subsequences. *Canadian J. Math.*, 13:179–191, 1961.
19. D. Sleator and R. Tarjan. Self-adjusting binary trees. *JACM*, 32:652–686, 1985.
20. R. Stanley. Theory and applications of plane partitions. *Stud. Applied Math.*, 50:259–279, 1971.
21. K.-P. Vo. More `<curses>`: the `<screen>` library. Technical report, AT&T Bell Laboratories, 1986.
22. K.-P. Vo and R. Whitney. Tableaux and matrix correspondences. *J. of Comb. Theory, Series A*, 35:323–359, 1983.
23. R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *JACM*, 21(1):168–173, 1974.
24. Sun Wu, Udi Manber, Gene Myers, and Webb Miller. An $O(NP)$ sequence comparison algorithm. *Information Processing Letters*, 35(6):317–323, 1990.
25. A. Young. The collected papers of alfred young. *Math. Exp.*, 21, 1977.