INFORMATIK

A Faster Algorithm for Computing a Longest Common Increasing Subsequence

Irit Katriel and Martin Kutz

MPI–I–2005–1–007 March 2005

FORSCHUNGSBERICHT RESEARCH REPORT

MAX-PLANCK-INSTITUT
FÜR
INFORMATIK

Stuhlsatzenhausweg 85 66123 Saarbrücken Germany

Authors' Addresses

Irit Katriel, Martin Kutz
Stuhlsatzenhausweg 85
Max-Planck-Institut für Informatik,
66123 Saarbrücken, Germany
email: `{irit,mkutz}@mpi-sb.mpg.de`

Abstract

Let $A = \langle a_1, \ldots, a_n \rangle$ and $B = \langle b_1, \ldots, b_m \rangle$ be two sequences with $m \geq n$, whose elements are drawn from a totally ordered set. We present an algorithm that finds a longest common increasing subsequence of A and B in $O(m \log m + n\ell \log n)$ time and $O(m + n\ell)$ space, where ℓ is the length of the output. A previous algorithm by Yang et al. needs $\Theta(mn)$ time and space, so ours is faster for a wide range of values of m, n and ℓ.

Keywords

Algorithms, Bounded Heap, Data Structures, Longest Common Increasing Subsequence, Pattern Matching

1 Introduction

Given two sequences $A =< a_1, \ldots, a_n >$ and $B =< b_1, \ldots, b_m >$ with $m \geq n$, a *common subsequence* of A and B is a subsequence $\langle a_{j_1} = b_{\kappa_1}, a_{j_2} = b_{\kappa_2}, \ldots a_{j_\ell} = b_{\kappa_\ell} \rangle$, where $j_1 < j_2 < \cdots < j_\ell$ and $\kappa_1 < \kappa_2 < \cdots < \kappa_\ell$.

Given one sequence $A =< a_1, \ldots, a_n >$ where the a_i's are drawn from a totally ordered set, an *increasing subsequence* of A is a subsequence $\langle a_{j_1}, a_{j_2}, \ldots, a_{j_\ell} \rangle$ such that $j_1 < j_2 < \cdots < j_\ell$ and $a_{j_1} < a_{j_2} < \cdots < a_{j_\ell}$.

Algorithms that search for the longest common subsequence (LCS) or the longest increasing subsequence (LIS) date back several decades. See, e.g., [1, 2, 3, 5, 6].

However, only recently Yang et al. [7] combined the two concepts, and defined the *common increasing subsequence* (CIS) of two sequences A and B, i.e., an increasing sequence which is a subsequence of both A and B. They designed an algorithm that finds a *longest CIS* (LCIS) of A and B using $\Theta(mn)$ time and space.

In this paper we present an algorithm for the LCIS problem which runs in $O(m \log m + n\ell \log n)$ time and $O(m + n\ell)$ space, where ℓ is the length of the LCIS. Whenever $n = \Omega(\log m)$ and either $m = \Omega(n \log n)$ or $\ell = o(n/\log n)$, it is faster than $\Theta(mn)$.

In Section 2, we construct a data structure which we call a *bounded heap* and which will be used by our LCIS algorithm. In Section 3 we present the algorithm and prove its correctness.

2 The Bounded-Heap Data Structure

A *bounded heap* (BH) is a data structure that resembles a priority queue, but also allows us to bound our queries. That is, we can ask for the minimum priority among all items in the heap whose keys are smaller than k. In this section we describe how to implement a bounded heap that supports the following operations:

- *Insert*$(\mathcal{H}, k, p, d)$: Insert into the *BH* $\mathcal{H}$ the key k with priority p and associated data d.
- *DecreasePriority*$(\mathcal{H}, k, p, d)$: If the *BH* $\mathcal{H}$ does not contain the key k, perform *Insert*$(\mathcal{H}, k, p, d)$. Otherwise, set this key's priority to $\min\{p, p'\}$, where p' is its previous priority.
- *BoundedMin*$(\mathcal{H}, k)$: Return the item that has minimum priority among all items in $\mathcal{H}$ with key smaller than k. If $\mathcal{H}$ does not contain any items with key smaller than k, return "invalid".

Assume that the keys are drawn from the set $\{\sigma_1, \sigma_2, \ldots, \sigma_{|\Sigma|}\}$ which is totally ordered, i.e., $\sigma_1 \leq \sigma_2 \leq \cdots \leq \sigma_{|\Sigma|}$. Let $BM(k)$ denote the priority of the item returned by the query $BoundedMin(\mathcal{H}, k)$. Clearly, $BM(\sigma_i) \leq BM(\sigma_{i-1})$ for any $1 < i \leq |\Sigma|$ (see Figure 1).

key$k$	1	2	3	4	5	6	7	8	9	10
priority	7	10	6	8	5	3	2	4	1	9
$BM(k)$	∞	7	7	6	6	5	3	2	2	1

Figure 1: Example of *BM* values.

Since we only need to support *BoundedMin* queries (and not queries about the priority of a specific key), it suffices to keep in our data structure only the smallest key for each *BM* value. These keys will be the leaves of a balanced search tree, sorted from left to right by increasing key order. Along with the key, we also store the corresponding *BM* value in each leaf. For the example in Figure 1, the data structure will contain the (key, BM) pairs $(1, \infty)$, $(2, 7)$, $(4, 6)$, $(6, 5)$, $(7, 3)$, $(8, 2)$, $(10, 1)$.

A $BoundedMin(\mathcal{H}, \sigma_i)$ query is handled by searching for the largest key which is smaller than σ_i. The *BM* value stored with this key is valid for σ_i (by definition).

What we achieved by this compression is that now we can efficiently support *DecreasePriority* operations. Assume that the priority of σ_i decreased from p' to p and that p is smaller than $BM(\sigma_j)$ for $i < j \leq i'$. This means that all leaves in the tree which correspond to keys in $\{\sigma_{i+1}, \sigma_{i+2}, \ldots, \sigma_{i'-1}\}$ need to be removed. This takes time $O(\log n + i' - i)$; we need to remove an interval of $O(i' - i)$ leaves as well as $O(i' - i)$ internal nodes. Finally, a leaf with key σ_{i+1} and *BM* value p is inserted to the tree.

An $Insert(\mathcal{H}, k, p, d)$ operation is handled as if it was a $DecreasePriority(\mathcal{H}, k, p, d)$ operation. To see that this works, note that the tree would not change if we were to insert the key k with priority ∞.

The $O(i' - i)$ time of *DecreasePriority* can be charged to the insertions of the leaves that were deleted. That is, when a new leaf is inserted, it receives a constant number of tokens with which it can pay for the *DecreasePriority* operation that caused its deletion from the tree. We get that on a bounded heap containing n items, *Insert* and *DecreasePriority* take $O(\log n)$ amortized time, and *BoundedMin* takes $O(\log n)$ worst case time.

3 The Algorithm

The algorithm appears in Figure 2. In the preprocessing step, it (1) Removes from each sequence all elements which do not appear in the other ("cleanup"), and (2) For every remaining element σ, generates a balanced search tree T_σ that contains ∞ and the indices of all occurances of σ in B.

Then, the algorithm identifies common increasing subsequences (CISs). In iteration i it identifies CISs of length i (using the results of iteration $i-1$). More precisely, for every element a_j in A, it identifies the minimum index κ in B such that there is a length-i CIS which ends at a_j in A and at b_κ in B. The index κ is stored in $L_i[j]$.

To compute the array $L_1[1 \dots n]$, the algorithm traverses A and for each a_j, sets $L_1[j]$ to be the minimum index in the tree T_{a_j}, i.e., the earliest occurance of a_j in B. Note that due to step (1) of the preprocessing, $L_1[j]$ is finite.

For $i > 1$, the ith iteration proceeds as follows. The algorithm traverses A again, and for every a_j, it checks whether a_j (together with some b_κ) can extend a CIS of length $i-1$ to a CIS of length i, and if so, identifies the minimum such κ. For this purpose, the algorithm maintains a bounded heap $\mathcal{H}$. When it begins processing a_j, $\mathcal{H}$ contains all elements $a_t \in \{a_1, \dots, a_{j-1}\}$ for which $L_{i-1}[t] \neq \infty$. The key of a_t in $\mathcal{H}$ is a_t, and its priority is $L_{i-1}[t]$, i.e., the minimum index of the endpoint in B of a length-$(i-1)$ CIS which ends, in A, at index t. The algorithm queries $\mathcal{H}$ to find the leftmost endpoint (in B) of a length-$(i-1)$ CIS which contains only elements smaller than a_j. Let κ' be this endpoint. Then, $L_i[j]$ is set to the first occurance of a_j in B which is after κ' – we will prove that this is the leftmost endpoint in B of a length-i CIS which ends, in A, at a_j.

The arrays $Link_1, Link_2, \dots$ are used to save the information we need in order to construct the LCIS: Whenever we detect that the index pair (j, κ) can extend a length-$(i-1)$ CIS which ends at the index pair (j', κ'), we set $Link_i[j] = j'$. Finally, if there is a length-$(i-1)$ CIS which ends at a_j, then a_j is inserted into $\mathcal{H}$ with priority $L_{i-1}[a_j]$; it may later be extended into a length-i CIS by some $a_{j'}$ with $j' > j$.

3.1 Proof of Correctness

The correctness of the algorithm relies on the following lemma, which states that if there is a solution then the algorithm finds it. It is straighforward to show that the algorithm will not find a CIS that does not exist.

Lemma 1 *Let A and B be two sequences that have a length-ℓ CIS which*

```
Function LCIS(A = ⟨a_1, ... a_n⟩, B = ⟨b_1, ... b_m⟩)
    Preprocess (* Clean A and B and build T_σ for every σ. *)
    i ← 1

    (* Compute L_1[1 ... n] *)
    for j = 1 to n do L_i[j] ← MinimumKey(T_{a_j})

    H ← [] (* Empty Bounded Heap. *)

    (* Main loop: *)
    do
        i ← i + 1
        for j = 1 to n do
            L_i[j] ← ∞
            (j', κ') ← BoundedMin(H, a_j)
            if (j', κ') ≠ "invalid" then
                L_i[j] ← min{κ : κ ∈ T_{a_j} ∧ κ > κ'}
                Link_i[j] = j'
            endif
            if L_{i-1}[j] ≠ ∞ then
                (* Recall that DecreasePriority inserts a_j if it is not already there. *)
                DecreasePriority(H, a_j, L_{i-1}[j], (j, κ))
            endif
        endfor
    while i < n and L_i ≠ ∞^n.

    (* Generate a LCIS in reverse order *)
    if L_i = ∞^n then i ← i - 1
    j ← an index such that L_i[j] ≠ ∞.
    while i > 0 do
        output a_j
        j ← Link_i[j]
        i ← i - 1
    end while
end
```

Figure 2: LCIS Algorithm.

ends in A at index j and in B at index κ. Then at the end of the iteration in which $i = \ell$, $L_\ell[j] \leq \kappa$.

Proof By induction on ℓ. For $\ell = 1$, the claim is obvious. Assume that it holds for any CIS of length $\ell - 1$ and that we are given A and B which have a CIS $c_1, \ldots, c_\ell$ of length ℓ, which is located in A as $a_{j_1}, \ldots, a_{j_\ell}$ and in B as $b_{\kappa_1}, \ldots, b_{\kappa_\ell}$.

By the induction hypothesis, at the end of the $i = \ell - 1$ iteration, L_{i-1} contains entries which are not equal to ∞. Hence, the algorithm will proceed to perform iteration $i = \ell$. Again by the induction hypothesis, $L_{\ell-1}[j_{\ell-1}] \leq \kappa_{\ell-1}$.

Since $a_{j_{\ell-1}} < a_{j_\ell}$, it is guaranteed that when $j = j_\ell$, $\mathcal{H}$ contains an item with key $a_{j_{\ell-1}}$, priority $\kappa' \leq \kappa_{\ell-1}$ and $d = (j_{\ell-1}, \kappa')$. So the *BoundedMin* operation will return a valid value. If the value returned is $(j_{\ell-1}, \kappa_{\ell-1})$, then the smallest occurance of a_ℓ in B after $\kappa_{\ell-1}$ is not beyond κ_ℓ. So the algorithm will set $L_\ell[j_\ell] \leq \kappa_\ell$. On the other hand, if the value returned is not $(j_{\ell-1}, \kappa_{\ell-1})$, then it is $(j_{\ell-1}, \kappa')$ for some $\kappa' \leq \kappa_{\ell-1}$. Since $a_{j'} < a_\ell$, again we get that the smallest occurance of a_ℓ in B after $\kappa_{\ell-1}$ is not beyond κ_ℓ. So the algorithm will set $L_\ell[j_\ell] \leq \kappa_\ell$. ∎

3.2 Complexity

The preprocessing phase takes $O(m \log m)$ time: Eliminating items that appear only in one of the sequences is easy after they are sorted. The construction of the T_σ's takes $O(m)$ time, because we need to build search trees, each over a static, sorted set of indices.

A is traversed $O(\ell)$ times, and in each traversal $O(n)$ operations are performed on balanced search trees of size n, each of which takes $O(\log n)$ amortized time. In total, this takes $O(n\ell \log n)$ time. Constructing the LCIS takes $O(\ell)$ time. We get that the total running time of the algorithm is $O(m \log m + n\ell \log n)$.

As for space complexity, note that in the main loop, we only use L_{i-1} and L_i. Therefore, we do not need to save the previous L's. This means that if we only wish to find the *length* of the LCIS, the space requirement is $O(m + n)$. If we also want to construct the LCIS, we need $O(n\ell)$ space for the *Link* arrays.

Acknowledgement We wish to thank Kanela Kaligosi for her comments on an earlier draft of this paper.

References

[1] L. Bergroth, H. Hakonen, and T. Raita. A survey of longest common subsequence algorithms. In *SPIRE '00: Proceedings of the Seventh International Symposium on String Processing Information Retrieval (SPIRE'00)*, page 39. IEEE Computer Society, 2000.

[2] S. Bespamyatnikh and M. Segal. Enumerating longest increasing subsequences and patience sorting. *Inform. Process. Lett.*, 76(1-2):7–11, 2000.

[3] M.L. Fredman. On computing the length of longest increasing subsequences. *Discrete Mathematics*, 11(1):29–35, 1975.

[4] Daniel S. Hirschberg. Algorithms for the longest common subsequence problem. *J. ACM*, 24(4):664–675, 1977.

[5] James W. Hunt and Thomas G. Szymanski. A fast algorithm for computing longest common subsequences. *Commun. ACM*, 20(5):350–353, 1977.

[6] M.S. Paterson W.J. Masek. A faster algorithm computing string edit distances. *J. Comput. System Sci.*, 20:18–31, 1980.

[7] I-Hsuan Yang, Chien-Pin Huang, and Kun-Mao Chao. A fast algorithm for computing a longest common increasing subsequence. *Information Processing Letters*, 93/5:249–253, 2005.

I N F O R M A T I K

Below you find a list of the most recent technical reports of the Max-Planck-Institut für Informatik. They are available by anonymous ftp from `ftp.mpi-sb.mpg.de` under the directory `pub/papers/reports`. Most of the reports are also accessible via WWW using the URL `http://www.mpi-sb.mpg.de`. If you have any questions concerning ftp or WWW access, please contact `reports@mpi-sb.mpg.de`. Paper copies (which are not necessarily free of charge) can be ordered either by regular mail or by e-mail at the address below.

Max-Planck-Institut für Informatik
Library
attn. Anja Becker
Stuhlsatzenhausweg 85
66123 Saarbrücken
GERMANY
e-mail: `library@mpi-sb.mpg.de`

MPI-I-2005-4-004	C. Theobalt, N. Ahmed, E. De Aguiar, G. Ziegler, H. Lensch, M.A.,. Magnor, H. Seidel	Joint Motion and Reflectance Capture for Creating Relightable 3D Videos
MPI-I-2005-4-003	T. Langer, A.G. Belyaev, H. Seidel	Analysis and Design of Discrete Normals and Curvatures
MPI-I-2005-4-002	O. Schall, A. Belyaev, H. Seidel	Sparse Meshing of Uncertain and Noisy Surface Scattered Data
MPI-I-2005-4-001	M. Fuchs, V. Blanz, H. Lensch, H. Seidel	Reflectance from Images: A Model-Based Approach for Human Faces
MPI-I-2005-2-001	J. Hoffmann, Carla Gomes	Bottleneck Behavior in CNF Formulas
MPI-I-2005-1-007	I. Katriel, M. Kutz	A Faster Algorithm for Computing a Longest Common Increasing Subsequence
MPI-I-2005-1-002	I. Katriel, M. Kutz, M. Skutella	Reachability Substitutes for Planar Digraphs
MPI-I-2005-1-001	D. Michail	Rank-Maximal through Maximum Weight Matchings
MPI-I-2004-NWG3-001	M. Magnor	Axisymmetric Reconstruction and 3D Visualization of Bipolar Planetary Nebulae
MPI-I-2004-NWG1-001	B. Blanchet	Automatic Proof of Strong Secrecy for Security Protocols
MPI-I-2004-5-001	S. Siersdorfer, S. Sizov, G. Weikum	Goal-oriented Methods and Meta Methods for Document Classification and their Parameter Tuning
MPI-I-2004-4-006	K. Dmitriev, V. Havran, H. Seidel	Faster Ray Tracing with SIMD Shaft Culling
MPI-I-2004-4-005	I.P. Ivrissimtzis, W.-. Jeong, S. Lee, Y.a. Lee, H.-. Seidel	Neural Meshes: Surface Reconstruction with a Learning Algorithm
MPI-I-2004-4-004	R. Zayer, C. Rössl, H. Seidel	r-Adaptive Parameterization of Surfaces
MPI-I-2004-4-003	Y. Ohtake, A. Belyaev, H. Seidel	3D Scattered Data Interpolation and Approximation with Multilevel Compactly Supported RBFs
MPI-I-2004-4-002	Y. Ohtake, A. Belyaev, H. Seidel	Quadric-Based Mesh Reconstruction from Scattered Data
MPI-I-2004-4-001	J. Haber, C. Schmitt, M. Koster, H. Seidel	Modeling Hair using a Wisp Hair Model
MPI-I-2004-2-007	S. Wagner	Summaries for While Programs with Recursion
MPI-I-2004-2-002	P. Maier	Intuitionistic LTL and a New Characterization of Safety and Liveness
MPI-I-2004-2-001	H.d. Nivelle, Y. Kazakov	Resolution Decision Procedures for the Guarded Fragment with Transitive Guards
MPI-I-2004-1-006	L.S. Chandran, N. Sivadasan	On the Hadwiger's Conjecture for Graph Products

MPI-I-2004-1-005	S. Schmitt, L. Fousse	A comparison of polynomial evaluation schemes
MPI-I-2004-1-004	N. Sivadasan, P. Sanders, M. Skutella	Online Scheduling with Bounded Migration
MPI-I-2004-1-003	I. Katriel	On Algorithms for Online Topological Ordering and Sorting
MPI-I-2004-1-002	P. Sanders, S. Pettie	A Simpler Linear Time 2/3 - epsilon Approximation for Maximum Weight Matching
MPI-I-2004-1-001	N. Beldiceanu, I. Katriel, S. Thiel	Filtering algorithms for the Same and UsedBy constraints
MPI-I-2003-NWG2-002	F. Eisenbrand	Fast integer programming in fixed dimension
MPI-I-2003-NWG2-001	L.S. Chandran, C.R. Subramanian	Girth and Treewidth
MPI-I-2003-4-009	N. Zakaria	FaceSketch: An Interface for Sketching and Coloring Cartoon Faces
MPI-I-2003-4-008	C. Roessl, I. Ivrissimtzis, H. Seidel	Tree-based triangle mesh connectivity encoding
MPI-I-2003-4-007	I. Ivrissimtzis, W. Jeong, H. Seidel	Neural Meshes: Statistical Learning Methods in Surface Reconstruction
MPI-I-2003-4-006	C. Roessl, F. Zeilfelder, G. Nürnberger, H. Seidel	Visualization of Volume Data with Quadratic Super Splines
MPI-I-2003-4-005	T. Hangelbroek, G. Nürnberger, C. Roessl, H.S. Seidel, F. Zeilfelder	The Dimension of C^1 Splines of Arbitrary Degree on a Tetrahedral Partition
MPI-I-2003-4-004	P. Bekaert, P. Slusallek, R. Cools, V. Havran, H. Seidel	A custom designed density estimation method for light transport
MPI-I-2003-4-003	R. Zayer, C. Roessl, H. Seidel	Convex Boundary Angle Based Flattening
MPI-I-2003-4-002	C. Theobalt, M. Li, M. Magnor, H. Seidel	A Flexible and Versatile Studio for Synchronized Multi-view Video Recording
MPI-I-2003-4-001	M. Tarini, H.P.A. Lensch, M. Goesele, H. Seidel	3D Acquisition of Mirroring Objects
MPI-I-2003-2-004	A. Podelski, A. Rybalchenko	Software Model Checking of Liveness Properties via Transition Invariants
MPI-I-2003-2-003	Y. Kazakov, H. Nivelle	Subsumption of concepts in $DL\ \mathcal{FL}_0$ for (cyclic) terminologies with respect to descriptive semantics is PSPACE-complete
MPI-I-2003-2-002	M. Jaeger	A Representation Theorem and Applications to Measure Selection and Noninformative Priors
MPI-I-2003-2-001	P. Maier	Compositional Circular Assume-Guarantee Rules Cannot Be Sound And Complete
MPI-I-2003-1-018	G. Schaefer	A Note on the Smoothed Complexity of the Single-Source Shortest Path Problem
MPI-I-2003-1-017	G. Schäfer, S. Leonardi	Cross-Monotonic Cost Sharing Methods for Connected Facility Location Games
MPI-I-2003-1-016	G. Schäfer, N. Sivadasan	Topology Matters: Smoothed Competitive Analysis of Metrical Task Systems
MPI-I-2003-1-015	A. Kovács	Sum-Multicoloring on Paths
MPI-I-2003-1-014	G. Schäfer, L. Becchetti, S. Leonardi, A. Marchetti-Spaccamela, T. Vredeveld	Average Case and Smoothed Competitive Analysis of the Multi-Level Feedback Algorithm
MPI-I-2003-1-013	I. Katriel, S. Thiel	Fast Bound Consistency for the Global Cardinality Constraint
MPI-I-2003-1-012		- not published -
MPI-I-2003-1-011	P. Krysta, A. Czumaj, B. Voecking	Selfish Traffic Allocation for Server Farms
MPI-I-2003-1-010	H. Tamaki	A linear time heuristic for the branch-decomposition of planar graphs
MPI-I-2003-1-009	B. Csaba	On the Bollobás – Eldridge conjecture for bipartite graphs
MPI-I-2003-1-008	P. Sanders	Polynomial Time Algorithms for Network Information Flow

MPI-I-2003-1-007	H. Tamaki	Alternating cycles contribution: a strategy of tour-merging for the traveling salesman problem
MPI-I-2003-1-006	M. Dietzfelbinger, H. Tamaki	On the probability of rendezvous in graphs
MPI-I-2003-1-005	M. Dietzfelbinger, P. Woelfel	Almost Random Graphs with Simple Hash Functions