MAXIMAL SETS OF k-INCREASING SUBSEQUENCES WITH APPLICATIONS TO COUNTING POINTS IN TRIANGLES (Extended Abstract)

by

R. Bar-Yehuda and S. Fogel

Technical Report #596
December 1989
Maximal Sets of k-increasing Subsequences with Applications to Counting Points in Triangles
(Extended Abstract)

Reuven Bar Yehuda* and Sergio Fogel
Computer Science Department
Technion - IIT, Haifa 32000, Israel

Abstract

We consider the problem of finding a maximal set of disjoint increasing subsequences of length \( k \) of a given sequence of real numbers. We achieve an \( O(n \log n + mk^2) \) algorithm, where \( m \) is the number of sequences found. Our algorithm is of a geometric nature. Using this result we show how to partition a sequence of \( n \) real numbers into \( 2\sqrt{n} \) monotone subsequences in time \( O(n^{1.5}) \), an improvement over the best previous algorithm.

An application to a fundamental problem in computational geometry is shown. Given \( n \) points in the plane, preprocess them into an \( O(n \log n) \) data structure such that counting points inside a query triangle can be done in time \( O(\sqrt{n \log n}) \). The best preprocessing algorithm for this problem runs in time \( O(n^{1.5} \log n) \). We improve this time to \( O(n^{1.6}) \). In fact, our algorithm permits a linear tradeoff between preprocessing and query time.

*The research was partially supported by Technion VPR Fund- Albert Einstein Research Fund
1 Introduction

Let $A = (a_1, a_2, \ldots, a_n)$ be a sequence of $n$ real numbers. Let $A' = (a_{i_1}, a_{i_2}, \ldots, a_{i_k})$. $A'$ is a subsequence of $A$ if $i_1 < i_2 < \ldots < i_k$. $A'$ is called $k$-increasing if $a_{i_1} \leq a_{i_2} \leq \ldots \leq a_{i_k}$ and $k$-decreasing if $a_{i_1} \geq a_{i_2} \geq \ldots \geq a_{i_k}$.

A famous result by Erdős and Szekeres [ES] states that every sequence $A$ of length $n$ contains either a $\lceil \sqrt{n} \rceil$-increasing or a $\lfloor \sqrt{n} \rfloor$-decreasing subsequence. An immediate consequence of this proposition is that every sequence of $n$ real numbers can be partitioned into $2\lceil \sqrt{n} \rceil$ or less monotone subsequences. An algorithm for finding a $k$-increasing subsequence in time $O(n \log n)$ has long been known (see e.g. [K]). By repeated application of the algorithm, a partition of a sequence into at most $2\lceil \sqrt{n} \rceil$ monotone subsequences can be easily found in time $O(n^{1.5} \log n)$.

Partitions into monotone subsequences have many applications in combinatorics; one of them is an algorithm proposed by Matoušek and Welzl [MW] for counting points in a query triangle. Their algorithm has query time $O(\sqrt{n} \log n)$, space $O(n \log n)$ and preprocessing time $O(n^{1.5} \log n)$. Other applications of monotone subsequences include book embedding [CLR], data compression [AHU], pattern recognition [PB], and molecular biology [D].

We show an algorithm that preprocesses a sequence $A$ of $n$ real numbers in time $O(n \log n)$ such that finding and deleting an increasing subsequence of length $k$ can be done in time $O(n + k^2)$. As a consequence, we solve the problem of finding a maximal set of disjoint $k$-increasing subsequences of $A$ in time $O(n \log n + mk^2)$ where $m$ is the number of sequences found. We also show how to partition a sequence of numbers into $2\lceil \sqrt{n} \rceil$ monotone subsequences in time $O(n^{1.5})$.

Using our result, the $O(n^{1.5})$ preprocessing and $(\sqrt{n} \log n)$ query time of [MW] become $O(n(k + \log^2 n))$ and $O(\frac{n \log n}{k})$ respectively for any $k \leq \sqrt{n}/2$ ($k$ is a function of $n$). This improves the preprocessing of [MW] to $O(n^{1.5})$ (for the particular case of $k = \lceil \sqrt{n}/2 \rceil$), while giving their algorithm a clean capability for tradeoff between preprocessing and query. Space requirements remain unchanged.

2 Basic Concepts

We want to find a $k$-increasing subsequence of a sequence $A$ of $n$ real numbers. We will solve an equivalent geometric problem: We map each element $a_i$ to the point in the plane $p = (i, a_i)$. The problem of finding an increasing subsequence of size $k$ becomes that of finding a subset of the points that is increasing both in $x$ and in $y$. 


This brings into play the theory of minimal layers (see, for example [S]). Given two points \( p_1 = (x_1, y_1) \) and \( p_2 = (x_2, y_2) \), we say that \( p_1 \) dominates \( p_2 \) \( (p_1 \succeq p_2) \) if \( x_1 \geq x_2 \) and \( y_1 \geq y_2 \). Given a set of points \( P \), a point \( p \in P \) is minimal if there is no \( q \in P \) such that \( p \succeq q \). The minimal layer of a pointset \( P \) is the set of minimal points of \( P \). The minimal layers of \( P \) consist of the successive layers of minimal points: The first layer is the minimal layer of \( P \), the second layer is the minimal layer after the first layer has been removed, etc (see figure 1).

The following lemmas are widely known (see, e.g. [S]):

**Lemma 1** Given two adjacent layers, for any vertex \( p \) on the higher layer, there is a vertex in the lower layer dominated by \( p \).

**Lemma 2** The number of minimal layers equals the length of the longest maximal increasing subsequence.

**Lemma 3** The minimal layers can be found in time \( O(n \log n) \).

The algorithm for the last lemma is a simple plane sweep algorithm which keeps track of the current layers. When a point is reached, the layer to which it belongs is found in time \( O(\log n) \) and the layer is modified accordingly.

If \( \mathcal{L} \) is a layer structure, \( |\mathcal{L}| \) is the number of layers in it. Given the layer structure \( \mathcal{L} \) of \( A \), we can, in time \( O(n) \), find a \( k \)-increasing subsequence. We start by choosing any point \( p \) of the highest level. We sweep the next layer until we find a point dominated by \( p \), and

---

\[1\] In the case where the \( a_i \)s are all integers in the range \( 1, 2, \ldots, U \) the time complexity is reduced to \( O(n \log \log U) \). See [1].
so on. We stop when we have \( k \) points. If we reached the first layer before, then, by lemma 2 there is no \( k \)-increasing subsequence. Since we visit each point only once, the complexity of the algorithm is \( O(n) \).

3 The algorithm

We want to be able to delete a set \( D \) of points from \( \mathcal{L} \) and update \( \mathcal{L} \) efficiently. When a point \( p \in D \) is deleted, its layer is left with a hole (see figure 2). Every hole \( H \) has a left endpoint \( H_l \) and a right one \( H_r \). We say that a point \( p \) belongs to the influence zone of \( H \) if \( p \) is inside the rectangle defined by \( H_l \) and \( H_r \). A point that belongs to the influence zone of a layer \( L \) does not dominate any point on \( L \) (In figure 3 the influence zone of a hole is marked on grey).

If layer \( L_i \) has a hole \( H \) and layer \( L_{i+1} \) has points on the influence zone of \( H \), then these points should belong to \( L_i \). Furthermore, if we assume that \( L_{i+1} \) has no holes, then the points on the influence zone of \( H \) form a subchain of \( L_{i+1} \). At the end of stage \( i \) of the algorithm, all points of \( D \) which lie on layers \( 1 \ldots i \), have been deleted. and some of those layers may still have holes on them. Furthermore, any hole that can be covered, has been covered, at the expense of points in the next layer. Therefore, if layer \( L_j \), \( j < i \) has a hole, then that hole is inside a hole of layer \( L_{j+1} \). \(^2\) (Otherwise, either some points from \( L_{j+1} \) could be passed to \( L_j \) or the hole could be closed).

\(^2\)We say that a hole \( H_1 \) is inside a hole \( H_2 \) if the upright corner of the rectangle of \( H_1 \) is inside the dominance zone of \( H_2 \).
In stage i, the algorithm will use the points in layer $L_i$ to cover holes in layer $i - 1$. As a result, layer $i - 1$ now has no holes, but some holes were made to layer $i$ (figure 4). Since $L_{i-1}$ now has new points, some of them can be used to cover holes in layer $i - 2$, and so on. Only after the layers have been corrected, we will delete the points that belonged to $L_i$ (which will now belong to other layers).

At the beginning of stage $i$, we will find, for each hole $H$ in $L_j$, $j < i$, the points in $L_i$ belonging to $H$'s influence zone. Since, for each such $H$, these points form a chain, it is enough to find the first and last points of the chain. Holes that do not have points of $L_i$ in their influence zone can be closed. Note that if we close a hole, then all holes contained in it are also closed. Now, we close every hole on $L_{i-1}$ by cutting the appropriate chain from $L_i$, leaving a new hole in $L_i$. Next we cut chains from $L_{i-1}$ and transfer them to $L_{i-2}$, and so on. Since every hole $H$ of $L_j$ is inside some hole $H'$ of $L_{j+1}$, the chain that we cut from
$L_i + 1$ is a subchain of the chain that $L_{i+1}$ received. Every cut closes a hole, and either generates a new one, or enlarges an existing one.

After layer $L_i$ has been incorporated, we may have to delete some points of $D$ that once belonged to $L_i$ (and now may belong to another layer). These points generate new holes to the layer to which they now belong. However no further modifications need to be done, since the chain to which the point belongs is inside the hole on the next layer.

So we have the following algorithm:

**Procedure Delete($L, D$)**

/* $L$ is a layer structure, $D$ is a set of points. The procedure returns the layer structure obtained by deleting the points in $D$ from $L$. */

1. Let $d$ be the lowest layer containing a point in $D$.

2. For $i = d$ to $|L|$ do:
   (a) For each hole $H$, let $\text{chain}_H^i$ be the subchain of $L_i$ in the influence zone of $H$. Store a pointer to the first and last points of $\text{chain}_H^i$. If $\text{chain}_H^i$ is empty, close $H$ (concatenate the chain before $H$ to the one after $H$).
   (b) For each hole $H$, in decreasing order of layers, cut $\text{chain}_H^i$ from the layer to which it now belongs, and use it to close the hole (concatenation). See figure 4.
   (c) Delete the points of $D$ that belonged to $L_i$. Make new holes if necessary.

3. Delete from $L$ all empty layers.

In order to make the algorithm efficient, we use some simple data structures. We hold a list $\text{ListH}_L$ of the holes, ordered by the layer to which they belong. We also use a list $\text{ListH}_y$ of the left endpoints of the holes, ordered by their $y$ coordinates, and another one, $\text{ListH}_z$ of the right endpoints ordered by their $z$ coordinates.

New holes may be created only when a point is deleted or when another hole is covered, therefore the total number of holes at any time is bounded by $|D|$.

We consider now the implementation of step 2(a). Merging the $y$ coordinates of the points of $L_i$ with $\text{ListH}_y$ we can find for each hole $H$, the first point of $L_i$ that is below its left endpoint (call it $l$). Similarly we can find the last point $r$ that is to the left of the right endpoint (by merging $L_i$ with $\text{ListH}_z$). If $l$ is to the left of $r$, then the subchain in $L_i$
between \( l \) and \( r \) is chain \( \text{chain}_H \). Otherwise, there is no point of \( L_i \) in the influence zone of \( H \). In this case \( H \) can be closed. The time complexity of this implementation for layer \( L_i \) is obviously \( O(|D| + |L_i|) \). Thus the overall cost of step 2(a) is \( O(|D|(|L| - d) + n) \).

In step 2(b) we cut a chain from a layer and transfer it to another layer. This is done by making use of the pointers from \( H \) to \( \text{chain}_H \) found in step 2(a). We are changing a hole \( H \) by a hole \( H' \) in the following layer. We replace \( H \) by \( H' \) in all the linked lists. Because of simple topological properties, the order in all those lists is preserved. The number of operations per hole is constant. The overall cost of step 2(b) is thus \( O(|D|(|L| - d)) \).

In step 2(c) we have to delete some points of \( D \) from \( L \). This makes new holes, which we have to insert into the three lists. Having the \( z \) and \( y \) coordinates of the hole, it is trivial to insert it into \( \text{List}_H \) and \( \text{List}_y \) by a linear scan. Inserting the hole in \( \text{List}_H \) is a bit more complicated. We have to find to which layer \( H \) belongs now. \( H \) will be inside a hole in the layer after it. Therefore, we scan \( \text{List}_H \) until we find the first hole containing \( H \). Let \( j \) be its layer number. \( H \) belongs to \( L_{j-1} \). We now proceed as with \( z \) and \( y \). Step 2(c) takes time \( O(|D|) \) per point in \( D \). The overall cost of step 2(c) is therefore \( O(|D|^2) \).

The total cost of the algorithm is thus \( O(D^2 + |L||D| + n) \).

Then we have

**Theorem 4** Given the layer structure \( L \) of \( n \) points, \( k \) points can be deleted, and \( L \) updated in time \( O(n + k(|L| - d) + k^2) \) where \( d \) is the lowest layer containing a point to be deleted.

We will use procedure delete in order to delete \( k \)-increasing subsequences from a layer structure \( L \). For this we have

**Theorem 5** A \( k \)-increasing subsequence can be found and deleted from a layer structure in time \( O(n + k^2) \).

If the number of layers in \( L \) is lower than \( k \) then there is no \( k \)-increasing sequence. Otherwise, we will extract a \( k \)-increasing sequence with points in the last \( k \) layers. The algorithm will have only \( k \) iterations, and its running time will be \( O(k^2 + n) \).

**Theorem 6** Any sequence of \( n \) real numbers can be partitioned into \( 2\lceil \sqrt{n} \rceil \) monotone subsequences in time \( O(n^{1.5}) \).

In order to partition a sequence \( A \) into \( 2\lceil \sqrt{n} \rceil \) (or less) monotone subsequences, we will first preprocess \( A \) and organize it into a layer structure \( L \) in time \( O(n \log n) \). We will now extract from \( L \) \( \lceil \sqrt{n} \rceil \)-increasing subsequences until \( |L| < \lceil \sqrt{n} \rceil \). Obviously, we get no
more than ⌈\sqrt{n}⌉ sequences. Add to them the \(|L|\) layers, which are decreasing subsequences, to get the desired partition. By theorem 5, the time complexity required to extract the ⌈\sqrt{n}⌉-increasing sequences is \(O(n)\) per sequence. The total time is then \(O(n^{1.5})\).

### 4 Application to Counting Points in Triangles

Given a set \(S\) of points in the plane, the triangle counting problem consists in preprocessing the points in such a way that the number of points inside a query triangle can be computed efficiently.

The algorithm in [MW] has query time \(O(\sqrt{n} \log n)\), space \(O(n \log n)\) and preprocessing time \(O(n^{1.5} \log n)\). We describe now a modification to the preprocessing of their algorithm. The problem of counting points in triangles can be transformed to the following problem: Given a set \(H\) of \(n\) non-vertical lines in the plane, compute how many lines of \(H\) lie above a query point \(p\).

In the preprocessing of their algorithm two set systems \(T_L\) and \(T_R\) of \(H\), and a vertical line \(b\) are found, such that in every set of \(T_L\) no two lines intersect to the left of \(b\), and in every set of \(T_R\), no two lines intersect to the right of \(b\). Moreover, the sets in \(T_L\) (and those in \(T_R\)) are mutually disjoint. The order of intersection of lines in \(L_i \in T_L\) with \(b\) agrees with the order of their slopes. In \(R_i \in T_R\) the order of intersection with \(b\) is opposite to the order of the slopes. If we sort the lines of \(H\) by the height of their intersection with \(b\), then the sets in \(T_L\) are increasing subsequences, and those in \(T_R\) are decreasing subsequences.

A \(k\)-splitter is a triple \((T_L, T_R, b)\) such that every set in \(T_L\) and \(T_R\) has at least \(k\) elements, and such that \(|T_L| > (n - k^2)/2\); \(|T_R| > (n - k^2)/2\).

[MW] find a \(k\)-splitter in time \(O(n^{1.5} \log n)\). The bottleneck of their algorithm is repeatedly extracting \(k\)-increasing and \(k\)-decreasing subsequences. Using theorem 5, a \(k\)-splitter can be found in time \(O(n \log^2 n + nk)\).

A \(k\)-good splitter is a \(k\)-splitter \((T_L, T_R, b)\) such that

1. \(|T_L| < \frac{3n}{k}\); \(|T_R| < \frac{3n}{k}\)
2. \(||T_L|| + ||T_R|| \geq n\)

Theorem 7 A \(k\)-good splitter with \(k \leq \sqrt{n}/2\) can be found in time \(O(n(k + \log^2 n))\).

---

\(S\) is a set system; \(|S|\) is the number of sets in \(S\), \(||S||\) is the number of elements of the union of the sets in \(S\).
Find a $k$-splitter $(T_L, T_R, b)$. Let $H'$ be $H - \cup T_L - \cup T_R$. Clearly, $|H'| < k^2$. Using theorem 6, in time $O((k^2)^{1.5}) = O(k^3)$ we partition $H'$ into $2k$ monotone subsequences. Add the increasing sequences found to $T_L$ and the decreasing ones to $T_R$. Clearly $(T_L, T_R, b)$ is still a splitter, and property 2 holds, since every line in $H$ participates in some sequence. Property 1 also holds, since $|T_L| < n/k + 2k \leq 3n/k$. The overall time is $O(n(k + \log^2 n) + k^3)$. Since $k^2 < n$, the time is $O(n(k + \log^2 n))$.

By repeatedly finding $k$-good splitters, a data structure (called ES-tree in [MW]) can be found in time $O(n(k + \log^2 n))$. This structure use space $O(n \log n)$, and gives a query time $O(\frac{n}{k} \log n)$.

By using our methods with the algorithm of [MW] we get

**Theorem 8** The triangle counting problem can be solved using $O(n \log n)$ space, $O(n(k + \log^2 n))$ preprocessing time, and query time $O(\frac{n}{k} \log n)$. 


5 References


