Partitioning a Sequence into Few Monotone Subsequences

Reuven Bar Yehuda* and Sergio Fogel
Computer Science Department
Technion - IIT, Haifa 32000, Israel
reuven@cs.technion.ac.il

April 6, 1997

Abstract

In this paper we consider the problem of finding sets of long disjoint monotone subsequences of a sequence of n numbers. We give an algorithm that, after $O(n \log n)$ preprocessing time, finds and deletes an increasing subsequence of size k (if it exists) in time $O(n + k^2)$. Using this algorithm, it is possible to partition a sequence of n numbers into $2\lfloor\sqrt{n}\rfloor$ monotone subsequences in time $O(n^{1.5})$.

Our algorithm yields improvements for two applications: The first is constructing good splitters for a set of lines in the plane. Good splitters are useful for two dimensional simplex range searching. The second application is in VLSI, where we seek a partitioning of a given graph into subsets, commonly refered to as the pages of a book, where all the vertices can be placed on the spine of the book, and each subgraph is planar.

*This research was partially supported by the NY Metropolitan Research Fund

1 Introduction

Erdös and Szekeres [ES35] proved that every sequence of n numbers has a monotone (increasing or decreasing) subsequence of size $\lceil\sqrt{n}\,\rceil$. A direct consequence of this is that every sequence of n numbers can be partitioned into $2\lfloor\sqrt{n}\rfloor$ monotone subsequences. A "folklore" algorithm for finding this partition has been known for quite a while (see e.g. [MW89]), and is described in this paper. This algorithm is based on repeated iteration of finding and extracting long monotone (increasing or decreasing) subsequences. Hence each extraction of a subsequence takes $O(n\log n)$ time, and the running time of the algorithm is therefore $O(n^{1.5}\log n)$. We discuss monotone subsequences, and describe this algorithm, in Section 2.

We present in this paper a more efficient way to carry out this extraction. For example, if a subsequence is of length $O(\lceil\sqrt{n}\,\rceil)$, then extracting this subsequence takes time $O(n)$. As a result, we present an $O(n^{1.5})$-time algorithm for partitioning a sequence of n numbers into $2\lceil\sqrt{n}\,\rceil$ monotone subsequences. The algorithm uses $O(n)$ space. It is described in Section 3

Two applications of our algorithm are discussed in Section 4. The first is a variation of VLSI *book embedding* [CLR87]. Given a graph and a linear order of its vertices along the spine of a book, find an assignment of each edge to a *page* of the book in such a way, that no two edges assigned to the same page intersect, and the number of pages is small. The variation that we consider allows the pages to continue on both sides of the spine of the book, so an edge of the graph can use both sides of the page (see Figure 8). We show that every graph with m edges can be embedded in a "double paged" book having $O(\sqrt{m})$ pages. The embedding can be found in time $O(m^{1.5})$.

The second application is an efficient scheme for constructing *good splitters*. Good splitters were proposed by Matoušek and Welzl [MW89] for solving the triangle counting problem. This problem consists of preprocessing a set of n points in the plane in such a way that the number of points inside a query triangle can be computed efficiently. Using our algorithm, the preprocessing time of the algorithm of [MW89] is improved from $O(n^{1.5}\log n)$ to $O(n^{1.5})$. Further applications of good splitters are given in [BF90].

2 Monotone Subsequences

Let $Y = (y_1, y_2, \ldots, y_n), n \geq 0$ be a sequence of numbers; $Y' = (y_{i_1}, y_{i_2}, \ldots, y_{i_k}), k \geq 0$ is a subsequence of Y if $1 \leq i_1 < i_2 < \cdots < i_k \leq n$. Y' is called monotone increasing if $y_{i_1} \leq y_{i_2} \leq \cdots \leq y_{i_k}$ and monotone decreasing if $y_{i_1} \geq y_{i_2} \geq \cdots \geq y_{i_k}$.

Suppose we want to find increasing subsequences of Y. We can replace every number in Y with its index in the sorted sequence. In this way increasing subsequences are preserved. A stable sorting algorithm[1] has to be used in order to preserve increasing subsequences. If we want decreasing subsequences we replace every number by its index in the sequence sorted in decreasing order. The sorting is done once and costs $O(n \log n)$ time. Therefore, in the rest of the paper we assume that Y is a permutation of $(1, 2, \ldots, n)$.

Looking at the problem from a geometric point of view turns out to be helpful for the intuitive understanding of the algorithms and proofs. We map every number y_i to the point in the plane $p_i = (i, y_i)$. Since Y is a permutation, no two points have the same x or y coordinates. An increasing subsequence is mapped to a set of points such that the sorting of the points on their x coordinates agrees with the sorting on their y coordinates. Such a set of points is called an increasing subset of the set $P = \{p_1, p_2, \ldots, p_n\}$. All the proofs in the sequel deal with points rather than with sequences, but they can be trivially translated to sequences.

2.1 Maximal Layers

Let P be a finite set of points in the plane, no two of them having the same x or y coordinates. We denote by $p.x$ and $p.y$ the x and y coordinates of a point p, respectively. Given two points p_1 and p_2 we say that p_1 *dominates* p_2 and denote $p_1 \succ p_2$, if $p_1.x > p_2.x$ and $p_1.y > p_2.y$. We say that a subset $S = \{s_1, s_2, \ldots, s_k\}$ of P is monotone increasing (or just increasing) if there exists some permutation $(i_1, i_2, \ldots, i_k)$ of $(1, 2, \ldots k)$ such that $s_{i_1} \succ s_{i_2} \succ \ldots \succ s_{i_k}$ (equivalently, the sorting of the points by x coordinate is the same as by y coordinate).

A point $p \in P$ is said to be *maximal* in P if there is no point $q \in P$ such that $q \succ p$.

[1] A sorting algorithm is called stable if it keeps the index of the occurrence between equal elements.

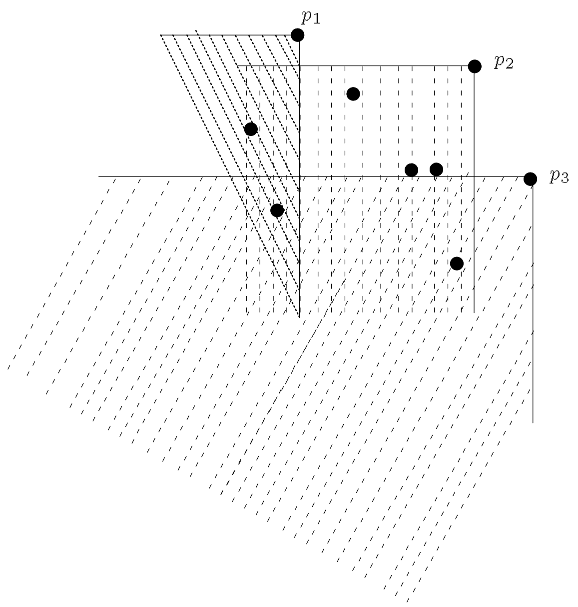

Figure 1: A set of points, where $M(P) = \{p_1, p_2, p_3\}$.

Let $M(P)$ be the set of maximal points of P. Maximal points play a crucial part in finding monotone subsequences. Let us investigate the structure of $M(P)$. For a point $p \in \mathbb{R}^2$ define

$$LQ(p) = \{q \in \mathbb{R}^2 \ : \ q.x \leq p.x \text{ and } q.y \leq p.y\}$$

Let $M = M(P)$ and let $\mathcal{R} = \cup_{p \in M} LQ(p)$. It is easily observed that each point of $P \setminus M(P)$ is in the (interior of) $\mathcal{R}$. Setting the *geometric realization* of M, denoted as $GR(M)$, to be the polygonal path which is the boundary of $\mathcal{R}$, this means that every point of $P \setminus M(P)$ is strictly below $GR(M)$. See Figure 1. Note that $GR(M)$ is an x-monotone path and a y-monotone path. That is, each line parallel to either the x or the y axis intersects $GR(M)$ at most once.

The *layer structure* of P, $\mathcal{L}(P)$, is the sequence

$$\mathcal{L}(P) = (L_1(P), L_2(P), \ldots, L_l(P))$$

where the i-th layer $L_i(P)$ is defined recursively as follows:

$$L_i(P) = M\left(P - \bigcup_{1 \leq j < i} L_j(P)\right)$$

If $P = \emptyset$, $M(P) = \emptyset$ and we get the empty layer structure $\mathcal{L}(P) = ()$. Otherwise $P \neq \emptyset$; then let $L_l(P)$ be the last non empty layer. Clearly, $0 \leq l \leq |P|$, and $L_i(P) \neq \emptyset$ for all

$1 \leq i \leq l$. Observe that $GR(M(P_i))$ is strictly above $GR(M(P_j))$ if $i < j$. We impose on each $L_i(P)$ the order defined by the increasing x coordinates of its points. It is easy to check that this is the same as the order by decreasing y coordinates. We say that $L_i(P)$ is *higher* than $L_j(P)$ if $i < j$. A *j-subsequence* of a point-set P is a subsequence of P whose length is j. The following observations will turn out to be very useful.

Lemma 1 **(i)** *For every point $p \in L_i(P)$, $i > 1$, there exists a point $q \in L_{i-1}(P)$* dominating p.

(ii) *A point $p \in P$ does not dominate any point in a higher or equal layer.*

(iii) *Let $p \in P$. If $p \in L_j(P)$ then there exists an increasing subset $\{p = p_j, p_{j-1}, \ldots, p_1\}$, such that $p_i \in L_i$, and p_i dominates p_{i+1}, (for $1 \leq i \leq j-1$). No increasing subsequence of length $j+1$ starting at p exists in P.*

(iv) *Let $p \in P$. Assume there is an increasing subsequence of points of P, starting at p, of length j, but no such subsequence of length $j+1$ exists. Then $p \in L_j$.*

(v) *The length of every increasing subsequence in P is at most $|\mathcal{L}(P)|$.*

Proof: To show (i), observe that if no such q exists, we could have added p to L_{i-1} (or an higher layer). (ii) (iii) and (iv) are immediate. To show (v) observe that if an increasing subsequence of size larger than $|\mathcal{L}(P)|$ exists, then two points of this subsequence must belong to the same layer, which contradicts (iv) □

We now describe an algorithm for constructing $\mathcal{L}(P)$. We shall later use $\mathcal{L}(P)$ for finding a partition of P into a small number of monotone subsequences.

For all the algorithms presented in this paper, a layer structure is stored as an array of doubly-connected linked lists of points, one list for each layer. In order to build the layer structure $\mathcal{L}(P)$, a plane sweeping technique, essentially equivalent to the algorithm of Fredman [Fre75] or to that of Dijkstra [Dij80], is used. The idea is to sweep the plane with a vertical line from right to left, and every time a point p is met, to determine the layer to which the point belongs. If no such layer exists, the algorithm creates a new layer. To find the layer to which p is added, the algorithm stores the y coordinate of the last point seen, of each layer $L_j(P)$, as the j-th element of an array A (Figure 2). The points of A form a current cut of the (geometric realization of the)

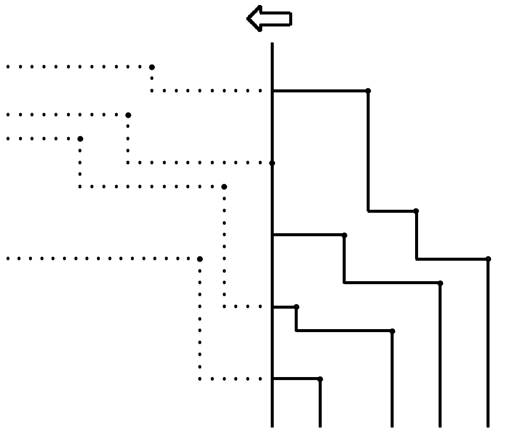

Figure 2: Sweeping the points in order to find the layers.

layers. Note that when considering the geometric realization of the layers constructed so far, p must be added to the highest layer below p (if exists).

Algorithm *Layers*
Input: $P = (p_1, p_2, \ldots, p_n)$ such that $p_1.x < p_2.x < \cdots < p_n.x$ and with pairwise different y coordinates.
Output: $\mathcal{L}' = (L'_1, L'_2, \ldots, L'_l) = \mathcal{L}(P)$.
Data Structure: A, an array of $|P| + 1$ real numbers.

$A[0] \leftarrow +\infty$ /* To simplify the loop */

$l \leftarrow 0$

For $i = n$ downto 1 do

Use binary search in $A[1..l]$ to find $j \leq l$, the largest index such that $A[j] > p_i.y$

/* L_j is the lowest level above p */

If $j = l$, let L'_{j+1} be a new, empty layer, and set $l \leftarrow l + 1$.

Add p_i to the front of L'_{j+1}.

$A[j+1] \leftarrow p_i.y$.

/* Invariant: $\mathcal{L}' = (L'_1, L'_2, \ldots, L'_l)$ is the layer structure of the points seen so far */

Lemma 2 *The following is invariant at the end of every i-th iteration of* **Algorithm** *Layers:* $\mathcal{L}' = \mathcal{L}(p_i, p_{i+1}, \ldots, p_n)$.

Proof: We shall prove the lemma by induction on $k = n - i$. For $k = 1$ the claim is trivially true. Assume that after $k-1$ iterations $\mathcal{L}' = \mathcal{L}(p_{i+1}, p_{i+2}, \ldots)$.

Since p_i is to the left of $p_{i+1}, \ldots, p_n$, it does not dominate any point already in $\mathcal{L}'$, and hence these points are in their correct layers. It remains to be shown that p_i is assigned to the proper layer. To this end we prove that $p_i \in M(\{p_i\} \cup \bigcup_{m \geq j+1} L'_m)$, and that either $j = 0$ or $p_i \notin M(\{p_i\} \cup \bigcup_{m \geq j} L'_m)$. According to the definition of j, either $j = l$ (in which case $\bigcup_{m \geq j+1} L'_m = \emptyset$), or the y coordinate of p_i is greater than $A[j+1]$ and therefore it is also greater than the y coordinate of all points in $\bigcup_{m \geq j+1} L'_m$, and p_i is maximal in $\{p_i\} \cup \bigcup_{m \geq j+1} L'_m$.

If $j = 0$ the second part of the proof is trivial. Otherwise, let q be the point whose y coordinate was last assigned to $A[j]$. Clearly, $q \in L_j(P)$ and $q \succ p_1$. $\square$

Corollary 3 *Algorithm Layers computes the layer structure of a set of points in $O(n \log n)$ time.*

2.2 Finding a Longest Increasing Subsequence

Given a set of points P under the same assumptions as in section 2.1, $\mathcal{L}(P)$, the layer structure of P, and an integer $k \leq |\mathcal{L}(P)|$, an k-increasing subsequence of P can be found with the following algorithm. Usually, k will be equal to $|\mathcal{L}(P)|$, and is, by Lemma 1, the longest increasing subsequence.

Algorithm *LIS*
Input: $\mathcal{L}(P)$, a positive integer $k \leq |\mathcal{L}(P)|$.
Output: An increasing subset $S = \{q_1, q_2, \ldots, q_k\} \subseteq P$.

1. If $k = 0$, return the empty sequance.
2. $l \leftarrow k$; $q_l \leftarrow$ a point in $L_l(P)$.
3. For $i = l - 1$ downto 1 do
   find q_i, a point in $L_i(P)$ dominating q_{i+1}.

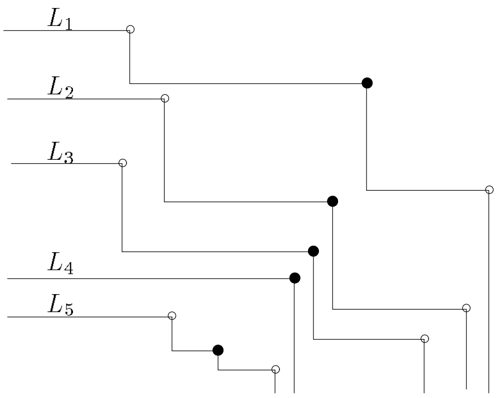

Figure 3: 5 layers yielding a monotone subsequences of length 5

Lemma 4 *Algorithm LIS finds an increasing subsequence of size k of P. The algorithm can be implemented in $O(n)$ time and space.*

Proof: By Lemma 1, in step 3 of the algorithm, a point complying with the requirement can always be found in L_i in time $O(|L_i(P)|)$, which sums to time $O(n)$. □

3 The Algorithm

We show in Section 3.1 that once $\mathcal{L}(P)$ is constructed, and a monotone subsequence D is extracted from P, the layer structure of $P - D$ can be constructed efficiently. In Section 3.2 we describe an $O(n^{1.5})$ time algorithm that uses this technique to partition P into monotone subsequences.

3.1 Finding Many Increasing Subsequences

We have shown that $\mathcal{L}(P)$ can be built in time $O(n \log n)$, and once $\mathcal{L}(P)$ is found, a longest increasing subset can be found in $O(n)$ time. Recall that our goal is to divide P into (disjoint) monotone subsequences, hence after a longest increasing subsequence is found, we would like to repeat finding longest increasing subsequences in the remaining set. A direct approach for doing this is finding a largest increasing subset $S_1 \subseteq P$, then finding a largest increasing subset $S_2 \subseteq P - S_1$, and so on. This clearly yields an $O(mn \log n)$ time algorithm, where m is the number of iterations. As in [KO88], we can reduce the time to $O(mn \log\log n + n \log n)$ by first sorting the points, and then working on points with coordinates in $\{1, 2, \ldots, n\}$. This allows us to use a Johnson tree [Joh82] for implementing the searches in $O(\log\log n)$ time (a Johnson tree stores a set of numbers in the range $1, \ldots, U$ and allows finding a key in time $O(\log\log U)$).

The dominant part of the time complexity is repeatedly building the layer structures. Our aim is, given $\mathcal{L}(P)$ and a set $D \subseteq P$, to compute $\mathcal{L}(P - D)$ efficiently. We could restrict ourselves to the case where every point in D belongs to a different layer, but we will need general deletions in the future. We first give a rough algorithm, which is simple to prove, and then refine it in order to get the desired complexity. We need some more definitions at this time.

A *partial layer structure* $\tilde{\mathcal{L}} = (\tilde{L}_1, \tilde{L}_2, \ldots, \tilde{L}_l)$ is a sequence of nonempty sets, called *layers*, such that for every $1 \leq i \leq l$:

$$\tilde{L}_i \subseteq M\left(P - \bigcup_{1 \leq j < i} \tilde{L}_j\right).$$

As with layer structures, we impose on each layer the order of the increasing x coordinates. As is easily shown by induction on the layers, when some points are deleted from a layer structure and the empty layers are removed, the result is a partial layer structure of the remaining points.

In order to avoid dealing with extreme cases, let us add to the beginning of each layer a dummy point with coordinates $(-\infty, +\infty)$, and at the end a dummy point with coordinates $(+\infty, -\infty)$. As a result of these dummy points, No layer $\tilde{L}_i$ is ever empty, and hence $\tilde{L}_i \subseteq L_i$, for $i = 1 \ldots l$.

For a point $p \in \tilde{L}_i$, define next(p) to be the point following p it $\tilde{L}_i$. $\boxed{p}$ is the (open) rectangle defined by p and next(p). p is the left corner of $\boxed{p}$ and $next(p)$ is the right corner of $\boxed{p}$. We say that $\boxed{p}$ is a *hole* if there is a point $p' \in \tilde{L}_j$, (for $j > i$) inside $\boxed{p}$. In this case, we say that p' is a *witness* to $\boxed{p}$ (see Figure 4). We say in this case that the hole $\boxed{p}$ is *contained* in $\tilde{L}_j$. It is easy to check that if p' is a witness to $\boxed{p}$, a point

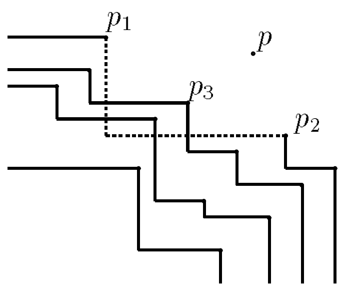

Figure 4: A hole caused by the deletion of p

is missing between them in $\tilde{L}_i$, since no point in that layer dominates p'. This in turns also implies that $\tilde{\mathcal{L}} \neq \mathcal{L}(P)$.

Let $\tilde{\mathcal{L}}_{\leq k}$ denote the upper k layers of $\tilde{\mathcal{L}}$.

Lemma 5 *Let $\tilde{\mathcal{L}}$ be a partial layer structure of P. Then $\tilde{\mathcal{L}} = \mathcal{L}(P)$ iff $\tilde{\mathcal{L}}$ has no holes.*

Proof: One direction has just been proved. Let us show that if $\tilde{\mathcal{L}} \neq \mathcal{L}$ then there must be a hole in $\tilde{\mathcal{L}}$.

Let i be the smallest index for which $L_i(P) \neq \tilde{L}_i$. By the definition of partial layer structure, $\tilde{L}_i \subseteq L_i(P)$. Let $p_3 \in L_i(P) - \tilde{L}_i$, let p_1 be the rightmost point of $\tilde{L}_i$ to the left of p_3, and p_2 the leftmost point of $\tilde{L}_i$ to the right of p_3 (p_1 and p_2 may be dummy points). Clearly p_3 is in a layer of $\tilde{\mathcal{L}}$ larger than $\tilde{L}_i$, and therefore p_1 and p_2 form a hole in $\tilde{\mathcal{L}}$. □

Let $H(\tilde{\mathcal{L}})$ denote the set of all the left upper endpoints of the holes of $\tilde{\mathcal{L}}$. We use superscripts in order to denote the iteration of the main loop (line 3). For instance, $\tilde{L}_j^t$ is $\tilde{L}_j$ after t iterations of the main loop. The algorithm will scan the layers, from highest to lowest, and every time a point p is found inside a hole, it will be transferred to the lowest possible layer containing a hole for which p is a witness.

Algorithm A
Input: $\mathcal{L}(P)$, $D \subseteq P$.
Output: $\tilde{\mathcal{L}} = (\tilde{L}_1, \ldots, \tilde{L}_{l'}) = \mathcal{L}(P - D)$.

1. $\tilde{\mathcal{L}} = (\tilde{L}_1, \ldots, \tilde{L}_l) \leftarrow \mathcal{L}(P)$.
2. $H^0 \leftarrow \emptyset$. /* H^i is the set of all top left endpoints of the holes in $\tilde{\mathcal{L}}^i$ */
3. For $i = 1$ to $|\mathcal{L}(P)|$
   (a) For every $p \in L_i(P)$ do
      If there is a hole whose top left endpoint is in H^{i-1} for which p is a witness:
      i. Find $\tilde{L}_j$ — the highest layer containing a hole in H^{i-1} for which p is a witness.
      ii. Transfer p from $\tilde{L}_i$ to $\tilde{L}_j$
   (b) Delete from $\tilde{\mathcal{L}}$ the point(s) of $L_i(P) \cap D$.
   (c) $H^i \leftarrow H(\tilde{\mathcal{L}})$.

Observation 1 *If a point p was transferred (in stage 3(a)) from $\tilde{L}_i$ to $\tilde{L}_j$, (for $j \geq 2$), then some point in $q \in \tilde{L}_{j-1}$ dominates p.*

Proof: Consider q, the first point in $\tilde{L}_{j-1}$ to the right of p. This point must be higher than p — otherwise it would be the right-hand point of a hole, to which p must be a witness. If this were the case, then p would have been transferred to $\tilde{L}_{j-1}$ or a higher layer. Hence q dominates p. $\square$

We denote by P_i the set $\bigcup_{1 \leq j \leq i} L_j(P)$.

Lemma 6 *After t iterations, $\forall j \leq t$, $\tilde{L}_j^t = L_j(P_t - D)$.*

Remark: The lemma implies that after t iterations, the first t layers of $\tilde{\mathcal{L}}$ are identical to the first t layers of $\mathcal{L}(P_t - D)$. This implies that in the remainder of the course of the algorithm, no point of these layers will be transferred, though points from lower layers may be transferred into these layers. Moreover, it implies that no point in $\tilde{\mathcal{L}}_{\leq t}^t$ is a witness to a hole in $\tilde{\mathcal{L}}_{\leq t}^t$, which by Observation 1, implies that each point in $\tilde{L}_j^t$ ($j \leq t$) is dominated by a point in $\tilde{L}_{j-1}^t$.

Proof: (of Lemma 6). We want to prove by induction that after t iterations, all points in P_t (points that have already been treated) have been assigned to their proper layers with respect to $P_t - D$ (and therefore with respect to $P - D$).

Basis: $t = 1$. Trivially true.

Step: Assuming correctness for $t-1$, the points in P_{t-1} are already in their final layers, and they will not be transfered. It is sufficient to prove that all points in $P_t - P_{t-1}$ are transferred (in step 3(a)) to the correct layer. Let p_3 be a point in $P_t - P_{t-1}$, and let $\tilde{L}_j$ be the layer to which p_3 belongs at the end of the t-th iteration. The algorithm guarantees that there exists a point q in $\tilde{L}_{j-1}$ that dominates p_3, and from Lemma 1 (ii), we know that q cannot be in $P_t - P_{t-1}$. By the induction hypothesis, q is in its final layer, and p_3 has to be in a layer larger than $j - 1$.

Now it is sufficient to prove the following claim:

$$p_3 \text{ is maximal in } P - D - \tilde{L}_1 - \cdots - \tilde{L}_{j-1}.$$

If there is no hole in H^{i-1} containing p_3, then p_3 has remained in layer $\tilde{L}_t$ (which means $j = t$). In this case the claim is clearly true, since $P - D - \tilde{L}_1 - \cdots - \tilde{L}_{j-1} = P - D - L_1(P) - \cdots - L_{j-1}(P)$ (induction step).

Otherwise (if $j < t$) there is a hole in layer j containing p_3. Assume that there is a point $q \succ p_3$ in a layer greater or equal to $\tilde{L}_j$. In this case, from geometric considerations, q must be inside the same hole as p_3. Thus, q is not in the proper layer. From the induction hypothesis, q cannot be in P_{t-1}. But this contradicts the definition of $\mathcal{L}(P)$. $\square$

Before we proceed to refine the algorithm, let us give some useful observations.

Observation 2 *Let $p_1 \in \tilde{L}_{j+1}^{t-1}$, and $p \in \tilde{L}_{t'}^{t-1}$, for $j < t \leq t'$. Then p_1 is not dominated by p.*

Proof: Obvious, since in $\mathcal{L}$, p_1 belongs to layer higher or equal to the one containing p, and none of the operations performed by the algorithm (until time "t") can change this relation. $\square$

Observation 3 *A point is a witness only to one hole of each layer. Moreover, during Algorithm A , if a point $p \in L_t(P)$ is a witness to a hole $\boxed{w}$ of $\tilde{L}_j^{t-1}$, $1 \leq j \leq t-2$, then it is also a witness to a hole of its successive layer $\tilde{L}_{j+1}^{t-1}$.*

Proof: The first part is immediate. To see the second, consider Figure 5. Let p_1 and $p_2 = next(p_1)$ be the points of $\tilde{L}_j^{t-1}$ determining a hole for which p is a witness. (i.e. $p \in \boxed{p_1}$). Let q_1 be the first point to the left of p, and let $q_2 = next(q_1)$. Since by Observation 2, p does not dominate q_1, then q_1 must be higher than p. On the other hand, $q_2 \notin \boxed{p_1}$, and is lower than p_2, and therefore is also lower than p. Thus $q_2.x > p.x$, and hence $p \in \boxed{q_1}$. $\square$

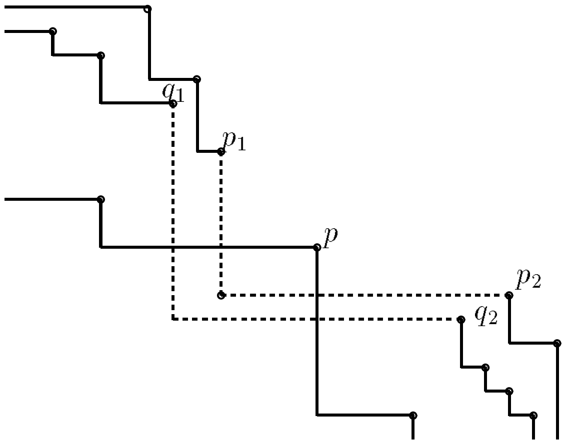

Figure 5:

Observation 4 *Let* $\boxed{w} \in \tilde{L}_j$, $(j \leq t)$ *be a hole,* $p \in L_{r+1}$ *(for* $r > t$*) be a witness to* $\boxed{w}$*, and assume* $p' \in L_r$ *dominates* p*. Then* p' *is also a witness to* $\boxed{w}$*.*

Proof: Let h and v be the lines containing the lower horizontal and the left-hand vertical edges of $\boxed{w}$. Since $p \in \boxed{w}$, p' must be above h and to the right of v. Surely, p' is lower than w. However, if p' is to the right of $next(w)$, then it must also be above $next(w)$, which contradicts the definitions of the layers. $\square$

Observation 5 *For every hole in* $\tilde{\mathcal{L}}^{t-1}$ *there is a witness in* $L_t(P)$*.*

Proof: It follows from the definition of $\tilde{\mathcal{L}}^0$ that there are no holes in layers below $\tilde{L}^{t-1}_{t-1}$. Consider $\tilde{\mathcal{L}}^{t-1}_{\leq t-1}$. By the assumption, no such hole has a witness in one of these layers. Hence if it has a witness in $\tilde{L}_r$, $r \geq t$, then by Observation 4, it also has a witness in $\tilde{L}_t$. $\square$

Observation 6 *Let* p_1 *and* $p_2 = next(p_1)$ *be two points that form a hole in* $\tilde{L}^{t-1}_j$ *while* q_1 *and* $q_2 = next(q_1)$ *form a hole in* $\tilde{L}^{t-1}_{j+1}$*, (for* $1 \leq j \leq t-2$*). Assume furthermore that* $p \in L_t(P)$ *is a witness to both holes* $\boxed{p_1}$ *and* $\boxed{q_1}$ *(that is,* $p \in \boxed{p_1} \cap \boxed{q_1}$*). Then the following relations, demonstrated in Figure 5 hold:*

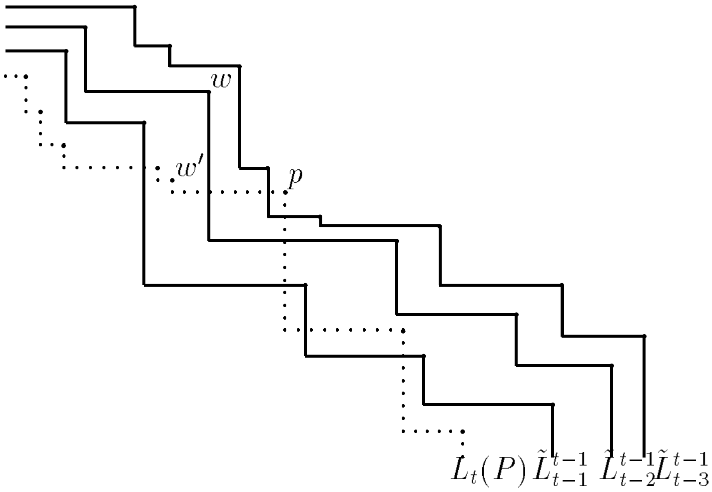

Figure 6: The situation when $L_t(P)$ is about to be added.

$$q_1.x < p_1.x < p.x < q_2.x \quad \textit{and} \quad q_2.y < p_2.y < p.y < q_1.y$$

Proof: Obvious.

Observation 7 *If a point $p \in P_t$ is not in $H(\tilde{\mathcal{L}}^t)$, it is not in $H(\tilde{\mathcal{L}}^{t+1})$ either.*

Proof: No point of P_t is moved from its place after the the t'th phase. Note, however, that a point p of $L_{t'}$ (for $t' > t$) might be transferred into $\tilde{\mathcal{L}}_{\leq t}$, and might then become the corner of new holes, but this does not affect the validity of the lemma, since $p \notin L_t$. $\square$

We are now ready for the first refinement of the algorithm. In algorithm A we use a set H^i in order to find the deepest hole that contains a point p. In algorithm B the set H^i of left endpoints of holes is substituted by a set W^i of left endpoints of *windows*, which will play a similar role; Each window is a hole, but not the other way around. Since the time complexity of the algorithm depends on the number of windows, we will insure that $|W^i|$ remains small. In particular, from each subset of windows "nested" in each other (such as in Figure 5), we maintain only the upper one, since this is the one that must be filled when it is found to be a hole. As in Algorithm A , in Algorithm B we scan the layers from highest to lowest, each from left to right. Dealing with a point p of layer $L_i(P)$, (in 3.(a)) we check if it is in some window of an upper layer, and if so, transfer it to the highest such layer.

Algorithm B
Input: $\mathcal{L}(P)$, $D \subseteq P$.
Output: $\tilde{\mathcal{L}} = (\tilde{L}_1, \ldots, \tilde{L}_{l'}) = \mathcal{L}(P - D)$.

1. $\tilde{\mathcal{L}} = (\tilde{L}_1, \ldots, \tilde{L}_l) \leftarrow \mathcal{L}(P)$
2. $W^0 \leftarrow \emptyset$ /* W^t are the left endpoints of the windows after t iterations */.
3. For $i = 1$ to $|\mathcal{L}(P)|$ /* See figure 6 */
   (a) For every $p \in L_i(P)$ do
   If $\exists w \in W^{i-1} : p \in \boxed{w}$: /* Is this window really a hole */
      i. $j \leftarrow \min\{k : \exists w' \in W^{i-1} \cap \tilde{L}_k,\ p \in \boxed{w'}\}$
      ii. Transfer p from $\tilde{L}_i$ to $\tilde{L}_j$.
   (b) $W^i \leftarrow \emptyset$
   For every $w \in W^{i-1}$ do
      Let $\tilde{L}_j$ be the layer of w.
      $w' \leftarrow$ Rightmost point of $\tilde{L}_{j+1}$ to the left of w. /* See explanations below */
      If $\boxed{w'} \cap L_i(P) \neq \emptyset$ then $W^i \leftarrow W^i \cup \{w'\}$.
   (c) Delete from $\tilde{\mathcal{L}}$ the points of $L_i(P) \cap D$.
   For each point deleted, let w' be its predecessor in $\tilde{\mathcal{L}}$. Insert w' to W^i.

While 3(a) and 3(c) are rather straightforward 3(b) is less natural. Assume we are at the start of 3(b) of the ith stage. Let us give some intuition for that operation. Observe first that (the left corner of) each window always belongs to a layer higher than L_i. In 3(a) we transfer from $\tilde{L}_i$ to (the proper) $\tilde{L}_j$ a point p, in order to "fill" the hole in $\tilde{L}_j$. This might cause $\boxed{p}$ to become a new hole. If this is the case, then there must be a point q of a layer *below* $\tilde{L}_j$, such that $\boxed{q}$ is also a hole. namely $\tilde{L}_{j+1}$. As can easily be seen, q must be the rightmost point of $\tilde{L}_j$, to the left of the "old" hole.

Lemma 7 *The following are invariants after each iteration of Algorithm B.*

HOLES(t):	$H(\tilde{\mathcal{L}}^t) \subseteq W^t$	*All holes are represented in W*
LAYERS(t):	$\forall i \leq t, \tilde{L}_i^t = L_i(P_t - D)$	*All treated points are in their layers.*

Proof: We will prove invariants HOLES(t) and LAYERS(t) simultaneously by induction on t according to the following scheme:
1. LAYERS(0)
2. HOLES(0)
3. HOLES(0),...,HOLES($t-1$) $\Rightarrow$ LAYERS(t)
4. HOLES($t-1$) , LAYERS(t) $\Rightarrow$ HOLES(t)

1. LAYERS(0) is vacuously true.

2. HOLES(0) is true since $\tilde{\mathcal{L}}^0 = \mathcal{L}(P)$, and therefore $\tilde{\mathcal{L}}^0$ has no holes.

3. The validity of HOLES(0),...,HOLES($t-1$) means that during the first t steps of Algorithm B, all holes are represented in W. Since we do nothing with windows that are not holes, the first t steps of Algorithm B make exactly the same changes in $\tilde{\mathcal{L}}$ as the first t steps of Algorithm A, and therefore Lemma 6 applies here as well. This implies that LAYERS(t) is true.

4. Given HOLES($t-1$), LAYERS(t) and $w' \in H(\tilde{\mathcal{L}}^t)$, we need to prove $w' \in W^t$.

   Holes may be created by the deletion of a point of D. If w' is such a hole, then w' entered W^t when the point was deleted, in step 3(c) of the algorithm.

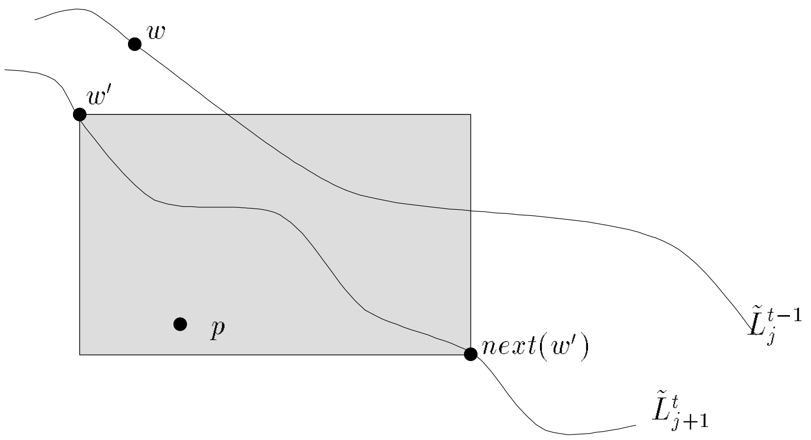

Figure 7:

Otherwise (the hole was created by the transfer of a point to a higher layer), let $\tilde{L}_{j+1}^t$ be the layer of w'. We want to show that (see step 3(b))

(1) $\exists w \in \tilde{L}_j^{t-1}$ **s.t.** $w \in W^{t-1}$,
and w' is the rightmost point of $\tilde{L}_{j+1}$ to the left of w

(2) $\boxed{w'} \cap L_t(P) \neq \emptyset$.

To prove (2) consider two cases. If $w' \notin L_t(P)$ then (2) follows from observation 5. Else, the assumptions that $w' \in L_t(P)$ and w' is a hole implies that $\text{next}^0(w') \neq \text{next}^t(w')$ and therefore $\text{next}^0(w') \in \boxed{w'}$. Surely $\text{next}^0(w')$ belong to $L_t(P)$.

Let us now turn to prove (1). Let p be the leftmost witness to $\boxed{w'}$ in $\boxed{w'} \cap L_t(P)$, which, by (2) is not empty. Let $w \in \tilde{L}_j^{t-1}$ be the rightmost point to the left of p (see Figure 7). In the first t phases of the algorithm, p was transfered to the highest layer $\tilde{L}_j$ that contained a hole containing p. Since $\boxed{w'}$ is currently a hole (by our assumption), this layer is not $\tilde{L}_{j+1}$. On the other hand, by Observation 7, $\boxed{w'}$ was also a hole at the end of the $(t-1)$'th iteration. Hence $\tilde{L}_i$ must be a layer higher than $\tilde{L}_{j+1}$. Then by Observation 7, p is also a witness to a hole $\bar{w} \in \tilde{L}_j$, yielding by Observation 6, that $\bar{w}$ is merely w, hence $w \in$HOLES$(t-1)$. $\square$

Now we give the final refinement. As before, we will treat the points layer by layer. For each layer, we will sweep the plane with a vertical line from left to right, and take some action every time an event occurs. An event is either meeting a point of $L_t(P)$ (a *Layer Event*) or a point in W^{t-1} (a *Window Event*). We also use two stacks. The stack *stackP* contains points of P to the left to the left of the sweeping line, which are suspected to be (leftmost points of) holes containing the last seen point p, as their witness. The bottom of the stack always contains the previous points of the last point seen in the current layer. The elements in the stack are sorted from left to right. At the top of the stack we maintain the point which would precede p in the layer to which p would be transferred. Especially, if there are no holes present, then $top(stackP)$ always contains $prev(p)$.

The stack *stackW* is almost identical to *stackP*, except for some technical details to be described later. Every time a point p of $L_t(P)$ is met, all the windows that contain p as a witness are inside the stacks. The stacks allow us to find, in constant time, into which layer to transfer p. Every time we meet a window we can determine in constant time using *stackP* which point to insert into W^t.

Algorithm C
Input: $\mathcal{L}(P)$, $D \subseteq P$
Output: $\tilde{\mathcal{L}} = (\tilde{L}_1, \ldots, \tilde{L}_{l'}) = \mathcal{L}(P - D)$

1. $\tilde{\mathcal{L}} = (\tilde{L}_1, \ldots, \tilde{L}_l) \leftarrow \mathcal{L}(P)$
2. $W^0 \leftarrow \emptyset$
3. For $i = 1$ to $|\mathcal{L}(P)|$ do
   (*) $W^i \leftarrow \emptyset$ /* W is a linked list holding the left endpoints of all the windows sorted by their x coordinate.*/
   $stackW \leftarrow \emptyset$ /* $stackW$ holds the windows containing the next Layer event */
   $stackP \leftarrow$ dummy point at the start of $L_t(P)$. /*$stackP$ holds the last point met in each layer */
   (a) While there is an event:
   Layer Event p
   $q \leftarrow pop(stackP)$.
   Push p into $stackP$.
   Transfer p to the layer of q, between q and next(q) in $\tilde{\mathcal{L}}$, if it is not there already.
   Pop from $stackW$ every w such that next$^0(p) \notin \boxed{w}$;
   Pop the same number of times from $stackP$
   Window Event w
   Let p be the next Layer Event.
   If $p \in \boxed{w}$ then:
   $w' \leftarrow top(stackP)$.
   Insert w' into W^i (only if it is not already there).
   Push w into $stackW$ and into $stackP$.
   (b) Delete from $\tilde{\mathcal{L}}$ the points of $L_i(P) \cap D$.
   For each point deleted, let w' be its predecessor in $\tilde{L}$. Insert w' into W^i.

Lemma 8 *Algorithm C is equivalent to Algorithm B .*

Proof: It is easy to see that every time a Layer Event p is found, all the windows of W containing p as a witness are in $stackW$ (this follows from observation 3). Moreover, the order of the windows in $stackW$ is the order of the x coordinates of their left endpoints (which is also the order of the layers, by observation 4). This implies that the top of the stack contains the "deepest" hole that contains p, which is exactly what is found in step 3(a)i of Algorithm B . Therefore, the transfer step is the same in both algorithms.

Consider now a Window Event w, and let p be the next Layer Event. If $p \notin \boxed{w}$ then w is not in $H(\tilde{\mathcal{L}}^t)$ and therefore can be ignored. Otherwise ($p \notin \boxed{w}$), let w' be the top of $stackP$. The layer of w' is obviously one larger than the layer of w, and w' is the rightmost point of that layer that is to the left of w. Therefore, w' is in fact the same as w' of step 3(b) in Algorithm B . The deletion of $L_t(P) \cap D$ is done exactly in the same way in both algorithms, thus W^t is equivalent in both algorithms.

It is easy to check that the x order of each of the layers and of W^t is also preserved.
□

Lemma 9 *Algorithm C runs in time $O(n + |\mathcal{L}(P)||D|)$ and uses $O(n)$ space.*

Proof: By the way W^t is built, $|W^t| \leq |W^{t-1}| + |L_t(P) \cap D|$, and therefore, for all t, $|W^t| \leq |D|$.

In each iteration of the algorithm, in order to find the next event we have to merge the x coordinates of W^{t-1} with those of $L_t(P)$. The merging takes time $O(|L_t(P)| + |W^{t-1}|)$. Summing over t we get the claimed time complexity. In order to get $O(n)$ space, we just have to make sure to discard each stack after it is used (and free its space). □

Lemma 10 *Let S be a sequence of n numbers. We can preprocess S in time $O(n \log n)$, so that an increasing subsequence in S of length k can be found and deleted from S in time $O(n + k^2)$ (or determine that no such subsequence exists).*

Proof: Transform the sequence into points, and build the layer structure of the points. In order to find an increasing subsequence, use points in the highest k layers of $\mathcal{L}(P)$. It is clear that Algorithm C will have only k iterations. In this way building $\mathcal{L}(P - D)$ from $\mathcal{L}(P)$ takes time $O(n + k^2)$. □

3.2 Partitioning a Sequence Into $2\lfloor\sqrt{n}\rfloor$ Monotone Subsequences

Theorem 11 *A sequence of n numbers can be partitioned into $2\lfloor\sqrt{n}\rfloor$ monotone subsequences in time $O(n^{1.5})$. All the subsequences can be chosen to be of size $\lceil\sqrt{n}\,\rceil$ or less.*

Proof: First we apply Lemma 10, and repeatedly find and delete increasing subsequences of size $k = \lceil\sqrt{n}\rceil$ until this is no longer possible. Then we again apply Lemma 10 with the same k on the residual sequence but this time for finding decreasing subsequences. Clearly, this took us time $O(n^{1.5} + n\log n) = O(n^{1.5})$, since $|D| = \lceil\sqrt{n}\,\rceil$. We now build two layer structures, one for increasing subsequences and one for decreasing ones. Since among the remaining points there are no monotone subsequences of size $\lceil\sqrt{n}\,\rceil$, the number of layers in each of the structures is at most $\lfloor\sqrt{n}\rfloor$.

All that we have to do now is to report a longest increasing or decreasing subsequence (the one that is longer), and delete it from both structures. Since the number of layers in both structures is not more than k, each deletion will take time $O(n + k^2) = O(n)$.

Let $S(n)$ be the number of monotone subsequences found. By Erdös and Szekeres' theorem [ES35], if the remaining sequence has size n', the subsequence reported has length $\geq \lceil\sqrt{n'}\rceil$. Therefore

$$S(0) = 0$$

$$S(n) \leq 1 + S(n - \lceil\sqrt{n}\,\rceil)$$

Which solves to $S(n) \leq 2\lfloor\sqrt{n}\rfloor$.

Since the time complexity of the last part of the algorithm is $O(nS(n))$, we get the claimed complexity. □

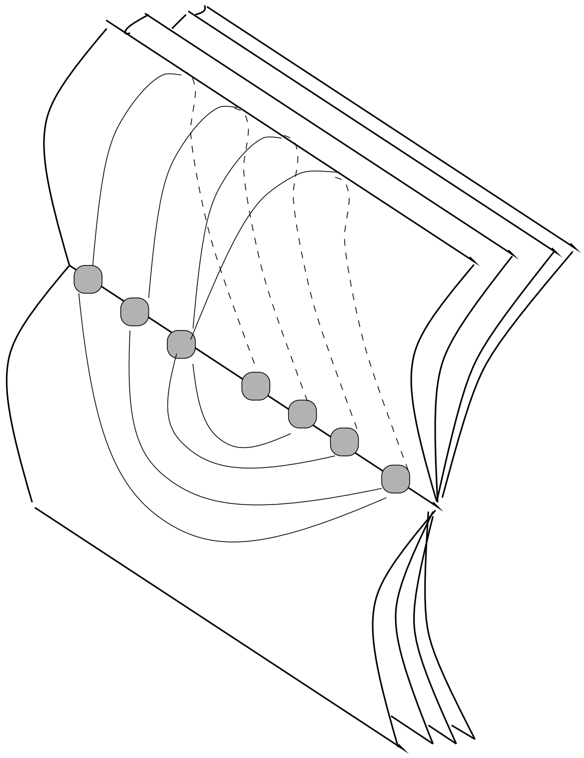

Figure 8: Using single and double sides of a book for the embedding.

4 Applications

4.1 Book Embedding

Note that $GR(M)$ is an x-monotone path and a y-monotone path. Given a graph and a linear ordering of its vertices, we are interested in finding a double-paged book embedding of the graph: We imagine a book, whose 'spine' (the part common to all pages) contains the vertices of the graph in the given order, and assign a page to each edge of the graph in such a way that it is possible to plot the edges on the "pages", such that no two edges assigned to the same page intersect. We are interested in minimizing the number of pages needed for the embedding. We draw our terminology here from [HR90].

Let $G = (V, E)$ be a graph with a fixed linear order of its vertices, so assume $V = (1, 2, \ldots n)$. A *rainbow* $\{(r_1, s_1), \ldots (r_k, s_k)\}$ is a subset of E such that

$$r_1 \leq r_2 \leq \cdots \leq r_{k-1} \leq \ r_k \leq s_k \leq s_{k-1} \leq \cdots \leq s_2 \leq s_1 \ .$$

A *twist* $\{(r_1, s_1), \ldots (r_k, s_k)\}$ is a subset of E such that

$$r_1 \leq r_2 \leq \cdots \leq r_{k-1} \leq r_k \leq s_1 \leq s_2 \leq \cdots \leq s_{k-1} \leq s_k \ .$$

Clearly, a rainbow or a twist can be embedded in a single side of a single page of the book (see figure 9). Suppose we have embedded a set of edges $E_1 \subseteq E$ in one page, and another set $E_2 \subseteq E$ in another page, and that all the endpoints of the edges of E_1 are to the left of the endpoints of the edges of E_2. Then E_1 and E_2 can be embedded in a single page with no intersections.

The idea of our algorithm is to find a *good bisection* of V; that is, a partition of $1, \ldots, n$ into a prefix $V_1 = \{1 \ldots i\}$ and a suffix $V_2 = \{i + 1 \ldots n\}$ such that none of the induced subgraphs contains more than $|E|/2$ edges. We divide the edges that cross the "bisection line" (from V_1 to V_2) into a small number of rainbows and twists, and apply the procedure recursively to both subgraphs. Obviously, solutions (twists and/or rainbows) from different branches of the recursions can be embedded in a single page. Before describing the algorithm, let us show how we handle bipartite graphs. Let $G = (X, Y, E)$ be a bipartite graph, where $X = \{x_1, x_2, \ldots\}$ and $Y = \{y_1, y_2, \ldots\}$.

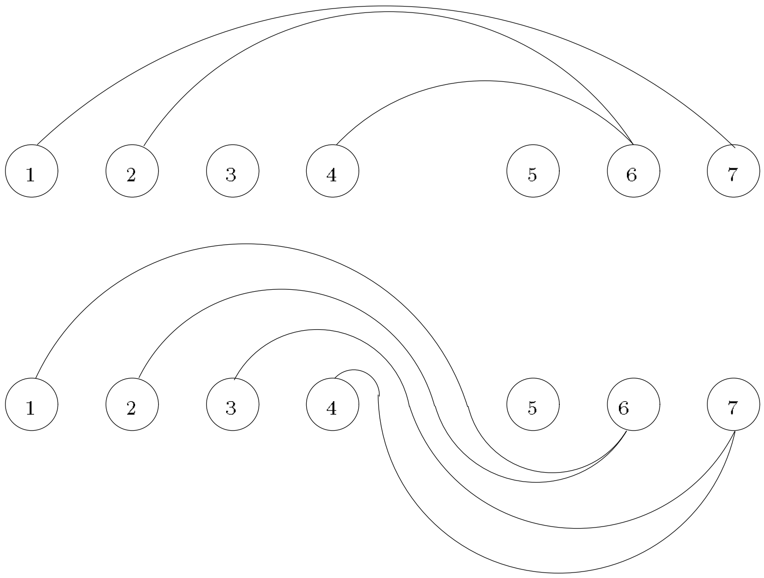

Figure 9: A rainbow (above) and a twist (below)

Assume $E = \{(x_{i_1}, y_{i_1}), \ldots (x_{i_m}, y_{i_m})\}$ is ordered in such a way that $k < l$ implies that $x_{i_k} \leq x_{i_l}$. A subsequence of E that is increasing (resp. decreasing) on y_i is surely a twist (resp. rainbow) of G. Therefore, using Theorem 11, we have

Lemma 12 *We can find, using $O(|E|^{1.5})$ time and $O(n)$ space, an embedding of G using $2\lfloor\sqrt{|E|}\rfloor$ pages.*

Before turning to regular graphs, let us show that a good bisection always exist. Let $G = (V, E)$ be a graph and $V' \subseteq V$. Denote by $\bar{G}(V')$ the subgraph of G induced by the vertices of V', and by $|\bar{G}(V')|$ the number of edges of $\bar{G}(V')$. Let $[i, j]$ denote in the rest of this section the integers $\{i, i+1, \ldots, j\}$.

Lemma 13 *For every graph $G = ([1,n]), E)$ there is a good bisection of G. That is, there exists an i such that $|\bar{G}([1,i])| \leq \frac{|E|}{2}$ and $|\bar{G}([i+1,n])| \leq \frac{|E|}{2}$.*

Proof: Consider the largest i such that $|\bar{G}([1,i]\})| \leq \frac{|E|}{2}$, so $|\bar{G}([1,i+1])| > \frac{|E|}{2}$, and $|\bar{G}([i+2,n])| < \frac{|E|}{2}$. There are no edges common to $\bar{G}(\{1,2,\ldots,i+1\})$ and $\bar{G}(\{i+1,i+2,\ldots,n\})$, and the sum of their sizes is at most $|E|$, and no edge whose endpoints are $k, i{+}1$ (for $k \leq i$) appears in $\bar{G}([i{+}1,n])$. It follows that $\bar{G}([i{+}1,n]) \leq \frac{|E|}{2}$.
□

Embedding General Graphs:

> **Algorithm** *Embed*
> **Input:** $G = (\{1,2,\ldots,n\},E)$
> **Output:** An embedding of G
>
> 0. If $|E| = 0$ exit.
> 1. Find i such that $|\bar{G}([1,i])| \leq \frac{|E|}{2}$ and $|\bar{G}([i+1,n])| \leq \frac{|E|}{2}$.
> 2. Embed the bipartite graph $G = (V, \{(j,k) \in E \ : \ j \leq i < k\})$, using Lemma 12.
> 3. Apply recursively the algorithm to $\bar{G}([1,i])$ and to $\bar{G}([i+1,n])$.
> 4. Let l_1 and l_2 denote the smallest number of pages needed (respectively) to embed each of the results of the recursive applications of algorithm *Embed*. combine these embedding into $\max\{l_1,l_2\}$ pages, as described above.

Lemma 14 *Let $n = |E|$. Algorithm Embed embeds any graph in $2(1+\sqrt{2})\sqrt{|E|}$ or less pages. The embedding is found in time $O(|E|^{1.5})$ and $O(|E|)$ space.*

Proof: By Lemma 13, step 1 can be implemented in time $O(n)$. Step 2 can be implemented in time $O(n^{1.5})$, (see Lemma 12). Step 4 is trivial, thus the worst case time $T(|E|)$ satisfies $T(0) = O(1)$ and

$$T(|E|) \leq O(|E|^{1.5}) + 2T(\frac{|E|}{2})$$

which implies $T(|E|) = O(|E|^{1.5})$.

Now let $P(m)$ denote the worst case number of pages produced by the algorithm for a graph with m edges. Thus

$$P(m) \leq \max_{\substack{x, y \leq \frac{m}{2} \\ x+y+z=m}} \left\{ 2\sqrt{z} + \max\{P(x), P(y)\} \right\}$$

By the induction hypothesis and elementary mathematics,

$$P(m) \leq 2 \max_{x \leq \frac{m}{2}} \left\{ \sqrt{m-x} + (1+\sqrt{2})\sqrt{x} \right\}$$

Using some algebraic manipulations

$$P(m) \leq 2\sqrt{m} \max_{0 \leq t \leq \sqrt{\frac{1}{2}}} \left\{ (1-t)\left(\sqrt{\frac{1+t}{1-t}} - (1+\sqrt{2}) \right) + (1+\sqrt{2}) \right\}$$

Since $(1+\sqrt{2}) = \max_{0 \leq t \leq \sqrt{\frac{1}{2}}} \sqrt{\frac{1+t}{1-t}}$, we have that

$$P(m) \leq 2\sqrt{m} \max_{0 \leq t \leq \sqrt{\frac{1}{2}}} \{(1-t)0 + (1+\sqrt{2})\} \ ,$$

Thus $P(m) \leq 2(1+\sqrt{2})\sqrt{m}$. □

4.2 Good Splitters

Good splitters were proposed in [MW89] as a tool for some planar simplex range searching problems. The basic idea of good splitters is that of a *sheaf*. Let H be a set of lines in the plane, and let b be a vertical line. A set $F \subseteq H$ is called a right (resp. left) sheaf with respect to b if no two lines of F intersect to the right (resp. left) of b (see figure 10). A k-sheaf is a sheaf of size k. The idea behind good splitters is the fact that if F is a right (left) sheaf with respect to b and P is a point to the right (left) of b, then the number of lines of F above P can be found by a binary search. If we sort the lines of H by the height of their intersection with b, then a right (left) sheaf is an increasing (decreasing) subsequence of the sequence of the slopes.

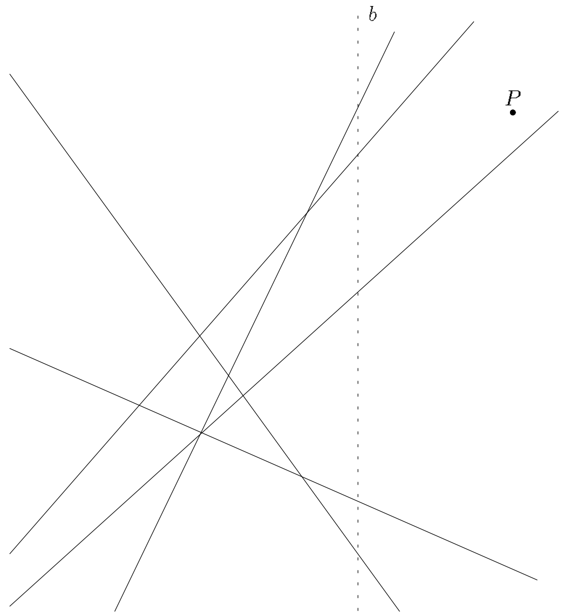

Figure 10: A right sheaf with respect to b

Let $\mathcal{S}$ be a collection of sets. We denote by $|\mathcal{S}|$ the number of sets in $\mathcal{S}$, and by $\|\mathcal{S}\|$ the cardinality of the union of the sets of $\mathcal{S}$. A good splitter is a triple $(b, \mathcal{T}_L, \mathcal{T}_R)$ where b is a vertical line and for $X \in \{L, R\}$:

1. $\mathcal{T}_R$ is a collection of right sheaves with respect to b.
   $\mathcal{T}_L$ is a collection of left sheaves with respect to b.
2. The sheaves of $\mathcal{T}_X$ are pairwise disjoint.
3. $|\mathcal{T}_X| < \frac{7}{4}\sqrt{n}$
4. $\|\mathcal{T}_X\| > \frac{3}{8}n$
5. $\|\mathcal{T}_L\| + \|\mathcal{T}_R\| \geq n$

The basis for finding good splitters is Algorithm *Split*.

Algorithm *Split* [MW89]

Input: A set of lines H, a positive number κ.

Output: A vertical line b, a set of left κ-sheaves S_L and a set of right sheaves S_R, both with respect to b, such that $\|S_L\| > \frac{n-\kappa^2}{2}$ and $\|S_R\| > \frac{n-\kappa^2}{2}$. Let $V(H)$ denote the intersection points between lines of H, and let $X(H)$ denote their x-coordinates.

(1) $b_L \leftarrow$ a vertical line at $-\infty$, $b_R \leftarrow$ a vertical line at $+\infty$, $t \leftarrow \frac{n-\kappa^2}{2}$.
While there is a set $S \subseteq H$ of parallel lines whose size is κ do
$H \leftarrow H - S$; $S_L \leftarrow S_L \cup S$; $S_R \leftarrow S_R \cup S$
If $\|S_L\| > t$ stop with answer $b = 0$, S_L and S_R.

(2) Choose a vertical line b_M such that the number of elements of $V(H)$ between b_R and b_M is approximately half of the number of elements of $V(H)$ between b_R and b_L.
$S'_L \leftarrow S_L$ and $S'_R \leftarrow S_R$.

(3) While it is possible, add to S'_L left κ-sheaves with respect to b_M (will be described in detail below). While it is possible, add to S'_R right κ-sheaves with respect to b_M.

(4.1) If $\|S'_L\| > t$ and $\|S'_R\| > t$ then stop with answer $b = b_M$, $S_L = S'_L$, $S_R = S'_R$.

(4.2) If $\|S'_L\| \leq t$ then $b_L \leftarrow b_M$, $S_L \leftarrow S'_L$. Continue at step (2).

(4.3) $b_R \leftarrow b_M$, $S_R \leftarrow S'_R$. Continue at step (2).

Lemma 15 [MW89] *For every set H of lines and every $\kappa > 0$, Algorithm Split computes a vertical line b and sets of κ-sheaves S_L and S_R such that $\|S_L\| > \frac{n-\kappa^2}{2}$ and $\|S_R\| > \frac{n-\kappa^2}{2}$.*

Lemma 16 *Algorithm Split can be implemented with $O(n)$ space and $O(n \log^2 n + n^2/\kappa + n\kappa)$ running time.*

Proof: Follows from the proof in [MW89] and Lemma 10. Step (1) can be implemented in time $O(n \log n)$ by just sorting the slopes. Step (2) can also be implemented in time $O(n \log n)$ (see [MW89]), giving an overall time of $O(n \log^2 n)$, since the algorithm has $O(\log n)$ iterations.

As in [MW89], in order to make steps (3) and (4) efficient, we consider them as a unit, and add κ-sheaves to S'_L and S'_R in alternating order until one of these cases occurs:

(A) $\|S'_R\| > t$: Then we try to add κ-sheaves to S'_L. If $\|S'_L\| > t$ we add as many sheaves as possible to S'_L and S'_R and stop as in (4.1). If S'_L cannot be made large enough, we continue as in (4.2).

(B) $\|S'_L\| > t$: Symmetric to (A)

(C) S'_R cannot be extended, although $\|S'_R\| \le t$: Continue as in (4.3)

(D) S'_L cannot be extended, although $\|S'_L\| \le t$: Continue as in (4.3)

In step (3), we will first sort the lines by the height of their intersection with $x = b$. The overall time needed for sorting is $O(n \log^2 n)$. Finding a right (left) κ-sheaf will be equivalent to finding an increasing (decreasing) subsequence of size κ of the sequence of the slopes. This can be done in time $O(n + \kappa^2)$, using Lemma 10. Every time we find a sheaf, unless we stop, S'_L or S'_R increases, say by n_i sets. Therefore, the time needed in each application of steps (3) and (4) is $O(n_i(n + \kappa^2))$. Since $\|S_L\|$ and $\|S_R\|$ never shrink, the overall cost of steps (3) and (4) is $O(n \log^2 + n\kappa)$ □

Lemma 17 *Every set H of n lines in the plane has a good splitter which can be computed in $O(n^{1.5})$ time and $O(n)$ space.*

Proof: In [MW89] it is proved that a good splitter can be found in time $O(n^{1.5} \log n)$. The complexity follows from two steps: Applying Algorithm *Split* with $\kappa = \lfloor \frac{1}{2}\sqrt{n} \rfloor$ to H, and partitioning the m lines that do not appear neither in S_L nor in S_R into $2\lfloor \sqrt{m} \rfloor$ sheaves. By Lemma 10, the first part can be done in time $O(n^{1.5})$, and by Lemma 10, the second part can also be done in time $O(n^{1.5})$. □

Good splitters are the basis of *ES-trees* [MW89]. An ES-tree for a set of lines H is a data structure that holds at its root a good splitter $(b, \mathcal{T}_L, \mathcal{T}_R)$, the right sibling is a recursively defined structure for $H - \cup \mathcal{T}_R$, and the left sibling is a recursively defined structure for $H - \cup \mathcal{T}_L$. The space required by an ES-tree is $O(n \log n)$, and the time required for building it is $O(n^{1.5} \log n)$. Using our algorithms, the time required for building an ES-tree is improved to $O(n^{1.5})$.

An ES-tree was used in [MW89] for counting how many lines of H lie above a query point p: If p is to the right (left) of b, count how many lines of each sheaf of $\mathcal{T}_R$ ($\mathcal{T}_L$) are above p, and apply recursively for the right (left) subtree. The time needed for a query is then $O(\sqrt{n}\log n)$. They also show that counting points inside a query triangle can be done in the same time bounds. In [BF90] we show some new applications of ES-trees, mainly for problems related to ray shooting.

References

[BF90] R. Bar Yehuda and S. Fogel. Variations on Ray Shooting, *Algorithmica* 11 (1994), 133-145.

[CLR87] F. R. Chung, F. T. Leighton, and A. L. Rosenberg. Embedding graphs in books: a layout problem with applications to VLSI design. *SIAM J. Algebr. Discr. Methods.*, 8:33–58, 1987.

[Dij80] E. W. Dijkstra. Some beautiful arguments using mathematical induction. *Acta Informatica*, 13:1–8, 1980.

[ES35] P. Erdös and G. Szekeres. A combinatorial problem in geometry. *Compositio Mathematica*, 2:463–470, 1935.

[Fre75] M. L. Fredman. On computing the length of longest increasing subsequences. *Discrete Mathematics*, 11:29–35, 1975.

[HR90] L. Heath and A. L. Rosenberg. *Graph Layout Using Queues, manuscript.* 1990.

[Joh82] D. B. Johnson. A priority queue in which initialization and queue operations take $O(loglogD)$ time. *Math. System Theory*, 15:295–309, 1982.

[KO88] R. Karlsson and M. Overmars. Scanline algorithms on a grid. *BIT*, 28:227–241, 1988.

[MW89] J. Matoušek and E. Welzl. Good splitters for counting points in triangles. In *5th Ann. ACM Conf. On Computational Geometry.*, pages 124–130, 1989.