# Reporting Red-Blue Intersections Between Two Sets Of Connected Line Segments

Julien Basch, Leonidas J. Guibas, and G. D. Ramkumar

Department of Computer Science
Stanford University
Stanford, CA 94305, USA
e-mail: {jbasch, guibas, ram} @ cs.stanford.edu

**Abstract.** We present a new line sweep algorithm, HeapSweep, for reporting bichromatic ('purple') intersections between a red and a blue family of line segments. If the union of the segments in each family is connected as a point set, HeapSweep reports all $k$ purple intersections in time $O((n+k)\alpha(n)\log^3 n)$, where $n$ is the total number of input segments and $\alpha(n)$ is the familiar inverse Ackermann function. To achieve these bounds, the algorithm keeps only partial information about the vertical ordering between segments of the same color, using a new data structure called a *kinetic queue*. In order to analyze the running time of HeapSweep, we also show that a simple polygon containing a set of $n$ line segments can be partitioned into monotone regions by lines cutting these segments $\Theta(n\log n)$ times.

## 1 Introduction

The problem of finding and reporting all pair-wise intersections in a set of line segments is among the first to have been studied in computational geometry; its solution established the use of sweep-line methods and introduced the notion of output sensitive algorithms. A common variation is the *red-blue intersection problem*, where two families of line segments are given – a red and a blue – and only the bichromatic (*purple*) intersections are to be reported. Various assumptions can be made about each family. For our purposes, a family of line segments will be called *disjoint* if no pair of segments intersect (except possibly at their endpoints), *connected* if the union of all segments is connected as a point set, and *general* otherwise. Each assumption leads to different algorithms and running times. In what follows, we denote by $n$ the number of (red and blue) input segments, and by $k$ the number of purple intersections.

The case where each input set is disjoint has been extensively studied. In the early days of computational geometry, Bentley and Ottmann [7] introduced their well-known line sweep algorithm, which reports all purple intersections in time $O((n+k)\log n)$. Mairson and Stolfi [22] were the first to find an asymptotically optimal algorithm running in $O(n\log n+k)$ time. This second algorithm searches for and reports purple intersections at endpoint events during the sweep; the use of a clever 'red-blue cone' invariant allowed the authors to get rid of the

$O(\log n)$ overhead on the number $k$ of intersections. The same optimal time bound for reporting purple intersections was achieved later by several different methods [9, 10, 12, 27]. Especially worthy of mention among these is the hereditary segment tree data structure of Chazelle, Edelsbrunner, Guibas, and Sharir [10], which can also be used to *count* all purple intersections in $O(n \log n)$ time. Thus, in the disjoint case, asymptotically optimal and practical algorithms are known.

For general inputs, where arbitrary intersections are allowed within each family, a Bentley-Ottmann sweep reports all purple intersections in $O((n + k + k') \log n)$ time, where $k'$ is the total number of monochromatic intersections. For certain inputs $k'$ can be $\Omega(n^2)$, while the desired output can be much smaller. This version of the problem seems to be of about the same difficulty as reporting all segment intersections in a given family. For this latter problem, Guibas, Overmars, and Sharir [20] gave a randomized $O(n^{4/3+\epsilon} + k)$ solution using partitioning techniques (what has since become known as cuttings; derandomization tools developed later can also be applied). Agarwal [1] adapted this technique to the red-blue version of the problem, lowering the running time to $O(n^{4/3} \log n + k)$ for reporting and obtained a method for counting in time $O(n^{4/3} \log n)$ as well, at the cost of increasing the space requirement to $\Theta(n^{4/3})$. A significant improvement in these time bounds is unlikely, as the general case falls within a class of problems at least as hard as Hopcroft's problem of detecting an incidence between a set of $n$ points and a set of $n$ lines; the fastest known algorithm for Hopcroft's problem is due to Matoušek [15, 24], and runs in $O(n^{4/3} 2^{O(\log^* n)})$ time. For a certain general class of algorithms, a lower bound of $\Omega(n^{4/3})$ for this problem was recently given by Erickson [19].

Given this situation, we were motivated to seek additional conditions which make easier the task of reporting the purple intersections. A natural condition is that of *connectedness* for each of the monochromatic inputs, as introduced above. This condition often pertains in situations where the purple intersection problem itself arises (e.g., in the overlay of line arrangements, or of simply-connected planar subdivisions – though in the latter case the disjoint methods above apply). Under the hypothesis of connectedness, Agarwal and Sharir [3] looked at the problem of detecting a *single* purple intersection. Their technique is based on the following idea: pick a point $z$ on the red collection and compute the blue face $\mathcal{F}$ that contains $z$ using a general algorithm to compute a single face in an arrangement of line segments. Next, pick a blue point on the boundary of $\mathcal{F}$ and compute the red face $\mathcal{F}'$ that contains it. If the blue and red segment sets are connected, the set of purple intersections is non-empty iff the boundaries of $\mathcal{F}$ and $\mathcal{F}'$ intersect, and this can be tested via a Bentley-Ottmann sweep. Using the recent randomized algorithm of Chazelle *et al.* [11] to compute a single-face, a purple intersection is detected in time $O(n\alpha(n) \log n)$, where $\alpha(n)$ is the slowly growing inverse of Ackermann's function.

Some of the algorithms described above are essentially combinatorial [7, 22, 3, 12], and work as well if the line segments are replaced by $x$-monotone algebraic arcs of bounded degree. Others [9, 11, 1] make a more essential use of the affine structure of the input and cannot be so adapted. When considering $x$-monotone arcs such that any pair intersects at most $s$ times, it is common to encounter the function $\lambda_s(n)$, which denotes the almost-linear maximum length of an $(n, s)$ Davenport-Schinzel sequence; for more material on Davenport-Schinzel sequences the reader is referred to [28].

In this paper, we present a new line sweep algorithm called HeapSweep, that reports all the purple intersections between two sets of connected segments. This algorithm runs in time $O((n + k)\alpha(n)\log^3 n)$. If the union of the $n_r$ segments of $R$ and the union of the $n_b$ segments of $B$ consist of $c_r$ and $c_b$ connected components respectively, the running time becomes $O((c_b n_r + c_r n_b + k)\alpha(n)\log^3 n)$. If a point is known in each component, then the algorithm can be adapted to report all purple intersections in $O((n_r\sqrt{c_b} + n_b\sqrt{c_r} + k)\alpha(n)\log^3 n)$ time. Furthermore, the HeapSweep algorithm generalizes to more general arcs, as described in the previous paragraph. In this case, it reports all purple intersections in time $O(\lambda_{s+2}(n + k)\log^3 n)$. To achieve these bounds, we revisit the line sweep paradigm and relax the requirement that the segments be completely ordered along the sweep line. The algorithm uses a new data structure called a *kinetic queue* [5], for which we describe both a randomized (a *heater*) and a deterministic implementation (a *kinetic tournament*). The analysis of HeapSweep requires a combinatorial lemma on the monotone decomposition of a polygon containing line segments, as was briefly mentioned in the abstract. We believe that both the kinetic queue and this lemma are of independent interest. In the connected case, our algorithm is the first output-sensitive and nearly optimal algorithm for the red-blue intersection reporting problem.

A reader interested enough to continue reading will find a description of HeapSweep in Section 2 (including a discussion of kinetic queues), the proof of the polygon partitioning lemma in Section 3, the analysis of the running time of the algorithm in Section 4, related results in Section 5, and a discussion of open problems in Section 6.

## 2 The HeapSweep Algorithm

We assume that the input is non-degenerate, i.e. that all end segment points and intersections occur at distinct $x$ values. Standard perturbation techniques can be used to guarantee that this is always the case [17].

In the traditional Bentley-Ottmann line sweep, a balanced search tree is used to represent the state of the sweep line. This search tree stores the exact top-to-bottom ordering of the line segments intersecting the sweep line. In order to maintain this ordering as the sweep progresses, *all* intersections between line segments, namely red-red, blue-blue, and purple intersections have to be detected and processed. Our algorithm follows the general principle of the line sweep technique, with a global event queue $\mathcal{Q}$ for segment endpoints and intersections (which are scheduled as they are discovered), but it stores only a partial ordering of the segments on the sweep line.

We divide the set of line segments intersecting a given position of the sweep line into a sequence of contiguous monochromatic blocks $\Gamma_1, \Gamma_2, \ldots, \Gamma_t$ ($\Gamma_1$ is the top block), and we keep the ordering of the blocks but not the exact order of segments within each block. In order to preserve this structure and to detect purple intersections as the sweep progresses, we need to look for possible intersections between the top and bottom segments of adjacent blocks along the sweep line. For this purpose, each block $\Gamma_i$ is stored in a new data structure, called a *kinetic queue*, which is an enhancement of a standard priority queue (with efficient insertion, deletion, and access to the top priority element) allowing for data with continuously changing priorities. In our case, the priority of a segment is the vertical coordinate of its intersection with the sweep line; actually, two kinetic queues with opposite priorities are maintained for each block, allowing efficient access to the smallest and largest element within each block.

A kinetic queue can be implemented by a standard priority queue data structure, such as a binary heap. What is novel in the kinetic setting is that the priorities of the elements in the heap are continuously changing, as the sweep proceeds. Whenever the sweep-line encounters a monochromatic intersection between a parent-child pair in this heap, the heap structure needs to be updated – but fortunately this is relatively straightforward. Furthermore, these updates can be scheduled using the same mechanism used by the sweep as a whole, namely by the maintenance of a global event queue $\mathcal{Q}$ ordered by time. We will call such monochromatic intersection events processed by the algorithm *internal* to the corresponding kinetic queue. What makes the implementation of a kinetic queue challenging is the need to keep low the number of internal events processed.

In order to process the events corresponding to when a new segment starts or an old segment ends during the sweep, we will store the top and bottom elements of each block in a balanced binary search tree $\mathcal{T}$, according to the ordering of the blocks along the sweep line. Note also that the purple intersections of interest will always be between the bottom segment in one block and the top segment in the following block in the ordering. Thus each pair of consecutive bichromatic elements along $\mathcal{T}$ schedules a possible future purple intersection event, which is re-scheduled each time the top or bottom segment of one of these blocks changes (due to an internal event in one of the associated kinetic queues). Purple intersection events themselves typically create two new blocks of size one (Fig 1–a), which necessitates the modification and re-balancing of $\mathcal{T}$.

### 2.1 Endpoint Events

In a traditional sweep line algorithm, segment endpoints are processed easily by locating their position along the sweep line, and inserting or deleting the corresponding segment at that position. The situation is more delicate when it comes to HEAPSWEEP.

When an endpoint starting a blue segment is encountered during the sweep, the block in which it is located can be found using $\mathcal{T}$. If this block is blue, the segment can be inserted directly into the corresponding kinetic queue(s); if the block is red, however, the blue segment *splits* the red block into two pieces.

Similarly, when a segment ends, it may cause a block to vanish and require a *merge* of two existing blocks of the same color (Fig. 1–b). Both block splits and merges can require time proportional to the size of the blocks on which they are performed. In a standard fashion, this cost can be reduced to a number of kinetic queue insertion or deletion operations *proportional to the size of the smaller block involved.* A merge is implemented by inserting each segment from the smaller of the two blocks into the larger one. A split is implemented by alternately deleting segments from the top and the bottom of the block until we reach the position where the new segment has to be inserted. If this position is reached from the top first, say, then the segments deleted from the bottom are inserted back into the block, while those deleted from the top are inserted into a new (and initially empty) block.

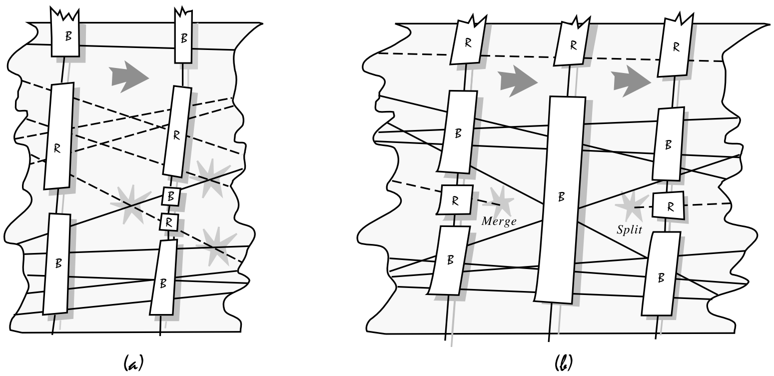


**Figure 1.** (a) A purple intersection: two new blocks are created, and new purple intersection events are scheduled. (b) A merge of two blue blocks at a red right endpoint followed by a split of a blue block caused by a red left endpoint.

In summary, a split and a merge can be implemented at a cost in terms of kinetic queue operations which is proportional to the number of segments present in the smaller of the two blocks involved. This cost is still high in the worst case but, as we will see in Section 3, the *connectedness* of the input guarantees that the worst case doesn't happen too often. Without the connectedness hypothesis (even in the disjoint case) it is straightforward to construct an example with no purple intersections where HeapSweep would run in quadratic time.

### 2.2 Kinetic Queue Implementations

As was mentioned above, a natural way to implement a kinetic queue is as a standard binary heap. However, we have been unable to prove satisfactory bounds on the number of internal events needed to maintain such a structure during the sweep. Instead, we propose two different implementations of kinetic queues below, in which the number of internal events is provably within a log factor of the total number of objects present in the queue. The first implementation, called a *heater*, is randomized. The second, called a *kinetic tournament*, is deterministic. Though the asymptotic performance of the heater is not superior to its deterministic counterpart, we have chosen to present it – even to emphasize it – below, because we feel it may be of independent interest and is likely to perform better in practice.

To define a heater we proceed as follows. If each object in a set is given two numbers, a *rank* and a *priority*, there is a unique binary tree which is both a search tree on the ranks and a heap on the priorities. Such a tree is well-known and called a *treap* – Aragon and Seidel [4] used it to create their popular randomized search tree data structure, which is a treap on objects with a given rank and a randomly assigned priority. The randomization guarantees that this structure is balanced with high probability. A heater is like a treap, but this time, priorities are given and ranks are random. When an element with a given priority is inserted in a heater, it is first assigned a random rank, and inserted at the appropriate leaf of the heater. It then bubbles up with a sequence of rotations until it reaches a position consistent with its priority. Deletions are implemented in an analogous way.

When a heater is used to implement a kinetic queue in HeapSweep, we need to detect and process certain internal events in order to keep the heater consistent with the continuously changing priorities (Fig 2). These internal events are the intersections between parent-child segment pairs in the heater. When such an event occurs, a rotation involving the parent and child is performed to keep the heater consistent – this is sufficient, as at that moment the parent and child have equal priorities and no other ordering in the heater can be changing at the same time (by our non-degeneracy assumption). After the rotation, two parent-child relationships change in the heater. Thus up to two existing events in the sweep event queue $\mathcal{Q}$ may have to be de-scheduled (deleted), and two new events scheduled (inserted).

A kinetic tournament is another implementation of a kinetic priority queue based on the static tournament tree structure for leader election. It is built on a perfectly balanced tree which represents a tournament among all elements which ever appear in the the kinetic queue. Here also, to keep the structure consistent with the changing priorities during the sweep, an internal event has to be scheduled in $\mathcal{Q}$ for each game of the tournament whose outcome can change in the future. When such an event happens, the new winner has to be percolated up the tree. Similar percolations need to happen when a new object appears or an old one disappears. Thus, unlike the heater case, tournament events in $\mathcal{Q}$ can cause a number of de-schedulings and re-schedulings proportional to the height of the tournament tree.

In section 4, we will obtain bounds on the number of internal events that may be required to maintain these two implementations of a kinetic queue.

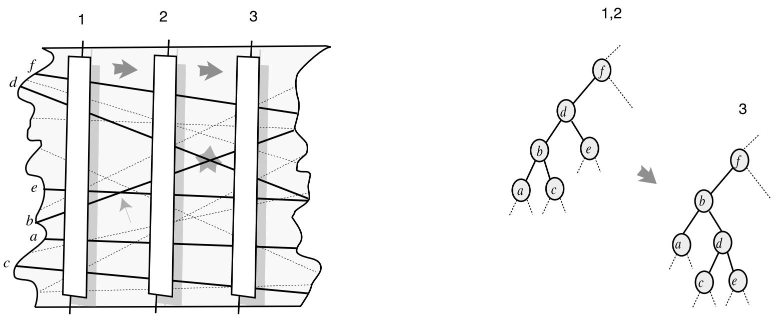


**Figure 2.** Three positions of the sweep line and the corresponding heaters with attention focused on line segments labeled (randomly) $a$ through $f$. As the sweep line moves from position 1 to 2 passing the intersection between $b$ and $e$, the heater remains unchanged (since $b$ and $e$ are not a parent-child pair). However, when it moves from position 2 to 3 passing the intersection between parent-child pair $b$ and $d$, a rotation is performed that swaps the priorities of $b$ and $d$ but otherwise preserves the inorder sequence.

## 3 Economical Polygon Regularization

We call regularization the process of decomposing a polygon (or more generally, a subdivision) in the plane into $x$-monotone simply connected regions. In this section we look at the following situation: we have a simple polygonal face $F$ (in an arrangement) containing a set $R$ of (possibly intersecting) line segments. We show that $F$ can be regularized via a number of vertical threads which cause only $O(|R| \log |R|)$ cuts of these segments. In the next section, the cost of Merge and Split operations of the HEAPSWEEP algorithm will be shown to depend on these cuts.

A point on the boundary of the polygon $F$ is called *critical* if it is locally $x$-extremal and concave. In other words, a critical point is a concave vertex of $F$ where both incident edges are to the left or to the right of the vertex.

**Definition 3.1. (Economical regularization)** The *$R$-economical regularization* of $F$ is the decomposition of $F$ into $x$-monotone regions, obtained by the following procedure: from each critical vertex $v$ of $F$, draw two threads vertically, up and down from $v$, until they reach the polygon boundary. Keep only the thread that intersects the fewest elements of $R$.

The crucial aspect here is that we only keep the 'shorter' of the two threads at each critical vertex. It is easy to see that the economical regularization defines a monotone decomposition of $P$. We define the cost of the regularization to be the total number of intersections ('cuts') between segments of $R$ and the selected threads. To analyze this cost, we use a lemma reminiscent of the classical analysis of the simplest UNION-FIND algorithm:

**Lemma 3.2.** *Let $\mathcal{T}$ be a tree and denote the left (resp. right) child of a node $\nu$ by $\ell(\nu)$ (resp. $r(\nu)$). Each node $\nu$ is given a (possibly negative) integer* weight *$w_\nu$. The* subtree weight *$W_\nu$ of a node $\nu$ is the weight of its subtree: $W_\nu = w_\nu + W_{\ell(\nu)} + W_{r(\nu)}$. Define the* cost *of a node $\nu$ as the weight of its lightest child, i.e. $c_\nu = \min(W_{\ell(\nu)}, W_{r(\nu)})$, and the* absolute weight *of $\mathcal{T}$ to be $S_\mathcal{T} = \sum_\nu |w_\nu|$. Then, if at least one node of $\mathcal{T}$ has a non-zero weight, we have:*

$$\sum_\nu |c_\nu| \leq S_\mathcal{T} \log S_\mathcal{T}.$$

*Proof.* By induction on the structure of the tree. □

### 3.1 The Reachability Tree

We proceed to define a tree decomposition of the simple polygon $F$, on which we later use the above lemma. In order to do so, the critical vertices need to be separated in two different groups: the *split vertices* of $F$, that are locally $x$-minimal, and the *merge vertices*, that are locally $x$-maximal. We now focus only on the split vertices, and later invoke the same technique for the merge vertices by reversing the $x$-axis. By non-degeneracy, we may assume that no two critical vertices of $F$ have the same $x$-coordinate.

**Definition 3.3. (Up/down reachability)** Let $F$ be a simple polygon, and $v$ be a split vertex of $F$. A point $p \in F$ to the right of $v$ is *up (resp. down) reachable* from $v$ if there exist a path from $v$ to $p$ inside $F$ which starts locally above (resp. below) $v$ and remains to the right of $v$.

**Definition 3.4. (Up/down child, up/down region)** A point $p$, $p \in F$ is an *up (resp. down) child* of split vertex $v$ if $v$ is the rightmost split vertex from which $p$ is reachable, and $p$ is up (resp. down) reachable from $v$. The set of all up (resp. down) children of $v$ is called the *up-region* (resp. *down-region*) of $v$, The set $r$ of points with no parent is called the *orphan region*.

Since $F$ is simple, no point can be simultaneously up and a down reachable from the same vertex, so that the regions defined above partition $F$. Each region is delimited by portions of the boundary of $F$ and by portions of vertical lines passing through split vertices. Note that (i) a region need not be connected, and (ii) an up-child is not necessarily 'above' its parent inn the plane (Fig. 3–a).

**Lemma 3.5.** *A region contains at most one split vertex on its right boundary.*

*Proof.* By contradiction. □

We now build a connectivity graph between these regions, with one node per region; two nodes are connected in this graph if they have a vertical boundary in common. By Lemma 3.5, this graph is a full binary tree called the *reachability tree* of $F$, which we root at the orphan region. If the right boundary of a region contains a split vertex $v$, the associated node in the tree bears the up and down regions of $v$ as its two children (Fig. 3–b).

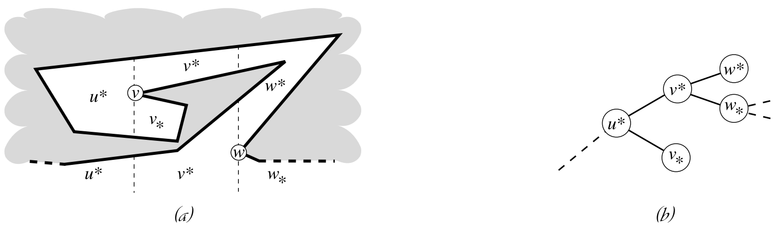


**Figure 3.** (a) A portion of a simply connected polygonal region (in white), and the regions associated with the split vertices (vertex $u$ is not shown); (b) the associated portion of the reachability tree. Note that the merge vertices are ignored in this decomposition.

## 3.2 Cost of the Economical Regularization

**Lemma 3.6.** *Let $F$ be a simple polygon in the plane that contains a set $R$ of (possibly intersecting) segments. The $R$-economical regularization of $F$ cuts the segments of $R$ a total of $O(|R| \log |R|)$ times.*

*Proof.* We consider only the cuts made by the split vertices of $F$, and prove that there are $O(|R| \log |R|)$ of them. The same argument holds for the merge vertices. We use below the vocabulary of Lemma 3.2.

Construct the reachability tree $\mathcal{T}$ of $F$. If a node $\nu$ of $\mathcal{T}$ has corresponding region $\rho$, set the weight $w_\nu$ of $\nu$ to be the number of segments that end in $\rho$ minus the number of segments that start in $\rho$ (a segment "starts" at its left endpoint and "ends" at its right one). Therefore, the subtree weight $W_\nu$ of $\nu$ is simply the number of segments crossing into $\rho$ along its left boundary. If the region $\rho$ terminates on the right with a split vertex $v$, the cost of $\nu$ is either the number of segments entering the up-region of $v$, or the number of segments entering the down-region of $v$, whichever is smaller.

The total absolute weight $S_\mathcal{T}$ of $\mathcal{T}$ is at most the number of end-points of all segments. It follows from Lemma 3.2 that the total cost of $\mathcal{T}$ is $O(|R| \log |R|)$.

For the $R$-economical decomposition of $F$, we choose to make a cut at each split vertex $v$ with the thread (going up or down) that cuts the fewer segments till we reach the boundary of $F$ again. Such a thread is a subset of the left boundary of either the up-region or the down-region of $v$. Therefore, it doesn't cut more segments than the cost (in the above sense) of the region to which $v$ belongs. In particular, by proceeding this way we may choose to cut in the 'wrong' direction, but we will reach the boundary of $F$ and stop cutting before we have spent more than we can afford. □

It is not hard to give an example showing that the bound in this lemma is tight to within a constant factor (a 'ruler function' polygon with one red segment per tick is such an instance – the two threads from each vertex cut the same number of red segments). Our purpose in defining the regularization is to bound the total cost of merge and splits in HEAPSWEEP. This cost is captured by the following corollary:

**Corollary 3.7.** *Let $B$ be a connected set of blue segments on the plane, and $R$ be a set of red segments that don't intersect with $B$. The $R$-standard regularization of all faces of the arrangement defined by $B$ cuts $O(|R| \log |R|)$ segments of $R$.*

*Proof.* Since the blue segments are connected, each face of their arrangement is a simple polygon, except for the outer face. To make the outer face simply connected, cut that face by drawing a vertical cut up from the leftmost blue vertex. All faces are now simply connected, and Lemma 3.6 applies to each (the linear cost of the extra cut in the outer face is absorbed in the overall bound). □

## 4 HeapSweep Analysis

In this section, we obtain expected and worst-case time and space bounds for the algorithm HeapSweep in the connected case of the red-blue intersection reporting problem. We first examine the expected time complexity of a sweep of a set of segments by one heater. Then we perform a similar analysis for a tournament tree. Finally, we show how the economical polygon regularization can be used to chop all segments in pieces so as to reduce the analysis of the HeapSweep algorithm to that of a set of independent kinetic queues. For clarity, the worst case update time of the main event queue $\mathcal{Q}$ is denoted by $t_Q$ (it is clear that this quantity is $O(\log n)$).

### 4.1 Expected Time Bounds for Heaters

We now consider a heater $H$ that sweeps over a set $S$ of $s$ line segments. We recall that the heater structure is at the same time a heap on the intersection height of the segments with the sweep line, and a search tree on random *ranks* assigned to each segment. Expectations are taken over a uniform distribution of all $s!$ rankings of $S$. The good behavior of a heater relies on two facts that we proceed to prove: (i) its depth $D_x(H)$ at sweep position $x$ is logarithmic in expectation, which allows efficient insertion/deletion of elements (Lemma 4.1), and (ii) there are not too many internal updates, that is, segment pairs that are in a parental relationship in $H$ at the time they intersect (Lemma 4.5).

**Lemma 4.1.** *Let $D_x(H)$ be the depth of the heater $H$, sweeping over a set $S$ of segments, at sweep position $x$. Then $E[D_x(H)] = O(\log |S|)$. The expected cost of an insertion or a deletion in $H$ at any time is $O(t_Q \log |S|)$*

*Proof.* This result is an immediate consequence of the original analysis of treaps by Aragon and Seidel [4], once we notice that uniformly random ranks on elements of preassigned priorities defines the same distribution on the space where the ranks are fixed and the priorities are random.

An insertion of an element in the heater thus causes an expected $O(\log |S|)$ rotations. Each rotation disturbs $O(1)$ parental relationships, and requires $O(1)$ updates of the global priority queue. The expected time of an insertion (and of a deletion) is thus $O(t_Q \log |S|)$. □

The rest of this section focuses on bounding the expected number of internal heater updates. The analysis makes crucial use of the randomness of the structure. The analysis proceeds as follows. We first observe that the probability of a given intersection to cause an update in the heater is related only to its *level* (defined below). We then perform a standard computation *à la* Clarkson-Shor [12], based on the well known combinatorial result bounding the complexity of the upper envelope of a set of line segments.

**Definition 4.2 (Level in an arrangement).** Given a set $S$ of segments on the plane assumed to be in a non-degenerate position, the *level* of a point $p$, denoted $\delta_S(p)$ is the number of segments above that point.

**Lemma 4.3.** *Let $H$ be a heater that sweeps over a set $S$ of segments. If $s_1, s_2 \in S$ intersect at vertex $v$, the probability that this intersection modifies the heater is $\frac{2}{\delta_S(v)+2}$.*

*Proof.* Let us place the sweep line just before $v$, where we assume that $s_1$ is below $s_2$, and consider only the set $S'$ made of $s_1, s_2$, and the $\delta_S(v)$ segments that are above $v$. These form a contiguous group at the top of the heater, and we restrict our attention to the pruned tree on $S'$, as well as to the induced random ranking amongst those segments. The intersection will trigger an internal update if and only if $s_2$ is the parent of $s_1$. However, as $s_1$ is the lowest segment in $S'$, it is a leaf of the subtree, so that $s_2$ can be its parent only if they are contiguous in the induced ranking. Now, the ranking on $S'$ is a uniform random variable on all permutations of $S'$, and thus the probability that $s_i, s_j$ are parent and child is $2/(\delta_S(v)+2)$. □

Levels in arrangements of lines and segments are a well-studied topic in computational geometry [2, 25]. Although estimating the exact number of vertices at level $\ell$ has proven difficult, a simple bound on the number of vertices of level at most $\ell$ can be obtained using standard random sampling techniques [12].

**Lemma 4.4.** *Let $S$ be a set of $s$ segments in the plane. Denote by $t_\ell$ the number of vertices that have level exactly $\ell$. Then:*

$$\sum_{i\le\ell} t_i \le (\ell+1)s\alpha(s)\,.$$

*Proof.* This is a special case of Clarkson and Shor [12], Theorem 3.1. □

**Lemma 4.5.** *Let $H$ be a heater sweeping over $s$ segments in the plane. Denote by $C(H)$ the number of intersections that cause an update of $H$. Then $E[C(H)] = O(s\alpha(s)\log s)$. The expected cost of the sweep is $O(t_Q s\alpha(s)\log s)$.*

*Proof.* By linearity of expectation and Lemma 4.3, we have:

$$E[C(H)] = \sum_{\ell=0}^{s-2} t_\ell \frac{2}{\ell+2}\,.$$

Using summation by parts, we replace $t_\ell$ in this expression by its partial sum $T_\ell = \sum_{i \le \ell} t_i$, which is bounded by Lemma 4.4. More precisely:

$$\begin{aligned} E[C(H)] &= 2\frac{T_\ell}{\ell+2}\Big|_0^{s-1} - 2\sum_{\ell=0}^{s-2} T_{\ell+1}\frac{-1}{(\ell+2)(\ell+3)} \\ &\le 2s\alpha(s) + 2\sum_{\ell=0}^{s-2}(\ell+2)s\alpha(s)\frac{1}{(\ell+2)(\ell+3)} \\ &= O(s\alpha(s)\log s)\,.\square \end{aligned}$$

As an internal update is implemented by a single rotation which disturbs the parental relationship of $O(1)$ nodes, the total amount of time spent in internal updates is $O(t_Q s\alpha(s)\log s)$. By Lemma 4.1, the additional expected cost of the $2s$ insertions/deletions is $O(t_Q s \log s)$.

### 4.2 Kinetic Tournament Trees

It turns out that $O(s\alpha(s)\log s)$ is also the correct bound for the number of internal events processed by a kinetic tournament used for sweeping over $s$ segments. Consider a divide-and-conquer strategy for computing the upper envelope of these segments, in which the binary structure of the tournament tree reflects the partitionings of the segments used during the recursion. The computation tree of this algorithm mirrors the kinetic tournament, and all internal events happening at a given node of the tournament tree correspond to features in the upper envelope of the segments subset defined by the subtree rooted at that node. Hence, the number of internal events is exactly equal to the running time of the divide and conquer algorithm, which is well known to be $O(s\alpha(s)\log s)$ [21]. This type of analysis will be elaborated upon in [5].

The tournament tree can be made dynamic as follows. Imagine a very large sequence of leaves and a perfectly balanced prototypical tournament tree built upon that sequence. At any one time the actual elements present in the tournament tree occupy a portion near the leftmost leaf of that prototypical tree, determined by the subtree rooted at the node where the current leader is determined. Insertions are processed by adding new leaves to the right of the current set of leaves, creating, on occasion, a new root to accommodate more elements. In order to hold the structure in an amount of space linear in the number of nodes present, upon deletion of a leaf element, the lowest node representing a game with that element absorbs and removes its other (non-deleted) child (this works just like the 'easy case' in binary tree node deletion – when the node being removed can just be short-circuited). Both insertions and deletions must then percolate up the tournament tree.

### 4.3 Time and Space Complexity of HeapSweep

It is time to conclude the analysis by putting together the regularization lemma and the analysis of heaters. Recall that $n$ is the total number of red and blue segments, and that $k$ is the number of bichromatic intersections that the algorithm will output.

**Theorem 4.6.** *The* HEAPSWEEP *algorithm reports all purple intersections between red and blue segments using linear space and* $O((n+k)\alpha(n)\log^3 n)$ *time.*

*Proof.* The working space requirement of HEAPSWEEP is simple to assess. Each kinetic queue requires linear space in the number of segments it maintains, and it is responsible for a number of events in the global queue $\mathcal{Q}$ that is also linear in its size. Other than endpoints and kinetic queues internal events, the only other events in $\mathcal{Q}$ are the scheduled purple intersections, of which at any time there can be at most one per red-blue alternation along the sweep line. Thus the total size at any time of the search tree $T$ for endpoints, of the event queue $\mathcal{Q}$, and of all the kinetic queues together, is $O(n)$. We use a straightforward binary heap implementation for $\mathcal{Q}$, so an update (insertion or deletion) in the global queue takes time $t_Q = O(\log n)$.

We now restrict our attention to the time complexity of the maintenance of the red blocks. As each segment may appear in several kinetic queues during its life-time, it is necessary to chop all segments into a reasonable number of *fragments*, so that each fragment appears in only one kinetic queue.

The red segments are broken up in two steps. First, red segments are broken at all purple intersections into *proto-fragments*. Such an intersection can now be seen (from the point of view of red blocks) as the deletion of a red proto-fragment from one kinetic queue, followed by an insertion of *another* proto-fragment, either in a new kinetic queue, or into an existing one (Fig. 1–a). If there are $n_r$ red segments, the number of red proto-fragments is exactly $n_r + k$. Second, the $R$-economical regularization of the faces defined by the blue segments, (Definition 3.1) further partitions the $n_r + k$ red proto-fragments into *fragments*, at the points where they are cut by the vertical threads. From corollary 3.7, the proto-fragments (which don't intersect the blue segments anymore) are cut in $O((n_r+k)\log n)$ fragments. The merge of two kinetic queues can now be seen as the destruction of all fragments from the smaller one, together with the insertion of equally many *new* fragments into the larger one. Similarly, the split operation involves the deletion of some fragments from one kinetic queue and the creation of a new one made only of *new* fragments.

It is now possible, if we set aside the blue segments, to describe the situation as follows: We have a family $S$ of $O((n_r + k)\log n)$ segments (fragments), partitioned into a number of disjoint subsets $(S_i)_i$, each swept over by a different kinetic queue. Using one of the prescribed kinetic queue implementations, the overall time of insertions, deletions, and internal updates is $\sum_i O(t_Q|S_i|\alpha(S_i)\log|S_i|) = O(t_Q(n_r + k)\alpha(n)\log^2 n)$. The same analysis applies to the cost related to the blue kinetic queues. The maintenance cost of the "interface" between blocks, i.e. the purple events, does not add to the complexity: the scheduling/de-scheduling of a purple event can only be caused by (and charged to) an internal update that modifies the top or bottom of one of the adjacent kinetic queues.

With the bound on $t_Q$ mentioned above, this implies the desired bound on the running time of HeapSweep, which is randomized or deterministic depending on the specific kinetic queue implementation. □

## 5 Applications and Related Results

The algorithm HeapSweep was developed in the context of a theory of polyhedral tracings and their convolution [6]. In a few words, one obtains the convolution of two polyhedra (a red and a blue) by first computing their geometric duals, projecting these duals on the unit sphere, and then computing all pairs of red-blue intersecting features. For non-convex polyhedra, the projection of the dual on the sphere has self-intersections, which do not contribute to the convolution. As this projection is connected, HeapSweep can be used, and it is possible to compute the convolution of two polyhedra of sizes $n_r$ and $n_b$ in output sensitive time $O((n+k)\alpha(n)\log^3 n)$, where $n = n_r + n_b$, and $k$ is the size of the convolution. Amongst other applications, the convolution can be used to obtain the Minkowski sum of two polyhedra.

We mention below some extensions to the main algorithm. The proofs of the stated results are obtained from minor modifications of the above analysis.

**Monotone arcs.** Although the analysis has been performed for line segments for the sake of the exposition, the same algorithm applies for a more general class of arcs. The only difference is the complexity of the upper envelope of a family of arcs, which is then carried through the entire analysis, for both the randomized and the deterministic implementations of a kinetic queue mentioned in this paper. The regularization lemma also holds if line segments are replaced by $x$-monotone arcs.

**Theorem 5.1.** *Let $R, B$ be two sets of $x$-monotone arcs, such that each monochromatic family defines a connected point set on the plane. Assume that two pairs of arcs intersect at most $s$ times. If $n$ is the number of input arcs, the algorithm* HeapSweep *computes all $k$ purple intersections in time $O(\lambda_{s+2}(n+k)\log^3 n)$, where $\lambda_s(m)$ is the length of the longest $(m,s)$-Davenport-Schinzel sequence.*

For instance, algebraic curves of bounded degree obey the requirements mentioned in this theorem (provided they are first decomposed into a constant number of $x$-monotone parts each).

**Several components.** HeapSweep can be used without modification if the blue set has $c_b$ components and the red set has $c_r$ components. In that case, Corollary 3.7 needs to be modified: from each component, a thread needs to be drawn to obtain a simple arrangement, and the algorithm runs in time $O((c_b n_r + c_r n_b + k)\alpha(n)\log n)$. If one point from each component is given, it is possible to compute a low stabbing number spanning tree between these points [23], and thus provide connected input to HeapSweep. The spurious intersections generated by the spanning trees cause a total running time of $O((n_r\sqrt{c_b}+n_b\sqrt{c_r}+k)\alpha(n)\log^3 n)$.

**General case in linear space.** In order to solve the general red-blue intersection problem in linear space, the cuttings technique [20, 1] can be applied, but the running time becomes $O(n^{4/3+\epsilon} + k)$. The $\epsilon$ term can be made as small as desired, but at the cost of a increased hidden constant in the space bounds. Using HeapSweep, we obtain a $O((n^{4/3} + k)\alpha(n)\log^3 n)$ algorithm to report all intersections in the general case, with a linear space cost that has a small implied constant. This can be done by preprocessing the input to identify connected subsets of segments (see [20, 3]), on which it is then possible to use the variation of *HeapSweep* for known components mentioned above. Details are omitted in this paper.
**$j$-level.** If the $j$-level in an arrangement of $n$ arcs has complexity $k$, the algorithm HeapSweep can be adapted to compute it in time $O(\lambda_s(n+k)\log^2 n)$, whether or not the input set is connected, by keeping two kinetic queues grouping all segments above and below the current $j$-level. This generalizes the results of Edelsbrunner and Welzl [18], who computed the $j$-level of an arrangement of line segments in $O((n + k)\log^2 n)$ time using a line sweep technique that can be retroactively be seen as a precursor of HeapSweep. Cole, Sharir, and Yap [13] improved these bounds to $O((n + k)\log^2 j)$.

Both methods mentioned above use the data structure of Overmars and van Leeuven [26] for dynamic maintenance of a convex hull with insertions and deletion cost of $O(\log^2 n)$ per operation. Mark de Berg pointed out that this data structure can also be used in HeapSweep, in replacement of the kinetic queue, giving an improved $O((n + k)\log^3 n)$ algorithm for reporting all $k$ purple intersections between $n$ red and blue line segments [14].
**Kinetic heaps.** We mentioned earlier that the most straightforward idea for implementing a kinetic queue is to use a standard binary heap. If we use a kinetic heap to sweep over an arrangement of $s$ infinite straight lines, then we can prove that the number of internal events processed will be $O(s\log^2 s)$ (the argument is non-trivial). But we were unable to extend this argument to the case of sweeping over a arrangement of line segments, as needed in this paper.

## 6 Conclusions

We have presented a new algorithm, HeapSweep, to report all *purple* intersections between red and blue possibly intersecting line segments on the plane. HeapSweep is a variation of the line sweep method, that stores only a partial ordering of segments on the sweep line. The sweep line is divided into maximal contiguous blocks of monochromatic segments and the segments of each block are stored in a novel data structure called a kinetic queue. A kinetic queue keeps track of the top and bottom segments of a block and allows the detection of purple intersections that can only occur between top and bottom segments of consecutive blocks. If the set of red segments (resp. blue segments) is connected, i.e. their union as a point set is connected, we have proved that HeapSweep reports all purple intersections in expected time $O((n + k)\alpha(n)\log^3 n)$, where $n$ is the input size and $k$ is the number of purple intersections.

The connectedness assumption is interesting as it is not uncommon in practice, and yet there does not seem to be any obvious way to take advantage of it in traditional techniques such as divide-and-conquer, random sampling, and segment trees – since this precious property is not preserved in subsets of the input set. In contrast, HeapSweep makes an interesting use of connectedness, through the regularization lemma, that may find use in other applications.

There are still too many logarithms in the running time of HeapSweep, in particular in the $k$ term. In order to reduce this overhead, we would like to see an integration of our technique (that relaxes the vertical ordering of the sweep line) with the methods of Edelsbrunner and Guibas [16] or Mairson and Stolfi [22] (that relax the horizontal ordering of the sweep). The bounds we obtained also give hope that a filtering search technique [8] coupled with geometric partitioning could achieve an optimal $O(n \log n + k)$ running time (in a sense, HeapSweep is an instance of filtering search, as it detects more intersections than necessary – but not too many more).

There doesn't seem to be any hope of adapting HeapSweep to solve the *counting* problem efficiently. The hereditary segment-tree approach [10] used to obtain an optimal algorithm for counting purple intersections in the disjoint case cannot be adapted either, since segment tree nodes may have a disconnected set of segments. In fact, we believe that connectedness does not simplify the counting problem, and we would like to see an $\Omega(n^{4/3})$ lower bound to substantiate this claim.

**Acknowledgments.** We would like to thank Jan Jannink for suggesting the term "heater", and Chandra Chekuri and Sanjeev Khanna for useful discussions. Special thanks to John Hershberger, who is largely responsible for the deterministic implementation of the kinetic queue with a tournament tree, and to Mark de Berg for suggesting the use of the dynamic convex hull data structure. We also wish to thank an anonymous referee for pointing out the reference [13].

Leonidas Guibas wishes to acknowledge support by NSF Grant CCR-9215219 and US Army Grant 5-23542A during this research.

## References

1. P. K. Agarwal. Partitioning arrangements of lines: II. Applications. *Discrete Comput. Geom.*, 5:533–573, 1990.
2. P. K. Agarwal, M. de Berg, J. Matoušek, and O. Schwarzkopf. Constructing levels in arrangements and higher order Voronoi diagrams. In *Proc. 10th Annu. ACM Sympos. Comput. Geom.*, pages 67–75, 1994.
3. P. K. Agarwal and M. Sharir. Red-blue intersection detection algorithms, with applications to motion planning and collision detection. In *Proc. 4th Annu. ACM Sympos. Comput. Geom.*, pages 70–80, 1988.
4. C. Aragon and R. Seidel. Randomized search trees. In *Proc. 30th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 540–545, 1989.
5. J. Basch, L. J. Guibas, and J. Hershberger. Data structures for mobile data. In preparation.

6. J. Basch, L.J. Guibas, G.D. Ramkumar, and L. Ramshaw. Polyhedral tracings and their convolution. In *Proc. 2nd Workshop on Algorithmic Fundations of Robotics*, 1996.
7. J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Comput.*, C-28:643–647, 1979.
8. B. Chazelle. Filtering search: a new approach to query-answering. *SIAM J. Comput.*, 15:703–724, 1986.
9. B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. *J. ACM*, 39:1–54, 1992.
10. B. Chazelle, H. Edelsbrunner, L. Guibas, and M. Sharir. Algorithms for bichromatic line segment problems and polyhedral terrains. *Algorithmica*, 11:116–132, 1994.
11. B. Chazelle, H. Edelsbrunner, L. Guibas, M. Sharir, and J. Snoeyink. Computing a face in an arrangement of line segments and related problems. *SIAM J. Comput.*, 22:1286–1302, 1993.
12. K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, II. *Discrete Comput. Geom.*, 4:387–421, 1989.
13. R. Cole, M. Sharir, and C. K. Yap. On $k$-hulls and related problems. *SIAM J. Comput.*, 16:61–77, 1987.
14. M. de Berg. Personal communication. 1996.
15. M. de Berg and O. Schwarzkopf. Cuttings and applications. Report RUU-CS-92-26, Dept. Comput. Sci., Utrecht Univ., Utrecht, Netherlands, August 1992.
16. H. Edelsbrunner and L. J. Guibas. Topologically sweeping an arrangement. *J. Comput. Syst. Sci.*, 38:165–194, 1989. Corrigendum in 42 (1991), 249–251.
17. H. Edelsbrunner and E. P. Mücke. Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms. *ACM Trans. Graph.*, 9:66–104, 1990.
18. H. Edelsbrunner and E. Welzl. Constructing belts in two-dimensional arrangements with applications. *SIAM J. Comput.*, 15:271–284, 1986.
19. J. Erickson. New lower bounds for Hopcroft's problem. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages 127–137, 1995.
20. L. Guibas, M. Overmars, and M. Sharir. Intersecting line segments, ray shooting, and other applications of geometric partitioning techniques. In *Proc. 1st Scand. Workshop Algorithm Theory*, volume 318 of *Lecture Notes in Computer Science*, pages 64–73. Springer-Verlag, 1988.
21. S. Hart and M. Sharir. Nonlinearity of Davenport-Schinzel sequences and of generalized path compression schemes. *Combinatorica*, 6:151–177, 1986.
22. H. G. Mairson and J. Stolfi. Reporting and counting intersections between two sets of line segments. In R. A. Earnshaw, editor, *Theoretical Foundations of Computer Graphics and CAD*, volume F40 of *NATO ASI*, pages 307–325. Springer-Verlag, Berlin, West Germany, 1988.
23. J. Matoušek. Spanning trees with low crossing number. *Informatique Théorique et Applications*, 25(2):103–123, 1991.
24. J. Matoušek. Range searching with efficient hierarchical cuttings. *Discrete Comput. Geom.*, 10(2):157–182, 1993.
25. K. Mulmuley. On levels in arrangements and Voronoi diagrams. *Discrete Comput. Geom.*, 6:307–338, 1991.
26. M. H. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *J. Comput. Syst. Sci.*, 23:166–204, 1981.

27. Larry Palazzi and Jack Snoeyink. Counting and reporting red/blue segment intersections. In *Proc. 3rd Workshop Algorithms Data Struct.*, volume 709 of *Lecture Notes in Computer Science*, pages 530–540, 1993.
28. M. Sharir and P. K. Agarwal. *Davenport-Schinzel Sequences and Their Geometric Applications*. Cambridge University Press, New York, 1995.