Discrete Comput Geom (2008) 39: 298–318
DOI 10.1007/s00454-007-9015-0

Improved Output-Sensitive Snap Rounding

John Hershberger

Received: 15 August 2006 / Published online: 13 September 2007
© Springer Science+Business Media, LLC 2007

Abstract This paper presents new algorithms for snap rounding an arrangement $\mathcal{A}$ of line segments in the plane. Snap rounding defines a set of *hot pixels*, which are unit squares centered on the integer grid points closest to the vertices of $\mathcal{A}$. Snap rounding simplifies $\mathcal{A}$ by replacing every input segment by a piecewise linear curve connecting the centers of the hot pixels the segment intersects. Let $\mathcal{H}$ be the set of all hot pixels, and for each $h \in \mathcal{H}$ let $is(h)$ be the number of segments with an intersection or endpoint inside h. If $\mathcal{A}$ contains n input segments, the running time of the first new algorithm is $O(\sum_{h\in\mathcal{H}} is(h) \log n)$. This improves previous input- and output-sensitive algorithms by a factor of $\Theta(n)$ in the worst case. The second algorithm has an even better running time of $O(\sum_{h\in\mathcal{H}} ed(h) \log n)$; here $ed(h)$ is the description complexity of the crossing pattern in h, which may be substantially less than $is(h)$ and is never greater.

Keywords Snap rounding · Robust geometric computation

1 Introduction

Arrangements of line segments in the plane are a central tool in computational geometry [6, 19]. They are often used as building blocks in more complex algorithms, and so the arrangement vertices induced by intersections of the line segments may be used as the basis of further computation. This may lead to robustness difficulties. If two segments are represented with a certain finite precision, approximately double that precision (twice as many bits) will be required to represent the intersection of the segments accurately. If this precision-doubling cascades through several levels of algorithmic building blocks, accurately representing the results of a computation

J. Hershberger (✉)
Mentor Graphics Corp., 8005 SW Boeckman Road, Wilsonville, OR 97070, USA
e-mail: john_hershberger@mentor.com

may require more precision than the machine arithmetic provides, forcing the use of a much slower software arithmetic implementation.

One approach to avoiding high-precision geometric computation is to *round* the vertices of the arrangement to some grid, which by suitable scaling one may take to be the integer grid. If this is done naïvely, it may lead to topological inconsistencies between the rounded and unrounded arrangements. A particularly successful approach to rounding an arrangement is the *snap rounding* scheme introduced by Greene [10] and Hobby [15], which is defined as follows: The plane is divided into unit square *pixels* centered on the integer grid points. Every pixel that contains a segment endpoint or an intersection of two segments is declared to be *hot*. Each segment is replaced by a polygonal path joining the centers of the hot pixels it intersects, in the order of intersection. The union of all the rounded segments defines the rounded arrangement. The rounded arrangement can be regarded as a graph $\mathcal{G} = (\mathcal{H}, \mathcal{E})$, where the nodes are identified with the set of hot pixels $\mathcal{H}$, and the *arcs* of the graph link nodes whose hot pixels are joined by rounded segments.

Each arc of $\mathcal{G}$ may correspond to multiple unrounded segments that in the original arrangement pass in parallel from one hot pixel to the next. Depending on the application of the snap rounded arrangement, it may be important to know the set of original segments associated with each arc of the rounded arrangement.

Guibas and Marimont [11] have shown that snap rounding has many desirable properties: every arc of the rounded arrangement has integer grid points as endpoints, every rounded segment is within half a pixel distance of the corresponding unrounded segment, and the rounded and unrounded arrangements are "topologically equivalent up to the collapsing of features." (As noted by Halperin and Packer [14], rounded arcs may still pass very near hot pixel centers; recent work by Packer [18] shows how to avoid this near-degeneracy while preserving the approximation guarantees of snap rounding.)

This paper focuses on methods to compute a snap rounded arrangement efficiently. The running times of these algorithms depend on several parameters: I is the number of pairwise intersections among all the input segments, $\mathcal{H}$ is the set of hot pixels, $|\mathcal{H}|$ is its size, and for any $h \in \mathcal{H}$, the number of original segments that intersect h is $|h|$. The size of $\mathcal{G}$, not counting edge multiplicities, is $O(|\mathcal{H}|)$, because it is a planar graph. The algorithm of Hobby [15] runs in time $O((n + I)\log n + \sum_{h\in\mathcal{H}} |h|)$. The algorithm of Guibas and Marimont [11] runs in time $O(n \log n + I + \sum_{h\in\mathcal{H}} |h| \log |h|)$, plus another term of usually-smaller magnitude that is specific to their approach. Goodrich et al. [9] avoid the dependence on the input size I with an algorithm that runs in time $O((n + \sum_{h\in\mathcal{H}} |h|) \log n)$; they claim optimality to within a logarithmic factor for their algorithm. However, as Halperin and co-authors have noted [5, 13, 14], the definition of $\sum_{h\in\mathcal{H}} |h|$ as the output size is misleading, because it can be as large as $\Theta(n^3)$, even though the size of the snap rounded arrangement, $\Theta(|\mathcal{H}|)$, can never be larger than $O(n^2)$. See Fig. 1. The discrepancy arises because $\sum_{h\in\mathcal{H}} |h|$ counts each arc of $\mathcal{G}$ with its multiplicity (the number of original segments that round to it). An algorithm of de Berg, Halperin, and Overmars [5] runs in time $O((n + I)\log n)$ and produces $\mathcal{G}$ without representing the segments associated with each arc explicitly. Unfortunately, that algorithm performs poorly on the star arrangement shown in Fig. 2; it runs in $O(n^2 \log n)$ time, whereas the algorithm of Goodrich et al. [9] needs only $O(n \log n)$ time.

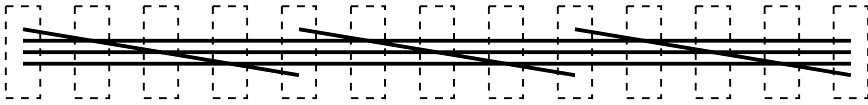

Fig. 1 The algorithm of [9] runs in $\Theta(n^3 \log n)$ on this arrangement because $\sum_{h\in\mathcal{H}} |h| = \Theta(n^3)$, even though $|\mathcal{G}|$ is only $O(n^2)$. (The arrangement is stretched vertically for clarity)

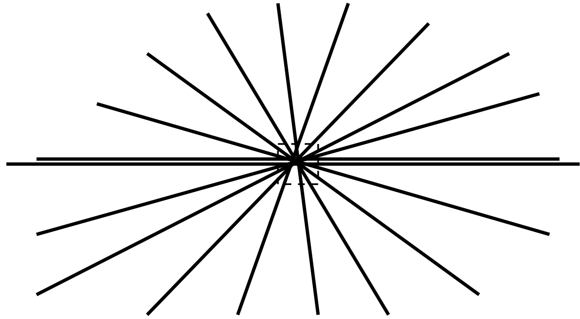

Fig. 2 The algorithm of [5] runs in $\Theta(n^2 \log n)$ on this arrangement, even though $|\mathcal{G}|$ is only $O(n)$, because $I = \Theta(n^2)$

The output an algorithm is required to produce strongly affects its minimum running time. If an algorithm is required to report every segment's intersections with all hot pixels, then the algorithm of [9] is within a logarithmic factor of optimal. Indeed, this is the measure used in that paper's claim of optimality. However, if only the embedded planar graph $\mathcal{G}$ is required, then [9] is far from optimal, as is [5]. See Figs. 1 and 2. Intermediate between these is the goal of producing $\mathcal{G}$ in a data structure that allows a client to report the set of segments associated with any arc of $\mathcal{G}$ efficiently. A further possible requirement on an algorithm is that the set should be spatially ordered and support spatial searches within the set (since no segments in the set intersect between the two hot pixel endpoints of the arc, this is possible).

This paper improves the running time of all previous algorithms. It presents two algorithms, whose running times depend on two new characterizations of the segments incident to a hot pixel. The *intersecting segment count* $is(h)$ is the number of segments that have an endpoint or an intersection inside h. The algorithm of Sect. 3 is very simple to describe and runs in time $O(\sum_{h\in\mathcal{H}} is(h) \log n)$. In particular, it runs in time $O(n^2 \log n)$ and $O(n \log n)$, resp., on the examples of Figs. 1 and 2. The algorithm of Sect. 4 depends on the *edit distance* $ed(h)$, which represents the complexity of the crossover between the segments passing through a hot pixel h. It is always less than $O(is(h))$, and sometimes significantly so. The algorithm of Sect. 4 is more complex than that of Sect. 3, but it runs in $O(\sum_{h\in\mathcal{H}} ed(h) \log n)$ time. Figure 5 in Sect. 4 shows an example for which the two algorithms differ in running time by a factor of $\Theta(\sqrt{n})$.

Both of the algorithms produce the snap rounded arrangement in a form that allows reporting of the original segments associated with each rounded arc. Each arc points to a data structure that records the full set of original segments that round to it, correctly ordered according to the original segment positions; the data structure supports point location among the segments corresponding to each arc. The segment sets are represented using persistent data structures [7], which allow different sets to share

portions of their representations. Although the total size of the sets is $\Theta(\sum_{h\in\mathcal{H}}|h|)$—as large as $\Theta(n^3)$ in some cases—the space needed to represent them is no greater than the algorithm's running time, which is at most $O(n^2\log n)$.

For convenience the algorithms are described assuming exact arithmetic. The necessary primitives can be implemented using fixed precision as in Hobby's algorithm [15].

2 Preliminaries

The input to a snap rounding algorithm is a collection S of n *ursegments* ("ur" can be taken to refer either to "**un**r**ounded" or to the German for "original"). The arrangement of the ursegments, denoted $\mathcal{A}$, has complexity $|\mathcal{A}|$ proportional to n plus the number of intersections between ursegments of S.

A snap rounded arrangement is an embedded planar graph $\mathcal{G}=(\mathcal{H},\mathcal{E})$, where the nodes are identified with the *hot pixels*, a set of unit squares centered on integer grid points. Each pixel is closed at the left and bottom, and open at the top and right. The hot pixels are exactly those pixels that contain a vertex of $\mathcal{A}$. Each ursegment $s\in S$ is snap rounded to a polygonal path that connects the centers of the hot pixels s intersects, in the order of intersection. There is an arc in $\mathcal{G}$ between two hot pixels $h_1,h_2\in\mathcal{H}$ if and only if there is an ursegment $s\in S$ whose snap rounded polygonal path visits h_1 and h_2 consecutively.

Note that many ursegments of S may map to the same arc of $\mathcal{G}$. The set of ursegments associated with an arc e is denoted by *segs*(e). No ursegment crossings or endpoints occur outside hot pixels, so the members of *segs*(e) can be ordered, which may be important for some applications.

A *sweepline* is a data structure that maintains the intersections of an arrangement of segments with a vertical line as the line sweeps over the arrangement from left to right [1, 2, 6]. The sweepline stores a collection of *active segments* in their order of intersection with the vertical line. Each active segment has a *next event*, which is either its right endpoint or its intersection with the segment below it in the sweepline, if that intersection exists to the right of the sweepline. The segments of a sweepline are stored at the leaves of a balanced binary tree, such as a red-black tree [3]. Internal nodes of the tree implement a min-queue on the x-coordinates of the next events in their subtrees: if a node v has children u and w, the value v stores is the minimum of the values stored at u and w. The sweepline tree supports insert, delete, search, split, and concatenate operations in $O(\log n)$ time.

3 Crossing-Segment Sensitivity

This section presents a simple sweepline algorithm for computing $\mathcal{G}$ that is an order of magnitude faster than all previous algorithms, at least on worst-case inputs. The new algorithm can be viewed as a modification of the sweepline algorithm of Goodrich et al. [9]. That earlier algorithm sweeps a vertical line over $\mathcal{A}$, looking for ursegment endpoints or intersections. Whenever it detects one of these *critical points*, it creates a new hot pixel h and performs surgery on the ursegments of S. Every ursegment s

that intersects h is cut into fragments at its crossings with the boundary of h, and the fragment of s inside h is deleted. Four new unit-length segments are inserted on the boundary of h. The fragments of ursegments outside hot pixels terminate on these newly added boundary segments, and all the vertices of $\mathcal{A}$ (all the critical points) are removed by the surgery. The complexity of the modified arrangement is proportional to the number of intersections of ursegments of S with hot pixels in $\mathcal{H}$, and it can be computed using time only a logarithmic factor greater. It is straightforward to extract $\mathcal{G}$ from the modified arrangement in linear time.

Although the algorithm of Goodrich et al. avoids processing potentially costly ursegment intersections by erasing the part of the arrangement inside the hot pixels of $\mathcal{H}$, it also erases parts of ursegments that are intersection-free, leading to the problem illustrated in Fig. 1. A simple remedy suggests itself: Why not erase only the ursegments that are known to have intersections? The new algorithm develops that simple idea. The algorithm computes $\mathcal{G}$ in two phases: it first computes the set of hot pixels $\mathcal{H}$, then computes the arcs that join the hot pixels.

3.1 Computing the Hot Pixels

As in the algorithm of Goodrich et al., the basis of the crossing-sensitive computation of $\mathcal{H}$ is a Bentley-Ottmann sweep [1, 2, 6] over $\mathcal{A}$. The algorithm assumes no ursegment is vertical, although this can be enforced if necessary by an infinitesimal symbolic rotation of vertical ursegments. A vertical sweepline passes over the ursegments of S, and the algorithm maintains a sorted list of the ursegments intersecting the sweepline in vertical order. A priority queue maintains the next event to the right of the sweepline—as noted in Sect. 2, the sweepline data structure itself (a balanced binary tree) serves as a priority queue for the next event involving an ursegment in the current active set. Events have four types: ursegment left and right endpoints, ursegment intersections, and ursegment *re-insertions*. The first three are standard, but the fourth is a feature of the algorithm: an ursegment that has an intersection inside a hot pixel h is removed from the sweepline and scheduled for re-insertion at the point where it crosses out of h. In essence, this modifies the set of segments swept over by the sweepline so that the fragments derived from any ursegment have at most one intersection per hot pixel.

The algorithm uses a subroutine $\mathit{trim}(s, h)$ that operates on an ursegment s and a hot pixel h it intersects. On entry to $\mathit{trim}(s, h)$, ursegment s is present in the sweepline. The subroutine removes s from the sweepline, then computes the intersections of s with the boundary of h. If s has a fragment that lies to the right of its intersection with the interior of h, then $\mathit{trim}(s, h)$ schedules s for re-insertion into the sweepline at the left endpoint of that fragment. Recall that the bottom boundary of h is contained in h (because h is closed on its left and bottom sides). Thus if s exits h through the bottom, the re-insertion happens infinitesimally after the crossing, so that s is re-inserted *after* it leaves h.

Here is the algorithm to compute the hot pixel set $\mathcal{H}$. For purposes of this algorithm, a hot pixel is represented as an (x, y) pair of integers denoting the center of the pixel, and the algorithm stores pixels in a set *HPSet*. The algorithm obtains integers from the real coordinates of intersections and endpoints using a function $\mathit{round}(r) \equiv \lfloor r + \frac{1}{2} \rfloor$. Thus if $\bar{r} = \mathit{round}(r)$, $r \in [\bar{r} - \frac{1}{2}, \bar{r} + \frac{1}{2})$.

Algorithm FINDHOTPIXELS:

$HPSet \leftarrow \emptyset$;
Initialize the event queue for the Bentley-Ottmann sweep.
while the event queue is nonempty do
  Remove the next event e from the queue.
  Let (x_e, y_e) be the coordinates of e.
  Advance the sweepline to x_e.
  If e is not a re-insertion then
    $HPSet \leftarrow HPSet \cup (round(x_e), round(y_e))$
  If e is a right endpoint of ursegment s then
    Remove s from the sweepline.
  Else if e is a left endpoint or a re-insertion of ursegment s then
    Insert s into the sweepline.
  Else $\{e$ is an intersection of ursegments s_1 and $s_2\}$
    $h = (round(x_e), round(y_e))$;
    $trim(s_1, h)$; $trim(s_2, h)$;

As in a standard Bentley-Ottmann sweep, any modification of the sweepline contents (insertion or deletion of a segment) causes the next events for those segments and their neighbors to change, and the modified x-coordinates of those events are propagated up the tree, so that each node records the x-value of the current leftmost event in its subtree. The following lemmas establish the correctness and runtime performance of FINDHOTPIXELS:

Lemma 3.1 *Algorithm* FINDHOTPIXELS *correctly computes all hot pixels in $\mathcal{G}$.*

Proof Because the algorithm recognizes a hot pixel only for ursegment endpoints or intersections, the set *HPSet* it computes is a subset of the hot pixels in $\mathcal{G}$. To argue that every hot pixel is added to *HPSet*, note that subsegments of an ursegment s are removed by $trim(s, h)$ only inside a known hot pixel h. The Bentley-Ottmann sweep algorithm detects all intersections between untrimmed ursegments. If a hot pixel contains no ursegment endpoints (i.e., it is made hot only by ursegment intersections), then at least one of the ursegment intersections it contains will be detected, because the participating ursegments will not be trimmed before the pixel is known to be hot. □

Define the *intersecting segment count* $is(h)$ to be the number of ursegments of S that have an endpoint or an intersection inside a pixel h. Note that $is(h)$ is *not* the number of intersections inside h. In fact, the number of ursegment intersections inside h may be as large as $\binom{|h|}{2} = \Theta(|h|^2)$, while $is(h)$ is never larger than $|h|$. Furthermore, $is(h)$ may be much less than $|h|$, which lends significance to the following result:

Lemma 3.2 *The running time of* FINDHOTPIXELS *is* $O(\sum_{h\in\mathcal{H}} is(h)\log n)$.

Proof An ursegment s is trimmed by $trim(s, h)$ after the first intersection that FINDHOTPIXELS detects for s inside h. Therefore the algorithm processes at most one intersection for each ursegment/pixel pair. The subsequent re-insertion event can be

charged to the $trim(s, h)$ operation that scheduled it. The algorithm performs work for an ursegment s only if it has an intersection or an endpoint inside h, and the number of operations for each ursegment is $O(1)$ per pixel. Each operation involves $O(1)$ standard binary tree operations on the sweepline and the event queue, which take $O(\log n)$ time apiece. □

3.2 Computing the Arcs of $\mathcal{G}$

This section presents an algorithm for computing the arcs of $\mathcal{G}$. The algorithm runs in the same time as FINDHOTPIXELS, and represents $segs(e)$ for each arc e using a persistent data structure. The core of the algorithm is a slight modification of the method of de Berg, Halperin, and Overmars [5]; a clean-up phase takes care of one special case that algorithm does not fully handle.

The algorithm of [5] assumes that the hot pixels have already been identified and computes the arcs between them using two Bentley-Ottmann sweeps. The first sweep processes ursegments with nonnegative slopes, and the second (symmetric) sweep processes those with negative slopes. The algorithm assumes that no ursegment is vertical, though this restriction is easy to enforce using a symbolic perturbation, if necessary.

This section presents the algorithm of [5] in some detail, because variations on this algorithm are important both here and in Sect. 4.3. The algorithm is based on a Bentley-Ottmann sweep over the ursegments of S with nonnegative slopes. The sequence of ursegments intersecting the sweepline is divided into subsequences (*bundles*) defined by the hot pixels that the ursegments intersect immediately to the left of the sweepline. A bundle is a maximal subsequence of ursegments with a single hot pixel predecessor. Bundles are recorded compactly in the tree representing the sweepline as follows (this detail differs from the algorithm in [5]): Call a node in the tree *pure* if all of its leaf descendants have the same predecessor hot pixel. Each maximal pure node (the node is pure, but its parent is not) is labelled with the identity of the hot pixel predecessor. Thus only $O(\log n)$ nodes in the tree are involved in labelling each bundle, and all these nodes are children of a path of length $O(\log n)$ in the tree.

The algorithm processes the hot pixels left-to-right, grouped in vertically aligned columns, from bottom-to-top within each column. The sweepline, instead of being a single vertical line, is a staircase consisting of at most five segments: below the current hot pixel h it coincides with the right side of the column, above h it coincides with the left side of the column, and within h a vertical segment sweeps from left to right. See Fig. 3. For each hot pixel h the algorithm FINDARCS performs the following steps:

1. Find the subsequence of ursegments in the sweepline that intersect h. The segments of the sweepline are properly ordered along the staircase profile, so this is a simple tree search.
2. For every bundle that hits h, create an arc e of $\mathcal{G}$ joining h to the bundle's predecessor hot pixel, and use persistence [7] to record the subset of the bundle that hits h (a subsequence in the current sweepline) as $segs(e)$. This can be done in $O(\log n)$ time per arc created.

Fig. 3 The sweepline in the algorithm of [5] is a staircase with five segments

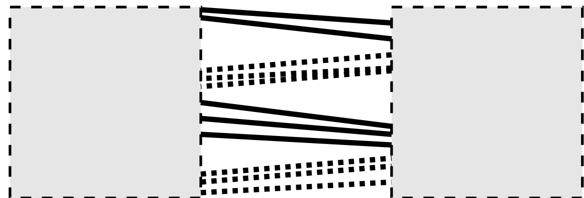

Fig. 4 The positive- and negative-slope ursegments for the arc joining these two hot pixels are stored in two separate sequences, one shown as dashed segments and one shown solid. Their interleaving is undetermined

3. Of the bundles that hit h, at most two may hit it with only a fraction of the bundle segments. One of these two partially passes below h, and the other above; the two rôles may even be filled by the same bundle. Split these (up to) two bundles and label the portion of each that misses h with the same label (the same predecessor) that it had originally.
4. Propagate the ursegments that hit h through the hot pixel, and insert or delete any ursegments with endpoints inside h. In the original paper [5] this propagation is done using a standard Bentley-Ottmann sweep in $O(\mathit{int}(h)\log n)$ time, where $\mathit{int}(h)$ is the number of ursegment intersections and endpoints inside h. It is straightforward to replace this step by a sweep that calls $\mathit{trim}(s, h)$ at the first intersection of an ursegment s, thereby reducing the time to $O(\mathit{is}(h)\log n)$. Note that no effort is needed to propagate the ursegments that do not hit the hot pixels. Their order is the same on both sides of the column.
5. Label all the ursegments that exit h on its top and right sides as a single bundle.

This algorithm finds all the arcs of $\mathcal{G}$ in $O(\sum_{h\in\mathcal{H}} \mathit{is}(h)\log n)$ time, and for every arc produces at most two sequences of ursegments (one for each sweep) that contain all the ursegments that belong to the arc. If the arc is not horizontal or vertical, it is discovered by only one of the two sweeps, and its sequence contains the ursegments in the order they appear in $\mathcal{A}$. If the arc is horizontal or vertical, the nonnegative-slope and negative-slope ursegments that define it are discovered in two separate sweeps, and recorded in two separate sequences. Each sequence is correctly ordered, but their possible interleaving is undetermined. See Fig. 4.

If it is important to represent every arc by a single properly ordered sequence of ursegments, this can be accomplished using two more sweeps, one horizontal and one vertical. Each sweep is responsible for creating bundles for the arcs perpendicular to the sweepline. The horizontal sweep, for example, passes a vertical sweepline over the arrangement as in algorithm FINDHOTPIXELS. When the sweep reaches the right side of a hot pixel h, it labels the bundle of ursegments that emerge from the right side of h with its predecessor h, as in algorithm FINDARCS. Because the algorithm is

not interested in ursegments that emerge from the tops or bottoms of h—they cannot contribute to a horizontal arc of $\mathcal{G}$—it is able to perform the labelling in a single logarithmic-time operation per hot pixel. When the sweep reaches the left side of a hot pixel h, it checks whether any of the ursegments that hit the left side of h emanate from a hot pixel h' straight left of h. If so (a logarithmic-time test), the subsequence that originates at h' and hits h can be identified and recorded using persistence as $segs(e)$, for $e = (h', h)$, in $O(\log n)$ time.

This completes the proof of the following theorem:

Theorem 3.3 *Given a set S of n ursegments, its snap-rounded arrangement $\mathcal{G} = (\mathcal{H}, \mathcal{E})$ can be computed in time $O(\sum_{h\in\mathcal{H}} is(h) \log n)$, where $is(h)$ is the number of ursegments of S that have endpoints or intersections inside a hot pixel h. The ursegment sequences associated with the arcs of $\mathcal{E}$ can be computed and recorded within a matching time and space bound.*

4 Edit Distance Sensitivity

Although the algorithm of the preceding section is a substantial improvement over both [5] and [9], it still seems suboptimal for some inputs. Consider the example shown in Fig. 5. In the figure, there are $\sqrt{2n}$ bundles containing $\sqrt{n/2}$ parallel ursegments apiece, with the bundles arranged in a grid. The total number of hot pixels is $2\sqrt{2n} + n/2$, and the total number of ursegment intersections is $n^2/4$. For each hot pixel h determined by ursegment intersections, $is(h) = \sqrt{2n}$ and $int(h) = n/2$. For this input, the algorithm of [5] runs in $O(n^2 \log n)$ time, and both [9] and the algorithm of Sect. 3 run in $O(n\sqrt{n} \log n)$ time. Nevertheless, it seems that the algorithms are missing an opportunity for efficiency, because the intersection pattern in each hot pixel is particularly simple. If one could take advantage of this simplicity, one could reduce the processing time further. The improved algorithm presented in this section achieves a running time of $O(n \log n)$ for the example in Fig. 5.

4.1 Edit Distance

Our intuition tells us that the crossover inside each hot pixel of Fig. 5 is simple. This section formalizes that intuition using the notion of *edit distance*, in particular

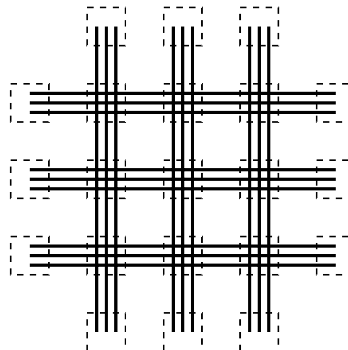

Fig. 5 In this grid of bundles, $\sum_{h\in\mathcal{H}} is(h) = \Theta(n\sqrt{n})$, although the crossover in each hot pixel is very simple

Fig. 6 A is transformed into B by three editing operations

$$
\begin{array}{ll}
A: & a\ b\ c\ d\ e\ f\ g\ h \\
 & \quad\ \ \text{insert } i \\
 & a\ b\ \not{c}\ d\ i\ e\ f\ g\ h \\
 & a\ b\ d\ i\ \underbrace{e\ f\ g}\ h \\
B: & a\ e\ f\ g\ b\ d\ i\ h
\end{array}
$$

edit distance with moves [4, 20]. Consider two sequences of symbols A and B, such that each symbol appears at most once in each of A and B. (That is, the sequences are *nonrepeating*.) The sequence A can be transformed into B by a series of *editing operations* of the following three types: insert a symbol at any position, delete a symbol at any position, and move a subsequence of symbols from any position in the sequence to some other position. See Fig. 6 for an example.

If the sequence is stored in a doubly linked list and pointers to the locations of operations are provided, each of these operations takes $O(1)$ time; if it is stored in a balanced binary tree, each takes $O(\log(|A| + |B|))$; if it is stored in a finger search tree [12, 16], then insert/delete take $O(1)$ amortized time and the move operation takes $O(\log(\ell + d))$, where ℓ is the length of the subsequence and d is the distance it moves.

The number of editing operations needed to transform one sequence into another is the edit distance between the sequences. However, this quantity may be difficult to calculate; an equivalent but easier-to-compute metric is the *neighbor difference distance*, defined to be the number of symbols in A and B whose neighbors are not identical in the two sequences. In Fig. 6 the neighbor difference distance is 8, because only f has its neighbors unchanged.

Lemma 4.1 *The edit distance and the neighbor difference distance between two nonrepeating sequences are equal to within a constant factor.*

Proof Each editing operation affects $O(1)$ neighbors. Thus there is a constant c such that the neighbor difference distance is at most c times the edit distance. On the other hand, if the neighbor difference distance between two sequences is d, then the sequences can be partitioned into $O(d)$ subsequences and singleton elements that can be rearranged with $O(d)$ editing operations to transform one sequence into the other. □

Because the edit distance and the neighbor difference distance are constant-factor equivalent, this paper uses the more euphonious name *edit distance* to refer to the easier-to-compute *neighbor difference distance*. That is, the edit distance $ed(A, B)$ between sequences A and B is actually computed as the neighbor difference distance.

The concept of edit distance for sequences carries over easily to the setting of a sweepline. Two different positions x_1 and x_2 of the sweepline induce two different sequences of ursegment intersections. Each ursegment is viewed as a symbol in the sequences, and the edit distance between sweepline positions x_1 and x_2, denoted

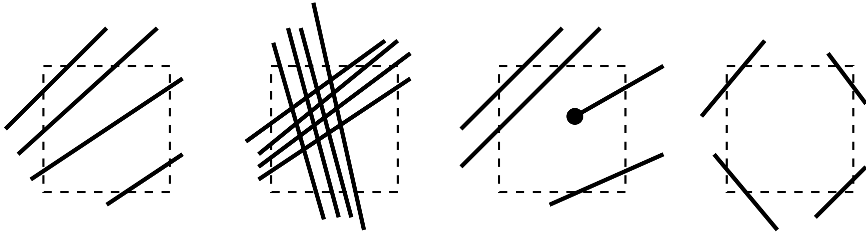

Fig. 7 The edit distances of these pixels in left-to-right order are 0, 4, 3, and 4(!)

$ed(x_1, x_2)$, is the number of ursegments whose neighbors along the sweepline differ at the two positions. If x_1 and x_2 are the left and right sides of a column of pixels C, then $ed(C) \equiv ed(x_1, x_2)$.

Extending the concept of edit distance to the sequence of ursegments crossing the boundary of a pixel is trickier, but possible. The complication arises because the sequence of ursegment intersections with the boundary is circular, and there is only one sequence, not two. Each ursegment that intersects a pixel h without having an endpoint inside it crosses the boundary of h twice. (An ursegment that is tangent to the boundary of h is defined to have zero or two intersections, depending on whether the boundary is open or closed at the point(s) of tangency.) The two intersections of such a *pass-through* ursegment s with the boundary of h are labelled s and s'. Intuitively, the edit distance of a pixel with no intersections or endpoints inside it should be zero. If a group of parallel ursegments passes through a pixel h without intersecting, then for every triple abc of ursegment boundary crossings that appears in counterclockwise order, the sequence must also contain the triple $c'b'a'$. Therefore the *edit distance of a pixel* h, $ed(h)$, is defined to be the number of ursegments that intersect h and either (a) have an endpoint inside, or (b) have neighbor pairs at the two boundary crossings that are not mirror reflections of each other. See Fig. 7. Ursegments with exactly one endpoint inside h cause a contribution of type (b) to $ed(h)$, so $ed(h)$ could also be defined in terms of the sequence at the boundary of h plus a term for the trivial ursegments fully contained in h.

Note that in the last example of Fig. 7, the edit distance is 4 even though there are no intersections or endpoints inside the pixel. This is somewhat of an anomaly, but it can be justified because such a configuration arises only if there are intersections or endpoints of the implicated ursegments in an adjacent pixel. The following lemmas characterize the pixel edit distance:

Lemma 4.2 *If a pixel h has no ursegment endpoints or intersections inside it, then $ed(h) = O(1)$.*

Proof The ursegments crossing h partition the interior of h into faces. Each face is a convex polygon, and all its vertices are either corners of h or intersections between ursegments and the boundary of h. The boundary of a face consists of an alternating sequence of ursegments and portions of the boundary of h. Each side of h appears at most once on the boundary of each face. An ursegment s contributes to $ed(h)$ if and only if at least one of the two faces it bounds has more than two ursegments on its

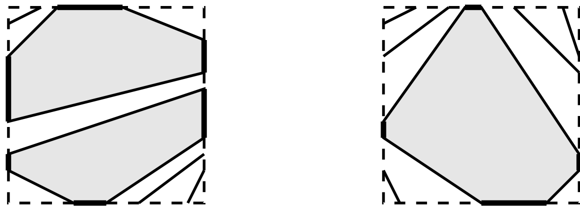

Fig. 8 A pixel containing no ursegment endpoints or intersections has edit distance at most 6

boundary. If a face has more than two ursegments on its boundary, it must also have portions of at least as many sides of h. Because the faces are interior-disjoint there can be at most two faces with three ursegments on their boundaries, or at most one face with four ursegments on its boundary. This proves the lemma. See Fig. 8. □

Lemma 4.3 *For any hot pixel* h, $ed(h) = O(is(h))$.

Proof Let T be the set of ursegments counted in $is(h)$. Estimate $ed(h)$ by removing all ursegments in T, then adding them back one by one. After all ursegments in T are removed from h, $ed(h) = O(1)$, by Lemma 4.2. Adding the ursegments back one by one increases $ed(h)$ by at most $O(1)$ per addition, because each new ursegment has at most four neighbors on the boundary of h. After all ursegments have been added, $ed(h) = O(1 + |T|) = O(is(h))$. □

Lemma 4.4 *For any hot pixel* h, $ed(h) > 0$.

Proof If any ursegment has an endpoint inside, the ursegment contributes to $ed(h)$, by definition. Otherwise, if there is an intersection inside h, then either the intersecting ursegments are neighbors on the boundary, or by induction on the number of ursegments separating them along the boundary there exists some pair of intersecting ursegments that *are* neighbors on the boundary. These two ursegments clearly have different neighbors on the opposite sides of h, and therefore contribute to $ed(h)$. □

The concept of edit distance for pixels can be generalized to convex regions, though the generalization of Lemma 4.2 holds only for constant-complexity convex polygons. The edit distance of a convex region R, $ed(R)$, is the number of ursegments that intersect R and either (a) have an endpoint inside, or (b) have neighbor pairs at the two boundary crossings that are not mirror reflections of each other. If there are no ursegments fully contained in a column C, then the convex region definition of edit distance is equivalent to $ed(C)$. The following lemma helps relate $ed(C)$ to $ed(h)$ for the hot pixels $h \in C$:

Lemma 4.5 *Let R be a convex region that is partitioned by a line ℓ into two convex fragments R' and R''. By convention assume that $\ell \cap R$ is included in at least one of R' and R''. Then*

$$ed(R) \leq ed(R') + ed(R'') + O(1).$$

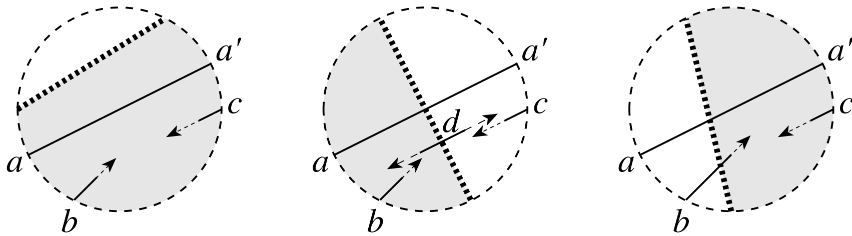

Fig. 9 Ursegment a is counted in the edit distance for at least one of the fragments R' and R'' created by the partitioning line

Proof It suffices to show that all but $O(1)$ segments that are counted in $ed(R)$ are counted in at least one of $ed(R')$ and $ed(R'')$. This is clear for segments with an endpoint in R, because the endpoint lies in R' or R''.

Consider a pass-through segment a, and without loss of generality suppose that the counterclockwise neighbor of crossing a is b and the clockwise neighbor of crossing a' is $c \neq b'$. Further assume that ℓ does not separate either of the pairs (a, b) or (c, a'). See Fig. 9. Without loss of generality suppose that R' contains (a, b). If R' also contains (c, a'), then a is counted in $ed(R')$. If (c, a') belongs to R'', then consider the intersections of ursegments a and b with ℓ. If crossings with b and a do not occur along ℓ in counterclockwise order in R', then a is counted in $ed(R')$; if they do, then the configuration of a in R is replicated in R'', and a is counted in $ed(R'')$. Because at most four ursegment crossings are separated from their neighbors along the boundary of R by ℓ, the lemma holds. □

The sequence edit distance for columns is related to the pixel edit distance as follows:

Lemma 4.6 *Let C be a column of pixels containing at least one hot pixel, and let $\mathcal{H} \cap C$ denote the set of hot pixels in C. Then*

$$ed(C) = O\left(\sum_{h \in (\mathcal{H} \cap C)} ed(h)\right).$$

Proof The proof of Lemma 4.2 extends trivially to show that if a rectangle R has no ursegment endpoints or intersections inside it, then $ed(R) = O(1)$. Let R_C be the convex hull of all the pixels in C that are crossed by ursegments. As observed earlier, $ed(C) = O(ed(R_C))$. Now partition R_C at the boundaries of the hot pixels into $|\mathcal{H} \cap C|$ hot pixels and at most $|\mathcal{H} \cap C| + 1$ rectangles containing no ursegment endpoints or intersections. By Lemma 4.5 and the extension of Lemma 4.2 to rectangles, $ed(R_C) = O(\sum_{h \in (\mathcal{H} \cap C)} ed(h)) + O(|\mathcal{H} \cap C|)$. By Lemma 4.4 this is $O(\sum_{h \in (\mathcal{H} \cap C)} ed(h))$. □

4.2 Edit-Distance-Sensitive Sequence Transformation

This section tells how to transform one sequence into another in time proportional to the edit distance times a logarithmic factor, given the availability of certain primitive operations that are easy to implement in the sweepline setting.

Consider the problem of transforming a sequence A into a second sequence B, assuming that A is stored at the leaves of a balanced binary tree, and that the transformation is implemented by destructive surgery on the tree. Define $n = \max(|A|, |B|)$, and assume the existence of a comparison operator $<_B$, which when applied to two elements $a, b \in B$ returns true if and only if a appears to the left of b in B. Further suppose that the input to the problem identifies all elements that appear in only one of A and B, and all consecutive pairs $a, b \in A$ such that $b <_B a$. The following algorithm dismantles A and constructs B from the resulting fragments:

Algorithm EDITDISTTRANSFORM:

Delete from A all elements in $A \setminus B$.
Break A into subsequences $A_1, A_2, \ldots, A_k$ by splitting at all the deletion sites and between every neighbor pair a, b such that $b <_B a$.
{Each subsequence is correctly ordered in both A and B, and the ends of each subsequence are counted in the edit distance $ed(A, B)$.}
Set $T \leftarrow \emptyset$, an empty sequence.
for $i \leftarrow 1$ to k do
$T \leftarrow merge(T, A_i)$; {Now $\forall a, b \in T$, $a <_B b$}
Insert into T all elements in $B \setminus A$.
return T;

The subroutine $merge(T, A_i)$ merges two sorted sequences into one in time $O(\log n)$ times the number of fragments into which A_i is subdivided. It is an unsophisticated variant on an algorithm of Hwang and Lin [17] and can be expressed recursively as follows:

merge(U, V)
if *empty*(U) then return V;
else if *empty*(V) then return U;
else if $head(V) <_B head(U)$ then
return $merge(V, U)$;
{Now $head(U) <_B head(V)$}
Split U into U' and U'' at $head(V)$.
return $concat(U', merge(V, U''))$;

The split and concatenate operations take $O(\log n)$ time apiece and all other operations take constant time. The total number of operations is proportional to the number of contiguous fragments into which the input sequences U and V are decomposed.

Lemma 4.7 *Algorithm* EDITDISTTRANSFORM *runs in time* $O(ed(A, B) \log n)$.

Proof The number of operations the algorithm performs outside the *merge*() calls is proportional to $ed(A, B)$. Each operation is a standard binary search tree operation and takes $O(\log n)$ time. Each call to $merge(T, A_i)$ takes time $O(t \log n)$, where t is the number of fragments into which A_i is split by the *merge*() subroutine. If A_i is split into t fragments, that means that $t - 1$ pairs of adjacent symbols in A_i are separated in B, and therefore the total time spent in *merge*() is $O(ed(A, B) \log n)$. □

Note that EDITDISTTRANSFORM is essentially an adaptive algorithm for sorting a partially pre-sorted sequence, where the measure of disorder is the edit distance. Other such adaptive sorting algorithms are described in [8].

4.3 Computing the Hot Pixels

This section shows how to provide the input data and comparison operator required by the algorithm EDITDISTTRANSFORM for the ursegment sequences determined by two positions of the sweepline. It follows that it is possible to advance the sweepline from position x to another position x' in time $O(ed(x, x') \log n)$. It seems more difficult to perform a similar operation for the ursegments incident to a hot pixel, largely because it is hard to determine which ursegments are entering and which are exiting; indeed, ursegments may both enter and exit (during a left-to-right sweep) through the top and bottom of a hot pixel. Therefore the algorithm presented here finesses the issue, reducing the problem to a series of cases in which algorithm EDITDISTTRANSFORM *is* applicable.

Hot pixels determined by ursegment endpoints are easy to detect; the algorithm focuses on finding hot pixels determined only by intersections. These fall into two classes: (a) *same-side* pixels in which two ursegments cross the same side of the pixel and intersect inside and (b) *cruciform* pixels in which every intersecting pair consists of one ursegment crossing the top and bottom of the pixel and another crossing the left and right sides. The two classes of hot pixels are detected separately.

Lemma 4.8 *If a hot pixel h contains no ursegment endpoints and there exist two ursegments that cross the same edge of h and intersect inside h, then there are two ursegments that intersect inside h and are adjacent along the specified edge.*

Proof The proof is by induction on the number of ursegments that separate the two chosen ursegments along the boundary of h. Suppose that ursegments a and b cross an edge e of h and intersect inside h. Ursegments a, b, and the edge e bound a triangle inside h. If a and b are not adjacent along e, then there exists some ursegment s that crosses e between them and intersects either a or b (say a) on the triangle boundary. Then a and s both cross e, intersect inside h, and are closer along e than a and b. By induction the lemma follows. □

As noted in Sect. 2, the binary tree implementing the sweepline stores a next-event x-value for each ursegment crossing the sweepline, and the nodes of the tree implement a tournament on these x-values. Thus it is possible to find in $O(t \log n)$ time all t active segments whose next scheduled event occurs left of any desired x-value. (Note that if an ursegment s has event x-value $\bar{x}$, that does not necessarily imply that the first event involving s occurs at $\bar{x}$; that claim is true only for the ursegment with the leftmost event x-value. What is true is that an ursegment with x-value $\bar{x}$ will have an event at or before $\bar{x}$, assuming the neighbor defining the event is not deleted first.) The t ursegments with events left of a given x-value $\bar{x}$ partition the sweepline into at most $t + 1$ subsequences with the property that each subsequence, considered in isolation, has no events left of $\bar{x}$. That is, the vertical order of each subsequence is the

same at $\bar{x}$ as at the current sweepline position. This partition into event-free subsequences is part of the input needed by algorithm EDITDISTTRANSFORM; the other part, the comparison operator, simply checks the intersection order of two ursegments with the vertical line $x = \bar{x}$.

The following algorithm finds all same-side pixels whose defining pair of ursegments crosses the left pixel boundary:

Algorithm EDITDISTSAMESIDE:

HPSet $\leftarrow \emptyset$;
Initialize the endpoint queue and the sweepline.
while the endpoint queue is nonempty do
  Let $\bar{x}$ be the x-coordinate of the next event (either an endpoint or an intersection).
  Let $x_h = round(\bar{x})$.
  {The sweepline currently holds ursegments in proper order for $x_h - \frac{1}{2}$.}
  Find all ursegments in the sweepline with a scheduled event before $x_h + \frac{1}{2}$.
  For each scheduled event e with $x_e < x_h + \frac{1}{2}$ do
    $HPSet \leftarrow HPSet \cup (x_h, round(y_e))$;
  Apply EDITDISTTRANSFORM to advance the sweepline from $x_h - \frac{1}{2}$ to $x_h + \frac{1}{2}$, updating the endpoint queue as necessary.

It follows from Lemmas 4.6, 4.7, and 4.8 that EDITDISTSAMESIDE finds all hot pixels determined by intersecting ursegments that enter through the left side in time $O(\sum_{h\in\mathcal{H}} ed(h)\log n)$. Applying the algorithm four times, once for each cardinal direction, finds all hot pixels except the cruciform pixels. In fact, it is enough to apply the algorithm twice: algorithm EDITDISTTRANSFORM detects every ursegment whose sweepline neighbors at $x_h - \frac{1}{2}$ differ from its neighbors at $x_h + \frac{1}{2}$, and this means that one invocation of EDITDISTSAMESIDE can detect all same-side hot pixels determined by ursegments crossing either their left or right sides.

The algorithm for detecting cruciform pixels is loosely based on the algorithm of [5] for arc computation described in Sect. 3.2. If the arc computation is omitted, that algorithm can be used to find intersections between pairs of positive-slope ursegments and pairs of negative-slope ursegments, but not between pairs with one positive and one negative slope. The extension described here depends on the observation that it is not the slope of the ursegments that is important for the algorithm, but how the ursegments cross the boundary of a hot pixel. In particular, ursegments are allowed to enter a hot pixel only through its left and bottom sides, and allowed to exit only through its right and top sides. This is automatically true for positive-slope ursegments, but it can be enforced for ursegments of all slopes by modifying the algorithm. In brief, the algorithm EDITDISTCRUCIFORM described below processes all the ursegments together, but gives special treatment to ursegments that enter a hot pixel h through its top boundary. Such ursegments are not processed inside h, but are instead merged directly into the sequence at the right boundary of the current column.

The algorithm EDITDISTCRUCIFORM processes hot pixels column-by-column from left to right. A hot column is identified by an ursegment endpoint or by an ursegment intersection event detected by the sweepline. Within each column C of hot pixels, the algorithm repeatedly identifies and processes the lowest unprocessed

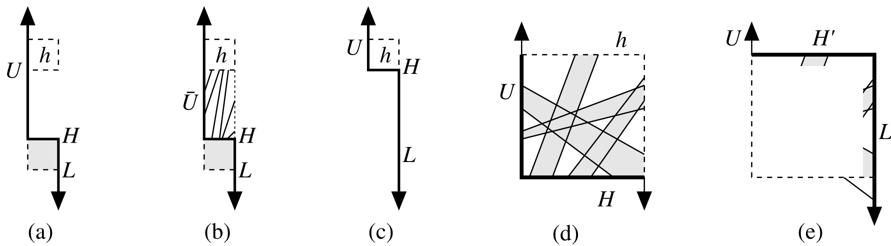

Fig. 10 **a** Configuration at the start of each step of EDITDISTCRUCIFORM. **b** Let $\bar{U}$ be the portion of U below the next hot pixel h. Separate the non-crossing ursegments in $\bar{U} \cup H$ into those that hit h and those that do not (those that cross the thin dashed segment in the figure). **c** Configuration before processing hot pixel h. **d** Split the sequence of ursegments entering h through its bottom and left sides into subsequences with no intersections inside or below h. Further split each subsequence into ursegments that cross the top of H and those that do not. Note that some ursegments from U may cross through the bottom of h. **e** Merge the subsequences at the top of h to create H'; merge the other subsequences into L

hot pixel. At the beginning of each step the sweepline profile is a stairstep, with one horizontal segment at the top of the previously processed hot pixel in C. See Fig. 10a. The sweepline is partitioned into three portions U, H, and L. U and L are the upper and lower vertical portions, with U at the left side of C and L at the right; H is the horizontal segment joining U and L. All ursegments associated with H have positive slope (they pass from below H to above it). As the algorithm runs, ursegments are removed from U and added to L; ursegments are both added to and removed from H, and the y-coordinate associated with H increases.

The ursegments crossing H (the top of the previous hot pixel) and U below the next hot pixel h form a contiguous subsequence on the sweepline. These ursegments have no events scheduled in the region between H and h (else h would be lower). Thus they are intersection-free in that region, and a single tree search separates the subsequence that hits the bottom of h from the one that reaches the right edge of C. (See Fig. 10b.) The portion that reaches the right edge of C is merged into L, and the portion that reaches h becomes the new contents of H; H moves up to the bottom of h (Fig. 10c). To process h, the algorithm examines the subsequence of $U \cup H$ that crosses the left and bottom sides of h, and splits it into subsequences with no scheduled events in or below h. Each of these subsequences consists of ursegments that pass through h without any neighbor intersections, starting from the bottom and left sides of h. Each subsequence is further split into one piece that hits the top of h and one that does not (Fig. 10d). The portions that hit the top are merged to create a new sequence H' to replace H; the portions that do not hit the top are merged into L. In the process, H is emptied and the part of U that crosses the left edge of h is deleted (Fig. 10e). Note that ursegments from U that hit the bottom of h receive no further processing in C below h. Any intersections those segments have in C below h are not detected. The propagation of U and H through h is very similar to the processing of Algorithm EDITDISTTRANSFORM, except applied on a pixel-by-pixel basis. The key to applying that algorithm is the separation between the source sequences (U and H) and the destination sequences (H' and L).

To find the lowest pixel above H containing a scheduled event, and to support the splitting of H and U into subsequences with no scheduled event inside or below a pixel h, the algorithm uses a quantized and enhanced version of the sweepline

described in Sect. 2. Each leaf node (representing an ursegment) has an associated event location (x_e, y_e), as before, but internal nodes store both coordinates of an event. Among the events in its subtree with minimal $round(x_e)$ values, each node stores the one with minimum y_e. That is, the events are grouped into columns of pixels, and a node selects the lowest event in the leftmost event-containing column. This value can be computed in constant time at each node based on the values stored at the node's children.

Observation 4.9 *The quantized sweepline data structure described above stores at its root the lowest scheduled event in the leftmost column of pixels containing an event.*

Lemma 4.10 *If all the ursegment events stored in a quantized sweepline lie in or to the right of a column of pixels C, then the sweepline can be used to find all scheduled events lying in or below a pixel $h \in C$ in $O(\log n)$ time apiece.*

Proof The search rule is simple: if and only if an internal node being visited stores an event in the query region, visit the node's children. Because each node stores an event belonging to one of its leaves, it is clear that if the children are visited, then the subtree contains an event of interest. But conversely, if the children are not visited, then the subtree contains no event of interest: if a node v's event location is (x_v, y_v) and the center of h is (x_h, y_h), then if $round(x_v) > x_h$ all descendants of v are right of C, and if $y_v \geq y_h + \frac{1}{2}$ then all descendants of v in C are above h. For each event found, the algorithm visits all the ancestors of the leaf containing the event and all the ancestors' siblings, for a total cost of $O(\log n)$ per event reported. □

Lemma 4.11 *If a cruciform pixel h contains a nonnegative-slope ursegment that passes through h from bottom to top, the algorithm* EDITDISTCRUCIFORM *detects an event inside h.*

Proof By a slight modification of the proof of Lemma 4.8 one can show that there exists a pair of ursegments that are adjacent along U and H (one on U and one on H) that intersect inside h. This pair defines an intersection event inside h that will be detected no later than the processing of the hot pixel below h. □

Note that EDITDISTCRUCIFORM does *not* detect all hot pixels. In particular, all ursegments that enter a pixel from the top are merged directly into the sweepline L, and so any cruciform pixels determined only by such ursegments will not be detected. See Fig. 11.

Lemma 4.12 *The total running time of algorithm* EDITDISTCRUCIFORM *is $O(\sum_{h\in\mathcal{H}} ed(h) \log n)$.*

Proof The algorithm spends $O(\log n)$ time to detect each hot pixel that it finds and to process the part of C between successive hot pixels. Within each hot pixel h the algorithm spends $O(ed(h) \log n)$ time to propagate H and the relevant part of U across h and into H' and L, as in Lemma 4.7. A further term of $O(ed(C) \log n)$

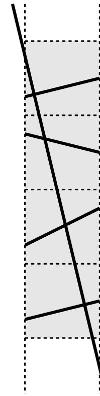

Fig. 11 Each of the shaded pixels is hot, but only the top one will be found by the positive-slope instance of EDITDISTCRUCIFORM. The others are found by the negative-slope instance

covers the cost of merging into L the portions of U that exit through the bottom of h for all hot pixels $h \in C$. Because $ed(C) = O(\sum_{h\in(\mathcal{H}\cap C)} ed(h))$ by Lemma 4.6, this term is dominated by the sum of the per-pixel costs. □

By Lemma 4.11, two applications of Algorithm EDITDISTCRUCIFORM, one each for positive and negative slopes, suffice to find all cruciform hot pixels. Combining this with two applications of EDITDISTSAMESIDE finds all hot pixels in a total of $O(\sum_{h\in\mathcal{H}} ed(h)\log n)$ time.

4.4 Computing the Arcs of $\mathcal{G}$

A relatively straightforward extension of the algorithms of Sect. 3.2 computes the arcs of $\mathcal{G}$ and their associated ursegment sets in $O(\sum_{h\in\mathcal{H}} ed(h)\log n)$ time. The step in the algorithm of de Berg, Halperin, and Overmars that processes a hot pixel h is modified, as in the previous subsection, to advance a quantized sweepline over h in time $O(ed(h)\log n)$. The algorithm is somewhat simpler than that in Sect. 4.3, however, because it does not need to handle negative-slope ursegments. Likewise, if the clean-up phase needs to perform additional sweeps to compute the ursegment sets of horizontal and vertical arcs, it uses the method of algorithm EDITDISTSAMESIDE to sweep over each column C of hot pixels in time $O(ed(C)\log n)$.

These observations, plus the algorithms of Sect. 4.3, establish the following theorem:

Theorem 4.13 *Given a set S of n ursegments, its snap-rounded arrangement $\mathcal{G} = (\mathcal{H}, \mathcal{E})$ can be computed in time $O(\sum_{h\in\mathcal{H}} ed(h)\log n)$, where $ed(h)$ is the edit distance of a pixel h. The ursegment sequences associated with the arcs of $\mathcal{E}$ can be computed and recorded within a matching time and space bound.*

The edit distance of every hot pixel where bundles cross in Fig. 5 is $O(1)$, and so the running time of this algorithm applied to that set of ursegments is $O(\sqrt{n}\log n)$ per endpoint-containing pixel and $O(\log n)$ per intersection-containing pixel, for a total time of $O(n\log n)$.

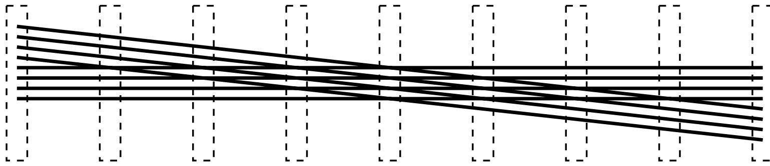

Fig. 12 $\sum_{h\in\mathcal{H}} ed(h) = \Theta(n^2)$, although $\mathcal{G}$ has linear size. (The arrangement is stretched vertically for clarity)

5 Conclusion

The algorithm of Sect. 3 is a practical addition to the available algorithms for computing snap rounded arrangements of line segments. It avoids the excessive running times of previous algorithms by a simple idea that leads to a simple algorithm.

The algorithm of Sect. 4 is primarily of theoretical interest because it requires several independent sweeps over the ursegments. Nevertheless, it points the way toward a new class of snap rounding algorithms that depend on the edit distance of the hot pixels. If an algorithm is required to produce output that represents the ursegments associated with each arc of $\mathcal{G}$ in a sorted sequence, bounds depending on the pixel edit distance are arguably the best possible. Reordering the sequence of ursegments entering a pixel to obtain the sequence of ursegments exiting seems to require a number of operations proportional to the edit distance.

If the order of ursegments associated with arcs of $\mathcal{G}$ is unimportant, some improvement may still be possible. Consider the arrangement of ursegments shown in Fig. 12, in which most of the hot pixels have edit distance $\Theta(n)$, and $\sum_{h\in\mathcal{H}} ed(h) = \Theta(\sum_{h\in\mathcal{H}} is(h)) = \Theta(\sum_{h\in\mathcal{H}} |h|) = \Theta(I) = \Theta(n^2)$. For this arrangement all the known algorithms run in time $\Omega(n^2 \log n)$, even though $\mathcal{G}$ has size $O(n)$ and the unordered value of $segs(e)$ is the same for every arc in $\mathcal{G}$.

References

1. Bentley, J.L., Ottmann, T.A.: Algorithms for reporting and counting geometric intersections. IEEE Trans. Comput. **C-28**(9), 643–647 (1979)
2. Brown, K.Q.: Comments on “Algorithms for reporting and counting geometric intersections”. IEEE Trans. Comput. **C-30**, 147–148 (1981)
3. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edn. MIT Press, Cambridge (2001)
4. Cormode, G., Muthukrishnan, S.: The string edit distance matching problem with moves. In: Proc. 13th ACM-SIAM Sympos. Discrete Algorithms, pp. 667–676, 2002
5. de Berg, M., Halperin, D., Overmars, M.: An intersection-sensitive algorithm for snap rounding. Comput. Geom. Theory Appl. **36**, 159–165 (2007)
6. de Berg, M., van Kreveld, M., Overmars, M., Schwarzkopf, O.: Computational Geometry: Algorithms and Applications, 2nd edn. Springer, Berlin (2000)
7. Driscoll, J.R., Sarnak, N., Sleator, D.D., Tarjan, R.E.: Making data structures persistent. J. Comput. Syst. Sci. **38**, 86–124 (1989)
8. Estivill-Castro, V., Wood, D.: A survey of adaptive sorting algorithms. ACM Comput. Surv. **24**, 441–476 (1992)
9. Goodrich, M., Guibas, L.J., Hershberger, J., Tanenbaum, P.: Snap rounding line segments efficiently in two and three dimensions. In: Proc. 13th Annu. ACM Sympos. Comput. Geom., pp. 284–293, 1997
10. Greene, D.H.: Integer line segment intersection (unpublished manuscript)

11. Guibas, L., Marimont, D.: Rounding arrangements dynamically. In: Proc. 11th Annu. ACM Sympos. Comput. Geom., pp. 190–199, 1995
12. Guibas, L.J., McCreight, E., Plass, M., Roberts, J.: A new representation for linear lists. In: Proc. 9th Annu. ACM Sympos. Theory Comput., pp. 49–60, 1977
13. Halperin, D.: Problem 1: Output sensitive algorithm for snap rounding, June 2005. Open Problem Session: 21st Annu. ACM Sympos. Comput. Geom.
14. Halperin, D., Packer, E.: Iterated snap rounding. Comput. Geom. Theory Appl. **23**, 209–225 (2002)
15. Hobby, J.D.: Practical segment intersection with finite precision output. Comput. Geom. Theory Appl. **13**(4), 199–214 (1999)
16. Huddleston, S., Mehlhorn, K.: A new data structure for representing sorted lists. Acta Inform. **17**, 157–184 (1982)
17. Hwang, F.K., Lin, S.: A simple algorithm for merging two disjoint linearly ordered sets. SIAM J. Comput. **1**(1), 31–39 (1972)
18. Packer, E.: Iterated snap rounding with bounded drift. In: Proc. 22nd Annu. ACM Sympos. Comput. Geom., pp. 367–376, 2006
19. Sharir, M., Agarwal, P.K.: Davenport–Schinzel Sequences and Their Geometric Applications. Cambridge University Press, New York (1995)
20. Tichy, W.F.: The string-to-string correction problem with block moves. ACM Trans. Comput. Syst. **2**(4), 309–321 (1984)