A Simple Dynamization of Trapezoidal Point Location in Planar Subdivisions

Milutin Brankovic
University of Sydney, Australia
mbra7655@uni.sydney.edu.au

Nikola Grujic
University of Sydney, Australia
ngru0072@uni.sydney.edu.au

André van Renssen
University of Sydney, Australia
andre.vanrenssen@sydney.edu.au

Martin P. Seybold
University of Sydney, Australia
mpseybold@gmail.com

Abstract

We study how to dynamize the Trapezoidal Search Tree (TST) – a well known randomized point location structure for planar subdivisions of kinetic line segments.

Our approach naturally extends incremental leaf-level insertions to recursive methods and allows adaptation for the online setting. The dynamization carries over to the Trapezoidal Search DAG (TSD), which has linear size and logarithmic point location costs with high probability. On a set S of non-crossing segments, each TST update performs expected $\mathcal{O}(\log^2 |S|)$ operations and each TSD update performs expected $\mathcal{O}(\log |S|)$ operations.

We demonstrate the practicality of our method with an open-source implementation, based on the Computational Geometry Algorithms Library, and experiments on the update performance.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases Dynamization, Trapezoidal Search Tree, Trapezoidal Search DAG, Backward Analysis, Point Location, Planar Subdivision, Treap, Order-maintenance

Digital Object Identifier 10.4230/LIPIcs.ICALP.2020.18

Category Track A: Algorithms, Complexity and Games

Related Version `https://arxiv.org/abs/1912.03389`

Supplementary Material `https://github.com/milutinB/dynamic_trapezoidal_map_impl`

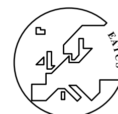
Funding *Martin P. Seybold*: Supported under the Australian Research Council Discovery Projects funding scheme (project number DP180102870).

Acknowledgements We want to thank the University of Sydney's undergraduate Research Internship Program, which fostered the pursuit of this topic.

1 Introduction

Our results on vertical ray shooting in dynamic planar subdivisions and dynamic order maintenance originate from the search for a simple and practical data structure, with guarantees, that facilitate building the intersection graph of thin geometric objects, which is a natural precursor in meshing fracture networks [14]. This note studies how to structure 2D kinetic and non-kinetic line segment data in a fully-dynamic online setting that supports

© Milutin Brankovic, Nikola Grujic, André van Renssen, and Martin P. Seybold;
licensed under Creative Commons License CC-BY



intersection reporting. Beside sweeping plane applications, such data structures are also of interest for map matching problems [22] and for point location queries in dynamic, constrained Delaunay triangulations [4].

Most works term the "dynamic point location problem" as to quickly find the first edge directly above or below a query point, which is vertical ray shooting queries over a dynamic set of edges. We also use this unfortunate terminology throughout, though there are works (e.g. [18]) that solve the seemingly more complicated problem of deciding if two query points are in the same face of the dynamic subdivision. Planar subdivisions are often categorized by additional geometric properties on the subdivisions' faces. Authors frequently consider cases where all faces are convex, or all faces intersect horizontal lines at most twice (monotone faces). Clearly, monotone faces and faces' boundaries formed by monotone curves (e.g. line segments) are distinct properties.

Trapezoids are a key ingredient to simplify the study of geometric problems concerning sets of line segments S. We use three well known [2, 11, 15, 20] and closely related [13] structures. These are (i) the trapezoidal decomposition $\mathcal{A}(S)$ of a plane induced by S, (ii) the Trapezoidal Search Tree (TST), and (iii) the Trapezoidal Search DAG (TSD) (see Section 2). TST $\mathcal{T}(S, \pi)$ and TSD $\mathcal{D}(S, \pi)$ stem from deterministic algorithms, that incrementally insert the segments in S according to a, typically random, permutation π over S. Decomposition $\mathcal{A}(S)$ has size $\mathcal{O}(|S| + k)$, where k denotes the number of *crossing pairs* of segments in S. Since segment boundaries of a general planar subdivision are non-crossing ($k = 0$), the refined decomposition $\mathcal{A}(S)$ of the boundary segments S is within a constant of the subdivision's size, regardless of the face shape.

Chiang and Tamassia's 1992 survey [9, Chapter 6] reviews several dynamizations for planar subdivisions. In [10], the two authors describe a dynamization for the special case of a trapezoidal decomposition $\mathcal{A}$ of monotone subdivisions. Using several tree structures, they achieve $\mathcal{O}(\log |S|)$ point location query time within an $\mathcal{O}(|S| \log |S|)$ size structure, which allows fully-dynamic updates in amortized $\mathcal{O}(\log^2 |S|)$ operations. Also for monotone subdivisions, Goodrich and Tamassia show with [12] how to maintain interlaced, monotone spanning trees over the edges of the planar graph and its dual graph. This leads to an $\mathcal{O}(|S|)$ sized structure with an update time of $O(\log |S| + m)$, for insertion of a monotone chain of length m, but point location queries take $\mathcal{O}(\log^2 |S|)$ operations. Cheng and Janardan [8] show how to achieve equal bounds for more general, connected subdivisions.

Most recent approaches that achieve *online, fully-dynamic updates* within $\mathcal{O}(|S|)$ size and $\mathcal{O}(\log |S|)$ query time are extensions of the works of Baumgarten et al. [5] that uses dynamic fractional cascading to enable vertical ray shooting queries. Arge et al. [3] show a trade-off improvement, i.e. raising the fan-out of tree nodes to $\Theta(\log^\varepsilon |S|)$ allows to lower the point location query time bound to $\mathcal{O}(\log |S|)$. However, insertions take $\mathcal{O}(\log^{1+\varepsilon} |S|)$ amortized time and deletions take $\mathcal{O}(\log^{2+\varepsilon} |S|)$ amortized time per update. Chan and Nekrich [7] added Multi-Colored Segment Trees to the approach and applied de-randomization and de-amortization techniques to finally derive update bounds of $\mathcal{O}(\log^{1+\varepsilon} |S|)$ for both, insertions and deletions. To our knowledge, these are currently the best bounds on the pointer machine (see Table 1 in [7] for an overview). Refining this approach, Munro and Nekrich [17] recently analyzed the I/O counts in an external memory model with block sizes of $\Omega(\log^8 n)$.

The *Randomized Incremental Construction (RIC)* of TSDs provides simple, and thereby practical, point location structures of expected $\mathcal{O}(|S|)$ size whose longest search path is with high probability also in $\mathcal{O}(\log |S|)$. Their analysis is a celebrated result of Mulmuley [15], who argues on a random experiment that draws geometric objects, and Seidel [20], who uses a backward-analysis to bound the expected costs. Vastly unmentioned is Mulmuley's book [16],

which extends TSDs to a specialized (offline) dynamic setting of a random insert/delete string for a fixed set of segments S, in which each element has a probability of $1/|S|$ to appear in each update request. The approach thrives on a randomly chosen, fixed order on S, rotations in the structure, and regular leaf-level insertions and deletions. Several distributed book chapters finally show that executing such a (+/-)-sequence of m updates has a total expected cost of $\mathcal{O}(m \log m + k \log m)$, where k is the number of intersections in the fixed set S. Schwarzkopf [19] independently describes a similar update model that also leads to a dynamization. His analysis provides an expected $\mathcal{O}(\log^2 n)$ time bound for update operations from the sequence, where n denotes the current number of non-crossing segments, but the expected point location query bound is w.h.p. in $\mathcal{O}(\log^2 n)$.

In [2], Agarwal et al. describe a TST over a static set of segments, that move over time. Their approach is based on a RIC of TSTs with expected $\mathcal{O}(|S| \log |S| + k)$ size whose depth is w.h.p in $\mathcal{O}(\log |S|)$. The authors mention a bound on the expected construction time of $\mathcal{O}(|S| \log^2 |S| + k \log |S|)$ and show that randomization is crucial to resolve a structural change, due to kinetic (adversarial) movement of the line segments, in expected $\mathcal{O}(\log |S|)$ operations. One may, however, form constant velocity trajectories for (point-like) segments that lead to $\Omega(n\sqrt{n})$ many topological updates for any binary space partition method [1].

Insightful work of Hemmer et al. [13] recently revealed a certain bijection between the search paths of $\mathcal{T}(S, \pi)$ and the "un-folded" search paths of $\mathcal{D}(S, \pi)$ by means of a structural induction argument (along fixed π) on the two incremental algorithms. The authors show Las Vegas type point location guarantees for the RIC of TSDs by bounding computation time of the length of a longest search path in a TSD by means of *the size* of the related TST.

1.1 Paper Organization and Section Contributions

Our surprisingly simple, thereby practical (c.f. Section 6), approach uses only one search structure and performs insertions and deletions directly on higher levels of it. Since we maintain randomness of the priority orders on the segment sets, the expected size and expected point location query bounds are retained for each state of the structures. Section 2 reviews the basic RIC of TSTs and TSDs in a unified notation and we introduce our natural, recursive extensions of those basic primitives in Section 3.

Our algorithms therein enable fully-dynamic updates in both data structures in the offline setting, i.e. the complete ground set of segments S is known in advance. Our extension to the online setting provides a simpler, randomized solution to the dynamic order-maintenance problem with expected update cost of $\mathcal{O}(\log |S|)$ based on an arithmetic coding scheme on Treaps (c.f. Section 5). Known solutions to this problem provide $\mathcal{O}(1)$ amortized update costs [24] or deterministic worst case $\mathcal{O}(\log^2 n)$ update cost based on de-amortization techniques [6]. Our algorithms are suitable for pointer machines with arithmetic since they only compare the spatial location of the input points, intersection points, and priority values. Our need of constant time arithmetic operations on fixed-point numbers with $\mathcal{O}(\log |S|)$-bits is solely due to the online order-maintanance problem. In particular, the method does not require indirect addressing of Random Access Memory.

To our knowledge, we provide the first analysis for the update costs of an *online fully-dynamic* data structure that originates from the classic RIC method. Though our Section 4 argues for non-crossing segments only, our geometric interpretation of TST nodes provides a 4-ply covering property that enables new proof methods. We show a new form of Backward-Analysis on the permutation space and a bound on simple "segment searches" for affected search nodes in the structure (avoiding maintenance of so-called conflict lists or $\mathcal{A}$). For TSTs, we obtain expected update costs of $\mathcal{O}(\log^2 |S|)$ for insertion and deletion. Which we

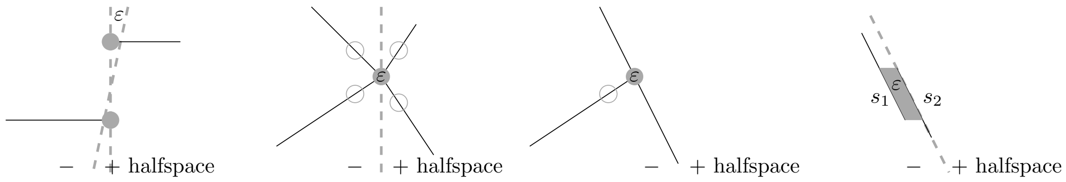


Figure 1 Standard shear transformation (left) and three refined tie-breaking rules for point/point meets, point/segment meets (geometric), and segment/segment overlaps (lexicographic).

can improve further for TSDs to an expected update cost of $\mathcal{O}(\log |S|)$ for insertion and deletion. Hence randomization allows improvements on the amortized deletion bound of $\mathcal{O}(\log^{2+\varepsilon} |S|)$ in [3] and on the $\mathcal{O}(\log^{1+\varepsilon} |S|)$ deterministic update bounds in [7], without losing the size or the point location query bounds.

2 Basic Definitions, Algorithms and Properties

A segment $s = \overline{pq}$ between two distinct points $p, q \in \mathbb{R}^2$ is the set $\{p + \alpha(q - p) : \alpha \in [0, 1]\}$ of points in the Euclidean plane. Two segments are called disjoint if $s \cap s' = \emptyset$ and *overlapping* if $s \cap s'$ contains more than one point. For the case of one common point, we say that two segments *meet* if it is an endpoint that is contained in the other segment and otherwise they are called *intersecting* (or *crossing*). The segment boundaries of a general planar subdivision, for example, have no intersections, but may well contain points in which arbitrarily many segments meet. A line, parallel to the y-axis, through a point is called a *vertical-cut* (e.g. through an end or intersection point) and the line through the endpoints of a segment is called an *edge-cut*. To avoid ambiguity, we denote the open halfspaces of a cut with "−" and "+". For vertical-cuts − denotes the one with lower x-coordinates and for edge-cuts − denotes the one with the lower values of the y-axis.

As in [11, Chapter 6], our presentation assumes that no two distinct points (end or intersection) have exactly the same x-coordinate, unless they are a common endpoint in which the segments meet. As usual, this assumption is resolved by an implicit, infinitesimal shear transformation which translates to a tie-breaking rule in the geometric orientation predicate that resolves to comparing the y-coordinates. A simple extension of the halfspace partitioning for the remaining cases (of segments and cuts due to a segment) conceptually moves the points infinitesimally away from the degeneracy along their segment for a consistent decision. To also assign co-linear cases consistently we finally resolve to lexicographic comparison (c.f. Figure 1). We also assume that the whole domain is bounded by a very large rectangle, which is also a trapezoid. Extensions to unbounded problems are commonly achieved by careful case treatment of unbounded faces of arrangements (e.g. in [23]).

Our combinatorial analysis (c.f. Section 4) of the proposed dynamization requires more precise terms to establish a geometric correspondence, which we introduce now in a unified review of the well known incremental algorithms. We frequently identify a permutation π from the set of permutations $\mathbf{P}(S)$ over S as bijection $\pi : S \to \{1, \ldots, |S|\}$ and call $\pi(s)$ the *priority* of a segment $s \in S$.

2.1 Trapezoidal Search Trees $\mathcal{T}(S, \pi)$

A TST over a trapezoidal domain $\Delta \subseteq \mathbb{R}^2$ is a certain hierarchical Binary Space Partition with vertical cuts and edge cuts induced by the segments in S. These rooted trees are defined by inductively applying the deterministic insertion algorithm for the segments of S in *ascending priority* order. Hence, the structure is uniquely determined by (S, π).

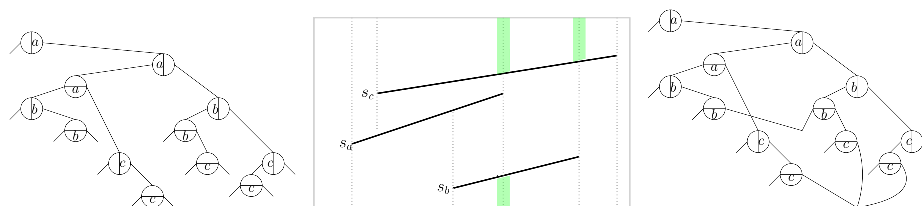


Figure 2 Example of TST Leaf-Insert (left) and TSD Leaf-Insert (right) for segments with priority order $s_a < s_b < s_c$ (middle). The vertical merges of Algorithm 2 are indicated in green.

Every node v of the binary tree is associated with a trapezoidal region $\Delta(v) \subseteq \Delta$ and, on the empty set of segments, $\mathcal{T}(\{\}, ())$ contains only the root r with $\Delta(r) = \Delta$. In TSTs, every non-leaf node v has two child nodes v^-, v^+ and stores a cut that signifies the halfspace partition of $\Delta(v)$ in $\Delta(v^-)$ and $\Delta(v^+)$, which enables point-location search descents. We say that this cut *destroys* the trapezoidal region $\Delta(v)$ and use this to extend the priority assignment to also comprise tree nodes.

We denote the priority of node v with $p(v)$. More precisely, leaves have priority $+\infty$ and non-leaf nodes have the priority of the segment whose cut insertion destroys the node. E.g. the cuts that constitute the (3 or 4) boundaries of $\Delta(v)$ have smaller priorities than $p(v)$.

Algorithm 1 Leaf-Insert($\mathcal{T}, s$).

1. Search for the leaf nodes $L = \{u_1, \ldots, u_l\} \subseteq \mathcal{T}$ with $\Delta(u_i) \cap s \neq \emptyset$.
2. Create refined slabs of these regions by vertically partitioning each $\Delta(u_i)$ with cuts due to endpoints of s or intersection points of s with edge-cuts bounding $\Delta(u_i)$.
3. Partition the intersected slab regions further with the edge cut through s.

See Algorithm 1 for the steps to insert the next segment s in a TST $\mathcal{T}$. Note that nodes' priorities are monotonically increasing on paths from the root. Procedure Leaf-Insert inserts $3, 2$ or 1 cuts on the region of a leaf and the edge-cut is always the last cut that is performed. We refer to these patterns as vertical-vertical-edge (VVE), vertical-edge (VE), and edge (E) destruction. To remove unnecessary ambiguity in the tree structure, we use the additional convention that in a VVE-destruction, the left of the two vertical cuts is inserted first (e.g. as parent of the right vertical cut). See Figure 2 for an example.

▶ **Theorem 2.1** (TST size and depth [2]). *Let S be a set of segments and k_S the number of intersecting pairs among them. The expected size of a TST over S is bounded by*

$$\mathop{\mathbb{E}}_{\pi \in \mathbf{P}(S)} |\mathcal{T}(S, \pi)| = \mathcal{O}(|S| \log |S| + k_S) \ .$$

The expected leaf depth is $\mathcal{O}(\log |S|)$ and the maximum leaf depth of $\mathcal{T}$ is w.h.p. $\mathcal{O}(\log |S|)$.

Clearly, any TST over a set of segments S contains at least $\Omega(|S| + k_S)$ nodes and there are certain instances that may have size $\Omega(|S|^3)$. Agarwal et al. [2] mention an upper bound of $\mathcal{O}(|S| \log^2 |S| + k_S \log |S|)$ expected time for the purely incremental construction.

2.2 Trapezoidal Decomposition $\mathcal{A}(S)$ and Search DAG $\mathcal{D}(S, \pi)$

One may save space in such a search structure by considering a certain planar subdivision $\mathcal{A}(S)$ that is induced from a set of segments S. Given aforementioned discussion of degeneracies, this subdivision is defined by emitting two vertical rays (in negative and positive y-direction)

from each end or intersection point until the ray meets the first segment or the bounding rectangle. Each face of $\mathcal{A}(S)$ is a trapezoid with 3 or 4 boundary edges and standard double counting establishes a size of $\mathcal{O}(|S| + k_S)$ for these planar decompositions. Note that $\mathcal{A}(S)$ is independent of segment priorities.

Mulmuley [15] and Seidel [20] consider the incremental process of inserting an additional segment s and obtaining $\mathcal{A}(S \cup \{s\})$ from $\mathcal{A}(S)$. Algorithm 2 shows the additional merging phase, to "contract" pre-existing vertical-cuts (between the regions of the nodes in L) towards their emission point until they meet s, to maintain the property that the leaf regions coincide with $\mathcal{A}(S)$. Hence, the planar subdivision of the leaves of a TST, i.e. $\{\Delta(v) \subseteq \mathbb{R}^2 : v \in \mathcal{T}(S, \pi), v \text{ is leaf}\}$, is a *refinement* of $\mathcal{A}(S)$, regardless of π.

Algorithm 2 Leaf-Insert($\mathcal{D}$, s).

---

1. Search for the leaf nodes $L = \{u_1, \dots, u_l\} \subseteq \mathcal{D}$ with $\Delta(u_i) \cap s \neq \emptyset$.
2. Create refined slabs of these regions by vertically partitioning each $\Delta(u_i)$ with cuts due to endpoints of s or the intersection points of s with edge-cuts bounding $\Delta(u_i)$.
3. Partition the intersected slab regions further with the edge cut through s.
4. Scan over the vertical cuts between nodes in L that cross s.
   Merge the vertical cuts of the refined regions whose emission point is now blocked by s.

Though the resulting search graph is a DAG (also known as History DAG), the basic data stored with a node is still identical with the binary TST tree nodes (e.g. Figure 2). One key ingredient for the well known, expected construction time of $\mathcal{O}(|S| \log |S| + k_S)$, is to perform the search for affected leaves (c.f. Step 1) quickly. A way to achieve this, without prior knowledge of all future segment insertions, is to also maintain an explicit graph representation of the planar subdivision $\mathcal{A}(S)$ of the segments S that are currently contained in the structure. This trick allows to perform only one point location query for, e.g., the left endpoint of s, followed by a walk along s through $\mathcal{A}(S)$. Our Section 3.1 solves this differently. We summarize the well known aspects of this incremental algorithm in the following statement.

▶ **Theorem 2.2** (TSD size and depth [11, 15, 20]). *Let S be a set of segments and k_S the number of intersecting pairs among them. The expected size of a TSD over S is bounded by*

$$\mathop{\mathbb{E}}_{\pi \in \mathbf{P}(S)} |\mathcal{D}(S, \pi)| = \mathcal{O}(|S| + k_S) .$$

The expected search path length is $\mathcal{O}(\log |S|)$ and the maximum search path length of $\mathcal{D}$ is w.h.p. $\mathcal{O}(\log |S|)$.

The TST and TSD structure also allow simple deletions of segments in a certain decremental setting, that deletes all segments in *descending* priority order. That is, the modifications that the structure undergoes in the steps of Algorithm 2 are simply undone in exactly the reverse order (steps 4 to 2), causing the same amount of work. This is what Seidel's classic backward-analysis, that analyses dropping the last element instead of appending it, exploits.

The insightful work of [13], recently revealed a certain bijection between the search paths of $\mathcal{T}(S, \pi)$ and the (valid) search paths in $\mathcal{D}(S, \pi)$ by means of a structural induction argument (along fixed π) on the two incremental algorithms. This allows them to bound the runtime of computing the length of a longest search path in a TSD $\mathcal{D}(S, \pi)$ by means of the size of the related TST $\mathcal{T}(S, \pi)$. Based on this technique, our analysis of the update time in TSTs (c.f. Section 4) carries over to an upper bound in TSDs as well.

The main geometric difference between the structures is that in TSTs the region of a node is always a subset of its parents' region, whereas in TSDs the region of a node may extend further to the left and right, due to vertical merges, but not across the top and bottom boundaries. We now describe our recursive algorithms for vertical partitions, vertical merges, edge partitions, and edge merges that operate on intermediary priority levels.

3 Recursive Primitives for Dynamic Updates

Characteristic for our approach is that insertions and deletions are performed directly on higher levels of the search structures rather than solely on leaves. We first describe the recursive primitives for inserting a new segment s in the structure of $\mathcal{T}(S,\pi)$.

We choose a random position $p \in \{1,\dots,|S|+1\}$ uniformly in which we emplace the element s, between $(p-1)$ and p, in the sequence π, calling the resulting priority order π'. Our recursive algorithms then update the structure $\mathcal{T}(S,\pi)$ exactly to $\mathcal{T}(S\cup\{s\},\pi')$. More precisely, we restrict the search (c.f. Step 1 of Algorithms 1 and 2) to those nodes with priority smaller than p, which leads to a set L of affected subtree roots whose priorities are larger than p in π'. We conceptually "hang out" these nodes by creating a copy L' of them and reverting those in L to leaves, temporarily. The insertion then proceeds in the same sequence as on regular leaves, but every binary space partition on nodes of L is accompanied by a matching call with the recursive primitive on the respective subtree of the node's copy in L'. After this process, we have a matching root in L' for each leaf that was created below L, which we then "hang in" instead of the simple leaf. Given the close relation of the two structures, it turns out that our recursive primitives, with few changes, already provide the necessities for dynamic updates of TSDs.

This and the following section assume that priority value comparisons, i.e. $\pi'(s') > \pi'(s)$, are decidable in constant time. Section 5 shows a new and simple solution, based on Treaps, to solve the dynamic order maintenance problem sufficiently fast for the online setting. Given the discussion above on the possible E-, VE- and VVE-destruction patterns, we simplify notation in this section by denoting with $\mathrm{Descend}(v)$ the tuple of $0,2,3$, or 4 descendants with the next higher priority value than $p(v)$. E.g. $\mathrm{Descend}(v) = ()$ if v is a leaf, $\mathrm{Descend}(v) = (v_a, v_b)$ if v underwent an E-destruction, $\mathrm{Descend}(v) = (v_l, v_a, v_b)$ or $\mathrm{Descend}(v) = (v_a, v_b, v_r)$ if v underwent a VE-destruction, and $\mathrm{Descend}(v) = (v_l, v_a, v_b, v_r)$ if v underwent a VVE-destruction. With v_l, v_a, v_b and v_r we denote the respective left, above, below and right child of the node's destruction (e.g. Figure 3).

3.1 Priority Restricted Searches

We use the following top-down refinement process to derive the node set L, such that its nodes u_i are sorted by the sequence in which s stabs $\Delta(u_i)$ from left-to-right: First locate the search node u, with maximal $p(u) < p(s)$, that fully contains s in $\Delta(u)$. Place u in an initially empty list. Successively replace the leftmost node u in the list with $p(u) < p(s)$ with the nodes of $\mathrm{Descend}(u)$, whose region intersects s, until all node priorities exceed $p(s)$.

Note that nodes with priority larger than $p(s)$ are not refined and the spatial location of the regions of the nodes in $\mathrm{Descend}(\cdot)$ allows us to easily keep the set L sorted by left-to-right stabbing sequence of s.

3.2 Recursive Vertical Partitions and Merges

Our methods are inspired by sorting algorithms on lists of integers, though they move cuts and child relations among nodes based on the nodes' priorities. We first introduce the recursive primitive V-Partition(u, q, v^-, v^+), where $q \in \mathbb{R}^2$ denotes the point that induces the vertical cut $c(q)$, u is an affected tree node, and v^-, v^+ the roots of the respective, initially empty, result trees. As outlined above, the primitives update the search structure to the state that regular leaf insertion (in ascending priority order) would have created under the presence of vertical cut $c(q)$ splitting $\Delta(u)$ in $\Delta^- \cup \Delta^+$. (A lengthy, rigorous proof fixes $\Delta(u)$ and performs a structural induction argument over the sequence of successive cuts that destroy the region.) See Figure 3 for the recursive cases by destruction patterns and Algorithm 3.

The recursive node visits are similar to a search for points on $c(q)$ and V-Partition performs at most two recursive calls per node in TSTs. Reverting these steps in exactly the reverse order provides the inverse operation V-Merge(u^-, u^+, q, v) on two trees with adjacent search regions (c.f. Algorithm 4). Note that the TSD primitives actually perform *fewer* recursive calls, since vertical cuts are contracted in this structure.

3.3 Recursive Edge Partitions and Merges

Based on the recursive primitives for vertical cuts, we now introduce the recursive edge partition Partition(u, c, v^-, v^+) on regions $\Delta(u)$ that are fully crossed by the edge cut c (c.f. Algorithm 5). As above, v^- and v^+ denote the initially empty result trees for the respective partition of $\Delta(u)$ in $\Delta^- \cup \Delta^+$.

If the edge cut c does not intersect the edge cut that destroys u, the recursive results on nodes of Descend(u) only need V-Merge calls on one side of c to produce the result. E.g. the left case in the figure of Algorithm 5: First a V-Merge call on (v_a^+, v_r^+) and then with v_l^+. For new intersections with c, let i denote the intersection point. We first use V-Partition(u_a, i, v_{al}, v_{ar}) and V-Partition(u_b, i, v_{bl}, v_{br}) to create trees v_{al}, v_{bl} and v_{ar}, v_{br} for the respective left and right sides of i, prior to the edge partition with c. Finally, we use V-Merge to combine those results properly. E.g. the two gray areas in right case in the figure of Algorithm 5.

Note that for segments from a planar subdivision, the treatment of intersections is not necessary. The inverse primitive for edge cut merges (of two adjacent regions) can be derived analogously, by executing the inverse primitives in exactly the reverse order.

4 Counting Search Nodes in Affected Regions

To improve readability in this section, we denote for singleton elements the set union $S \cup \{s\}$ with $S + s$ and the set difference $S \setminus \{s\}$ with $S - s$. With $\mathbf{P}(S)$ we denote the set of bijective mappings $\pi : S \to \{1, \dots, |S|\}$. For $S' \subseteq S$ we use the predicate "$\pi(S') \leq |S'|$" to abbreviate that $\pi(s) \leq |S'|$ holds for each $s \in S'$, i.e. the permutation begins with the elements of S'.

The following definitions concern sets of segments and their (deterministic) induced trapezoidal subdivision $\mathcal{A}$ of the plane. For $s \in S \setminus B$, we define $F_B(s) \subseteq \mathbb{R}^2$ to denote the region of the faces in $\mathcal{A}(B)$ that are intersected by s, that is $F_B(s) = \bigcup_{\{f \in \mathcal{A}(B) : f \cap s \neq \emptyset\}} f$. Moreover, for $s \in B$ we define the neighborhood zone $N_B(s) \subseteq \mathbb{R}^2$ to be the union of all faces of $\mathcal{A}(B)$ that are adjacent to s (e.g. s contributes as edge, vertex or intersection cut). Note that every point in $\mathbb{R}^2$ is in at most 4 neighborhood zones and $N_{B+s}(s) = F_B(s)$.

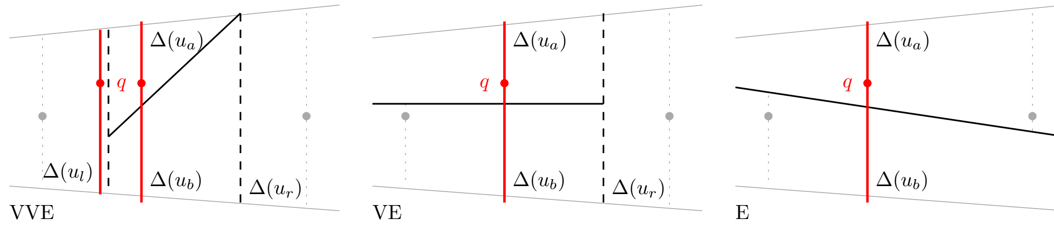


Figure 3 Cases during recursions of V-Partition(u, q, v^-, v^+) and V-Merge(u^-, u^+, q, v).

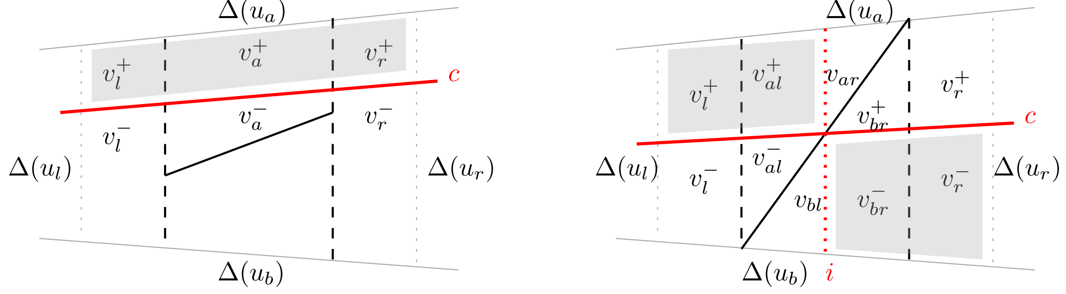
Algorithm 3 V-Partition(u, q, v^-, v^+): Assertion: $c(q)$ intersects $\Delta(u)$.

Stop if u is a leaf; Let $(u_l, u_a, u_b, u_r) :=$ Descend(u).
TST: **IF** $c(q)$ intersects $\Delta(u_a)$ and $\Delta(u_b)$
TSD: **IF** $q \in \Delta(u_a)$ "$q \in \Delta(u_b)$" analogue
If present, move u_l and its parent's vertical cut in v^-.
If present, move u_r and its parent's vertical cut in v^+.
Cut both unoccupied leaves, that is one child of v^- and one of v^+, with the edge cut. Let $v_a^-, v_b^-, v_a^+, v_b^+$ denote these leaves.
V-Partition(u_a, q, v_a^-, v_a^+)
TST: V-Partition(u_b, q, v_b^-, v_b^+)
TSD: Set both below pointers on u_b instead of v_b^- and v_b^+.
ELSE IF $c(q)$ intersects $\Delta(u_l)$ "$c(q)$ intersects $\Delta(u_r)$" symmetrically
Move u_r (if present) , u_a, u_b and their parents' cuts in v^+.
Let v_l^+ denote the unoccupied leaf.
V-Partition(u_l, q, v^-, v_l^+)

Algorithm 4 V-Merge(u^-, u^+, q, v): Assertion: $c(q)$ bounds $\Delta(u^-)$ and $\Delta(u^+)$.

IF u^- is leaf **THEN** Move contents of u^+ in v and stop. "u^+ leaf" analogue
Let $(u_l^-, u_a^-, u_b^-, u_r^-) :=$ Descend(u^-) and $(u_l^+, u_a^+, u_b^+, u_r^+) :=$ Descend(u^+).
TST: **IF** $p(u^-) = p(u^+)$
TSD: **IF** $p(u^-) = p(u^+)$ and $u_b^- = u_b^+$ "$u_a^- = u_a^+$" analogue
If present, move u_l^- and its parent's vertical cut in v.
If present, move u_r^+ and its parent's vertical cut in the unoccupied leaf of v.
Cut the new unoccupied leaf of v with the edge cut. Let v_a, v_b denote these leaves.
V-Merge(u_a^-, u_a^+, q, v_a)
TST: V-Merge(u_b^-, u_b^+, q, v_b)
TSD: Set the below pointer on u_b^- instead of v_b.
ELSE IF $p(u^-) < p(u^+)$ ">" symmetrically
Move u_l^- (if present) , u_a^-, u_b^- and their parents' cuts in v.
Let v_r denote the unoccupied leaf.
V-Merge(u_r^-, u^+, q, v_r)

Algorithm 5 Partition(u, c, v^-, v^+): Assertion: c crosses $\Delta(u)$ entirely.



Stop if u is a leaf.
Let $(u_l, u_a, u_b, u_r) :=$ Descend(u) and l_1, l_2 denote the vertical cuts destroying $\Delta(u)$.
IF c does not intersect $\Delta(u_b)$ properly "not $\Delta(u_a)$" analogue
  Allocate new nodes $v_l^-, v_l^+, v_a^-, v_a^+, v_r^-, v_r^+, v_{ar}^+$.
  Partition(u_l, c, v_l^-, v_l^+); Partition(u_a, c, v_a^-, v_a^+); Partition(u_r, c, v_r^-, v_r^+)
  V-Merge$(v_a^+, v_r^+, l_2, v_{ar}^+)$
  V-Merge$(v_l^+, v_{ar}^+, l_1, v^+)$
  Place v_l^-, v_a^-, u_b, v_r^- below respective cuts under v^-.
ELSE IF edge cut of u is steeper than c "less steep" symmetrically
  Let i denote the vertical cut (induced by the intersection of the edge-cut and c).
  Allocate new nodes $v_l^-, v_l^+, v_r^-, v_r^+$ as well as $v_{ar}, v_{al}, v_{al}^-, v_{al}^+$ and $v_{bl}, v_{br}, v_{br}^-, v_{br}^+$.
  Partition(u_l, c, v_l^+, v_l^-); Partition(u_r, c, v_r^+, v_r^-)
  V-Partition(u_a, i, v_{al}, v_{ar}); V-Partition(u_b, i, v_{bl}, v_{br})
  Partition$(v_{al}, c, v_{al}^-, v_{al}^+)$; Partition$(v_{br}, c, v_{br}^-, v_{br}^+)$

  Place V-Merge$(v_l^+, v_{al}^+, l_1, \cdot)$ as left child under cut i below v^+.
  Place v_r^+ and its parent's cut in the unoccupied leaf of v^+.
  Cut the unoccupied leaf of v^+ with the edge cut of u.
  Place v_{ar} and v_{br}^+ in these leaves.

  Place v_l^- and its parent's cut in v^-.
  Place V-Merge$(b_{br}^-, v_r^-, l_2, \cdot)$ as right child under cut i in v^-.
  Cut the unoccupied leaf of v^- with the edge cut of u.
  Place v_{al}^- and v_{bl} in these leaves.

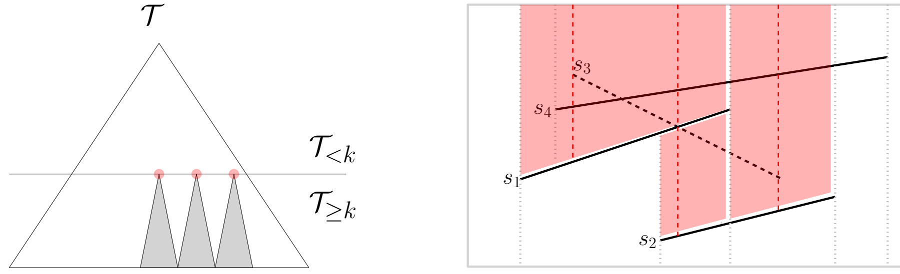
Given a TST $\mathcal{T}(S, \pi)$ and some priority value $r \in \{1, \ldots, |S|\}$, we consider partitions of its nodes in classes $\mathcal{T}_{<r}, \mathcal{T}_{=r}$, and $\mathcal{T}_{>r}$, having priority less, equal or larger than r. E.g. $\mathcal{T}_{<1} = \emptyset$ and $\mathcal{T}_{>r}$ always contains the leaf nodes. As introduced in Section 2.1, we identify every node v in $\mathcal{T}$ with its associated search region $\Delta(v) \subseteq \mathbb{R}^2$.

From the refinement property (c.f. Section 2.2), we have that the leaves' partition $\{\Delta(v) \subseteq \mathbb{R}^2 : v \in \mathcal{T}(S, \pi),\ p(v) = \infty\}$ of the domain is a (set theoretic) refinement of $\mathcal{A}(S)$, for any S and $\pi \in \mathbf{P}(S)$. This applies in particular for a set of segments $S_{\leq r} = \{s \in S : \pi(s) \leq r\}$. Since additional leaf-insertions only refine further, we have that the region $\Delta(v)$ of a node $v \in \mathcal{T}_{>r}$ is either contained in or disjoint from neighborhood zone $N_{S_{\leq r}}(s)$ for any $s \in S_{\leq r}$.

This provides the following 4-ply covering property. For any $B \subseteq S$ with a $\pi \in \mathbf{P}(S)$ such that $\pi(B) \leq |B|$, we have that

$$\sum_{s \in B} \Big| \{ v \in \mathcal{T}_{=l}(S, \pi) \ : \ \Delta(v) \subseteq N_B(s) \} \Big| \ \leq \ 4 \Big| \mathcal{T}_{=l}(S, \pi) \Big| \ , \tag{1}$$

for each $l > |B|$.



(a) Partition of nodes in $\mathcal{T}$ in $\mathcal{T}_{<k}$ and $\mathcal{T}_{\geq k}$ due to node priority. Affected nodes are in red and their subtrees in gray.

(b) The neighborhood zone of s_3 (shaded in red) is $F_{\{s_1,s_2\}}(s_3) = N_{\{s_1,s_2,s_3\}}(s_3)$.

Figure 4 Illustration of affected subtrees (left) and neighborhood zone refinement (right).

▶ **Definition 4.1** (Affected Nodes). *Given a segment $s \in S$ of designated priority rank $r \in \{1, \ldots, |S|\}$, we call a node v of $\mathcal{T}_{\geq r}$ over $S - s$ affected if and only if $\Delta(v) \cap s \neq \emptyset$ and it is topmost in $\mathcal{T}$. That is, v has no parent u in $\mathcal{T}_{\geq r}$ with $p(u) \leq p(v)$.*

Clearly, for every $v \in \mathcal{T}_{\geq r}$ whose region $\Delta(v)$ intersects s, we have $\Delta(v) \subseteq F_{S_{<r}}(s)$ as well. In other words, the affected nodes correspond precisely to the leaves of $\mathcal{T}(S_{<r}, \pi)$, the tree over the first $r-1$ segments, that are intersected by s. See Figure 4a for an illustration and Figure 4b for an example. In this example, decomposition $\mathcal{A}(\{s_1, s_2\})$ has 7 faces of which s_3 intersects 3. The neighborhood region of s_3 is $F_{\{s_1,s_2\}}(s_3) = N_{\{s_1,s_2,s_3\}}(s_3)$ and shaded in red. The TST for $\left(\{s_1, s_2, s_4\}, \left(\begin{smallmatrix} s_1 & s_2 & s_4 \\ 1 & 2 & 3 \end{smallmatrix}\right)\right)$ has 13 leaf regions and the TSD has 10 leaves. Note that for TST nodes v with priority $p(v) \geq 3$, the region $\Delta(v)$ is either fully contained or outside the red zone.

Given a $\pi \in \mathbf{P}(S-s)$ and a value $p \in \{1, \ldots, |S|\}$, we call the region $F_{\{s' \in (S-s)\ :\ \pi(s') < p\}}(s)$ the *p-neighborhood of s*.

▶ **Lemma 4.2** (Zone Covering). *Let S be a set of non-crossing segments, $p \in \{1, \ldots, |S|\}$ a fixed value, and $s \in S$. For any $l \geq p$, the expected number of priority l nodes of a random TST over $S - s$ that are in the p-neighborhood of s is upper bounded with*

$$\mathop{\mathbb{E}}_{\pi \in \mathbf{P}(S-s)} \left| \left\{ v \in \mathcal{T}_{=l}(S-s, \pi) \ : \Delta(v) \subseteq F \right\} \right| \ \leq \ \frac{4}{p} \mathop{\mathbb{E}}_{\pi \in \mathbf{P}(S-s)} \left| \mathcal{T}_{=l}(S-s, \pi) \right| ,$$

where F denotes the p-neighborhood of s.

On a sequence of elements, we call the process to place a new element between the elements of positions $p-1$ and p *emplacing* at p and *dropping* is the reverse operation. E.g. old elements of index less than p remain and those of at least p are moved one position to the right. We perform a backward analysis and consider dropping the element at position p in sequences of $\mathbf{P}(S)$ instead of emplacing a $s \in S$ at position p in sequences of $\mathbf{P}(S-s)$.

Proof. The main idea is to group the sequences in $\mathbf{P}(S)$ by those having the same cardinality p subsets $B \in \binom{S}{p}$ in the first p entries to use that $\bigcup_{s \in B} N_B(s) = \mathbb{R}^2$, as in Eq. 1.

For $\pi \in \mathbf{P}(S)$, we denote with $\pi^{-1}(p) \in S$ the element that π maps to position p and with $\pi - s$ we denote the sequence of $\mathbf{P}(S-s)$ that results from dropping element s. To shorten notation, we define a random variable γ that counts the relevant nodes of a tree within a certain region $N \subseteq \mathbb{R}^2$, that is $\gamma(S, \pi, N) = |\{v \in \mathcal{T}_{=l}(S, \pi)\ :\ \Delta(v) \subseteq N\}|$.

We count the expected number of nodes within $\mathcal{T}(S-\pi^{-1}(p), \pi-\pi^{-1}(p))$, that fall into the region $N_{\{s\in S\ :\ \pi(s)\leq p\}}\left(\pi^{-1}(p)\right)$, which is the neighborhood zone of the element at position p among those with priority at most p.

$$\underset{\pi\in\mathbf{P}(S)}{\mathbb{E}}\ \gamma\left(S-\pi^{-1}(p), \pi-\pi^{-1}(p), N_{\{s\in S\ :\ \pi(s)\leq p\}}\left(\pi^{-1}(p)\right)\right) \tag{2}$$

$$=\frac{1}{|S|!}\sum_{B\in\binom{S}{p}}\sum_{s\in B}\sum_{\substack{\pi\in\mathbf{P}(S)\ :\\ \pi(B)\leq p,\ \pi(s)=p}}\gamma(S-s, \pi-s, N_B(s)) \tag{3}$$

$$\leq\frac{1}{|S|!}\sum_{B\in\binom{S}{p}}\sum_{s\in B}\frac{4}{p}\sum_{\substack{\pi\in\mathbf{P}(S)\ :\\ \pi(B)\leq p,\ \pi(s)=p}}\gamma(S-s, \pi-s, \mathbb{R}^2) \tag{4}$$

$$=\frac{4}{|S|!\ p}\sum_{s\in S}\sum_{B\in\binom{S-s}{p-1}}\sum_{\substack{\pi\in\mathbf{P}(S-s)\ :\\ \pi(B)<p}}\gamma(S-s, \pi, \mathbb{R}^2) \tag{5}$$

$$=\frac{1}{|S|}\sum_{s\in S}\frac{4}{p}\frac{1}{|S-s|!}\sum_{\pi\in\mathbf{P}(S-s)}\gamma(S-s, \pi, \mathbb{R}^2) \tag{6}$$

$$=\frac{1}{|S|}\sum_{s\in S}\ \frac{4}{p}\underset{\pi\in\mathbf{P}(S-s)}{\mathbb{E}}\gamma(S-s, \pi, \mathbb{R}^2) \tag{7}$$

Equation (3) is due to regrouping the summation terms by the sequences that have the same sets of elements B in the first p positions, (4) due to the ordinary[1] "camel trick" of the form $\sum_{s\in B} f(s) = \sum_{s\in B}\frac{1}{|B|}\sum_{s\in B} f(s)$ and the 4-ply covering. Equation (5) uses the combinatorial identity $\binom{n}{k}\binom{k}{1} = \binom{n}{1}\binom{n-1}{k-1}$ to first choose $s\in S$ for position p. Since the terms in (7) do not depend on the spatial location of s, the bound holds for any $s\in S$. ◀

We now bound the expected total size of subtrees below affected nodes (c.f. Figure 4a).

▶ **Lemma 4.3** (Subtree Sizes). *Let S be a set of non-crossing segments, $p\in\{1,\ldots,|S|\}$ uniformly at random, and $s\in S$. The expected total size of affected subtrees in a random TST over $(S-s)$ is $\mathcal{O}\left(\log^2|S|\right)$.*

Proof. We use Lemma 4.2 to bound the number of nodes in $\mathcal{T}(S-s,\pi)$ that have a priority of at least p and have a region that is in the p-neighborhood of s.

$$\frac{1}{|S|}\sum_{p=1}^{|S|}\underset{\pi\in\mathbf{P}(S-s)}{\mathbb{E}}\left|\left\{v\in\mathcal{T}_{\geq p}(S-s,\pi)\ :\ \Delta(v)\subseteq F_{\{s'\in(S-s)\ :\ \pi(s')<p\}}(s)\right\}\right| \tag{8}$$

$$\leq\frac{1}{|S|}\sum_{p=1}^{|S|}\frac{4}{p}\underset{\pi\in\mathbf{P}(S-s)}{\mathbb{E}}\left|\mathcal{T}_{\geq p}(S-s,\pi)\right| \tag{9}$$

$$\leq\frac{1}{|S|}\sum_{p=1}^{|S|}\frac{4}{p}\underset{\pi\in\mathbf{P}(S-s)}{\mathbb{E}}\left|\mathcal{T}(S-s,\pi)\right| \tag{10}$$

$$\leq\frac{1}{|S|}\sum_{p=1}^{|S|}\frac{4}{p}\mathcal{O}\left(|S-s|\log|S-s|\right)\leq\mathcal{O}\left(\log^2|S|\right) \tag{11}$$

Bound (11) is due to the expected size of the whole tree (c.f. Theorem 2.1). ◀

[1] E.g. proof of Theorem 3.1.4 in [16, p. 90] or proof of Theorem 6.3 in [11, p. 136].

With a slightly more advanced application of our arguments presented thus far, we now bound the cost for simple iterated ray-shooting search to find the affected subtree roots for a query segment that has a designated random priority rank (c.f. Section 3.1).

▶ **Lemma 4.4** (Segment Search). *Let S be a set of non-crossing segments, $p \in \{1, \ldots, |S|\}$ uniformly at random, and $s \in S$. The expected time to find the affected nodes within a random TST over $(S - s)$ is $\mathcal{O}\left(\log^2 |S|\right)$.*

Proof. The search visits only nodes v with regions $\Delta(v) \cap s \neq \emptyset$. Since Lemma 4.3 bounds the size of the result set, we only need to bound the number of nodes with priority smaller p. We do so by invoking Lemma 4.2 $\lceil \log_2 p \rceil$ times.

$$\frac{1}{|S|} \sum_{p=1}^{|S|} \mathop{\mathbb{E}}_{\pi \in \mathbf{P}(S-s)} \left| \left\{ v \in \mathcal{T}_{<p}(S - s, \pi) \;:\; \Delta(v) \cap s \neq \emptyset \right\} \right| \tag{12}$$

$$\leq \frac{1}{|S|} \sum_{p=1}^{|S|} \sum_{j=1}^{\lceil \log_2 p \rceil} \mathop{\mathbb{E}}_{\pi \in \mathbf{P}(S-s)} \left| \left\{ v \in \mathcal{T}(S - s, \pi) : 2^{j-1} < p(v) \leq 2^j ,\; \Delta(v) \cap s \neq \emptyset \right\} \right| \tag{13}$$

$$= \frac{1}{|S|} \sum_{p=1}^{|S|} \sum_{j=1}^{\lceil \log_2 p \rceil} \frac{4}{2^{j-1}} \mathop{\mathbb{E}}_{\pi \in \mathbf{P}(S-s)} \left| \left\{ v \in \mathcal{T}(S - s, \pi) \;:\; 2^{j-1} < p(v) \leq 2^j \right\} \right| \tag{14}$$

$$\leq \frac{1}{|S|} \sum_{p=1}^{|S|} \sum_{j=1}^{\lceil \log_2 p \rceil} \frac{4}{2^{j-1}} \mathcal{O}\left(2^j \log(2^j)\right) \tag{15}$$

$$= \frac{1}{|S|} \sum_{p=1}^{|S|} \sum_{j=1}^{\lceil \log_2 p \rceil} \mathcal{O}\left(j\right) \leq \frac{1}{|S|} \sum_{p=1}^{|S|} \mathcal{O}(\log^2 p) \leq \mathcal{O}\left(\log^2 |S|\right) \tag{16}$$

In (13), we use “linearity of expectation” to count search nodes of $\mathcal{T}_{<p}$ in batches of priority intervals of the form $(2^{j-1}, 2^j]$. Bound (14) is due to invoking Lemma 4.2 for s according to the priority interval. To bound a batch j in (15) we use Theorem 2.1 on the whole TST size over the first 2^j segments. ◀

Now we have all necessary arguments for our bound on dynamic update costs in TSTs.

▶ **Theorem 4.5.** *Let S be a set of non-crossing segments and $s \in S$. The expected cost of inserting s in a random TST over $(S - s)$ is $\mathcal{O}(\log^2 |S|)$.*

Proof. Let $1 \leq m \leq |\mathcal{T}|$ denote the total number of nodes within affected subtrees of $\mathcal{T}$, prior to the insertion call. We argue that the total number of nodes visited by the insertion procedure is $\mathcal{O}(m)$, which establishes the result based on Lemma 4.3 and 4.4 for non-crossing segments.

For each root of an affected subtree, we have at most 2 calls of V-Partition to slab the subtree. After the slabbing stage of the insertion, we have at most $3m$ nodes in total.

The edge Partition calls on one slab are independent from those of another slab in $\mathcal{T}$. Neglecting node removals due to V-Merge briefly, the total number of temporarily created nodes of Partition increases at most by a factor of 2. Since we cannot remove more than what was created, the total number of nodes that V-Merge may visit is bounded by $6m$. ◀

Given the duality of the update procedures, deletion of a $s \in S$ having priority $p(s)$ visits exactly the same nodes as insertion of s among $(S - s)$ with priority value $p(s)$. Hence the expected node visits are equal as well. Since our bound on expected update operations

counts TST search nodes, we may employ the method of Hemmer et al. [13], on the bijection between TST and TSD search paths, to obtain equal asymptotic bounds for the expected update costs in TSDs. A more detailed consideration, however, allows an improvement.

4.1 Improved Update Bounds For TSDs

The region of each TSD node may be represented by its top, bottom and vertical boundary cuts. To use "linearity of expectation", we decompose the random variable $|\mathcal{D}(S,\pi)|$ in a sum of $\mathcal{O}(|S|^5)$ binary indicator variables $\mathcal{N}_{i,i',i'',i''',j}$ that are 1 if and only if $\mathcal{D}(S,\pi)$ contains a node whose boundary is constituted by cuts with the priorities i, i', i'', i''' and gets destroyed by the segment with priority j. That is $|\mathcal{D}_{=l}| = \sum_{i,i',i'',i'''} \mathcal{N}_{i,i',i'',i''',l}$. The number of these indicator variables only depends on $|S|$, but not on π or the spatial location of the segments.

▶ **Lemma 4.6** (Overlap into Zones). *Let S be a set of non-crossing segments, $p \in \{1,\dots,|S|\}$ a fixed value, and $s \in S$. For any $l \geq p$, the expected number of priority l nodes of a random TSD over $S - s$ that overlap the p-neighborhood of s is upper bounded with*

$$\mathop{\mathbb{E}}_{\pi\in\mathbf{P}(S-s)} \left|\left\{v \in \mathcal{D}_{=l}(S-s,\pi) \;\; : \Delta(v) \cap F \neq \emptyset\right\}\right| \;\leq\; \frac{\mathcal{O}(1)}{p} \mathop{\mathbb{E}}_{\pi\in\mathbf{P}(S-s)} \left|\mathcal{D}_{=l}(S-s,\pi)\right| \,,$$

where F denotes the p-neighborhood of s.

For the proof recall that, for any subset $C \subseteq S$ and its maximum priority element $m \in C$, classic Backward Analysis (e.g. Chapter 6 in [11]) shows that the number d_C, denoting the number of faces in $\mathcal{A}(C-m)$ that segment m intersects (a.k.a destroys), is expected $\mathcal{O}(1)$. Since the neighborhood zones are a union of faces of $\mathcal{A}(C-m)$, element m also intersects no more than $4 \cdot d_C$ of the neighborhood zones in N_{C-m}.

Proof. The proof is analogue to Lemma 4.2, except from Eq. 4 which uses that every TST node is in at most 4 neighborhood zones. Given the refined decomposition of $|\mathcal{D}_{=l}|$, showing that *each* of the priority l nodes intersects at most $\mathcal{O}(1)$ zones of N_B gives the statement due to linearity of expectation.

A priority $l < \infty$ node, i.e. a non-leaf node, is either fully crossed by the segment $\pi^{-1}(l)$ or contains an endpoint of it. In the first case, the number of N_B zones that the node region intersects is bounded by $4 \cdot d_C$, where $C = B + \pi^{-1}(l)$. For the endpoint case, let $|B| < l', l'' < l$ denote the priorities of the top and bottom segment that bound the node's trapezoidal region. Hence, the node does not intersect more than $4(d_{C'} + d_{C''})$ neighborhood zones where $C' = B + \pi^{-1}(l')$ and $C'' = B + \pi^{-1}(l'')$. The argument for a leaf node, i.e. $l = \infty$, is analogue to the endpoint case. ◀

▶ **Corollary 4.7.** *Let S be a set of non-crossing segments and $s \in S$. The expected cost of inserting s in a random TSD over $(S-s)$ is $\mathcal{O}(\log|S|)$.*

Proof. Using Lemma 4.6, Equation 11 improves to $\frac{1}{|S|}\sum_{p=1}^{|S|} \mathcal{O}(\frac{1}{p}|S-s|)$, which evaluates to an expected number of $\mathcal{O}(\log|S|)$ TSD nodes to overlap into the affected zone. Moreover, Equation 15 improves to $\frac{1}{|S|}\sum_{p=1}^{|S|}\sum_{j=1}^{\lceil\log_2 p\rceil} \mathcal{O}\left(\frac{1}{2^{j-1}}2^j\right)$, which evaluates to an expected number of $\mathcal{O}(\log|S|)$ TSD nodes that are visited in the segment search. Since our recursive update primitives visit a TSD node at most once, using the same arguments as in the proof of Theorem 4.5 on TSDs provides the expected update cost as stated. ◀

5 Offline vs Online – Maintaining Small Codes For Dynamic Orders

To support online emplacement and dropping of additional elements in a sequence $\pi \in \mathbf{P}(S)$, we use a more flexible representation than standard bijections $\pi : S \to \{1, \ldots, |S|\}$. We represent sequences as injective mappings $\tau : S \to (0,1) \subseteq \mathbb{R}$. If the values of τ are stored with the elements of S, we can evaluate the required $\pi(s) < \pi(s')$ predicates with comparison on τ in constant time. However, dynamically extending τ for a new element $s \notin S$ such that $\tau(s)$ falls with equal probability in the intervals between $\{\tau(s') : s' \in S\}$ is challenging. E.g. simple random sampling of $\tau(s)$ uniformly out of $(0,1)$ does not provide this. Since single registers of the pointer machine model are confined to numbers with only $\mathcal{O}(\log |S|)$ bits, simple interval halving strategies may well produce inefficiently long codes for τ as well.

We solve this problem using one additional randomized data structure to maintain orders among n elements of a totally ordered set of "keys'. Treaps are a conceptual fusion of Binary Search Trees (BSTs), over the nodes' key values, and (Min-)Heaps, over the nodes' priority values. Insertions and deletions are performed similarly to BSTs and additionally standard tree rotations are used to maintain the heap property on the randomly chosen priority values. Seidel and Aragon [21] show not only that Treaps achieve $\mathcal{O}(\log n)$ depth with high probability but also that, even if the cost of a rotation is proportional to the whole subtree size, insertions and deletions still perform expected $\mathcal{O}(\log n)$ operations.

For our simple in-order numbering scheme in a fixed BST, we consider an injective mapping from root-to-node paths, which we read as smaller/larger strings over the alphabet $\{s, l\}$, to binary fractional number from $(0,1) \subseteq \mathbb{Q}$ coded as "1"-terminated strings over the alphabet $\{0,1\}$. More precisely, for the empty path string (the root node r) the associated value is $\gamma(r) = 0.1_{(2)}$, s-edges are coded as 0 and l-edges are coded as 1. E.g. the left child node v_s of the root has code $\gamma(v_s) = 0.01_{(2)}$, the right child node v_l of the root has code $\gamma(v_l) = 0.11_{(2)}$, and we have $\gamma(v_s) < \gamma(r) < \gamma(v_l)$. On paths of a BST, the search tree property extends to the order of these γ values of the nodes. This allows us to check if u is before v in the in-order sequence by comparing the values of $\gamma(u)$ and $\gamma(v)$. Moreover, this recursive code definition can be assigned in a top-down fashion for all nodes in a subtree under v, once $\gamma(v)$ is assigned.

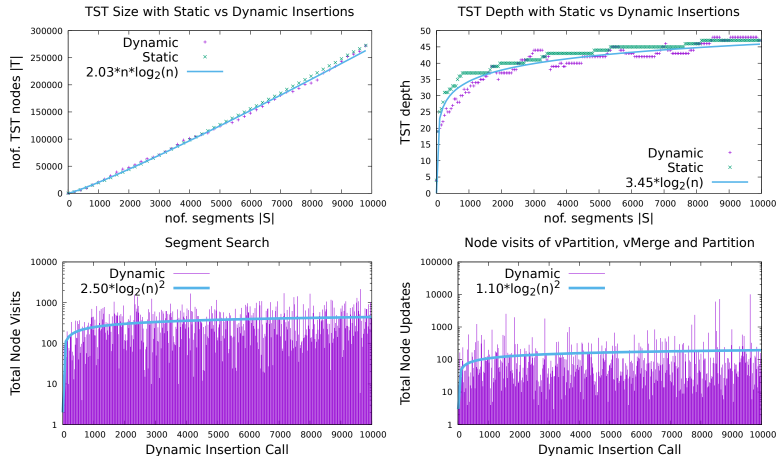
In order to maintain the proper values τ for a set S, we augment the Treap nodes to also store the total number of nodes in their subtree. To emplace a new element, we first choose an integer $1 \leq p \leq |S| + 1$ uniformly at random. Next we use the standard Treap insertion to add a new node resembling the p-smallest key-value as a new leaf. After the bottom-up re-balancing rotations are finished, we simply re-visit the whole rotated subtree in a top-down fashion to overwrite the key value of each node v with the new value $\gamma(v)$. The deletion of an element is handled analogously. To evaluate an order predicate between two elements, we simply compare the current key values due to the Treap, which are stored with the elements of S. We summarize this with the following statement.

▶ **Observation 5.1.** *Representation of sequences $\pi \in \mathbf{P}(S)$ requires no more than $\mathcal{O}(|S|)$ space and dynamic updates take expected $\mathcal{O}(\log |S|)$ operations. After successful updates, evaluation of a $\pi(s) < \pi(s')$ predicate on $s, s' \in S$ takes $\mathcal{O}(1)$ operations.*

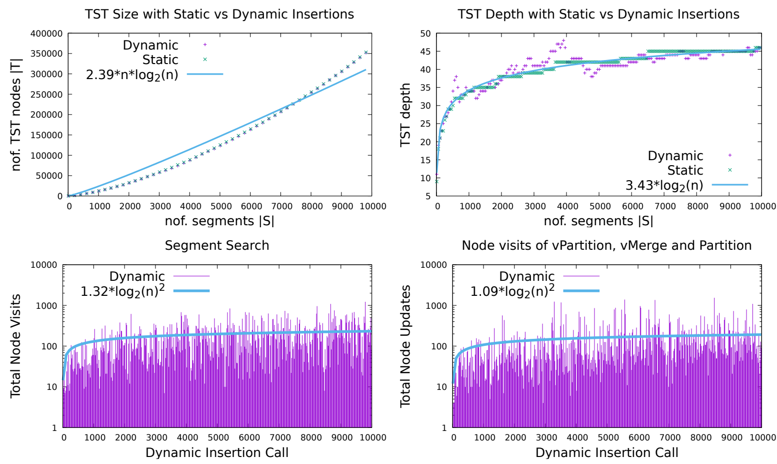
6 Implementation and Experiments

We implemented[2] our fully-dynamic approach for TSTs based on the exact predicates and exact construction for line segments of the Computational Geometry Algorithms Library [23].

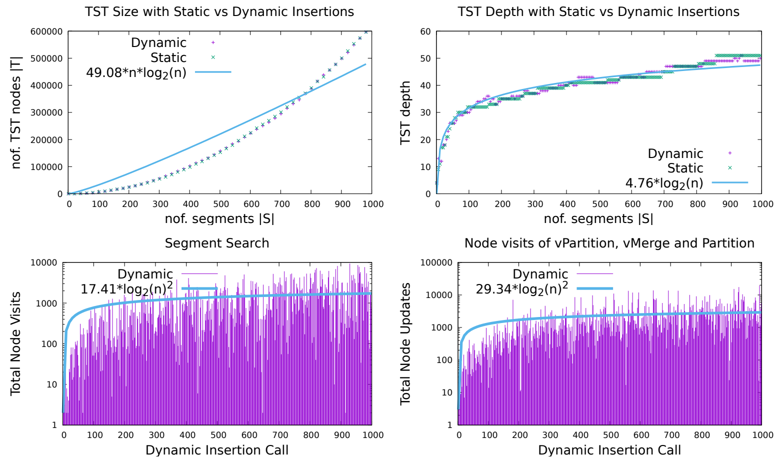
[2] https://github.com/milutinB/dynamic_trapezoidal_map_impl



(a) Random non-intersecting segments ($|S| = 10^4,\ k_S = 0,\ k_S/|S| = 0$).



(b) Random segments with few intersections ($|S| = 10^4,\ k_S = 35195,\ k_S/|S| \approx 3.5$).



(c) Random segments with many intersections ($|S| = 10^3,\ k_S = 120730,\ k_S/|S| \approx 120.7$).

Figure 5 Experimental results on TST size (top left), TST depth (top right), and node visits of the segment search (bottom left) and updates (bottom right) for RIC TSTs (green) and Dynamic TSTs (purple). The blue curve indicates the respective function fit on the data of Dynamic TSTs.

Our experiments focus on evaluating the practicality of the approach and the tightness of the analysis. We use the standard RIC of TSTs as verification and baseline comparison. To measure the performance of our segment search and dynamic updates, we count the total number of node visits during recursions of the update and the search. To capture the asymptotic behavior in the experiments, we perform many insertion calls, with either ascending segment priority values (static TST) or a random shuffle of them (dynamic TST).

We created random sets of segments with varying numbers of intersections based on the 64-bit Mersenne Twister random number generator of the C++ Standard Library. The experiment without intersections is comprised of horizontal segments, for which we first chose a y-coordinate uniformly (from the domain range) and then two x-coordinates uniformly (see Figure 5a). The experiment with few intersections is comprised of short segments, for which we chose a point, a direction, and a length uniformly at random (on average 3% of the domain boundary length; see Figure 5b). The experiment on many intersections are generated by choosing the coordinates of both endpoints uniformly at random (see Figure 5c).

Figure 5 shows the results on the segment search and recursive node visits of updates. For ease of comparison, we also plot a function fit[3] (thick blue lines) of the well known size and depth bounds (top rows) and our new segment search and update bounds (bottom rows) as overlay on the experimental data. Note that only on the non-intersecting data set a comparison of function fit and bounds is meaningful. We do however also provide it for the other experiments as a "trend line" that indicates the additional costs due to segment intersections. To allow a better visual perception of the average cost per insertion, Figures 5a and 5b show every 20th data point and Figure 5c shows every 2nd data point as impulse.

The experiment on non-intersecting segments matches with our analysis of segment search and update operations on TSTs. As anticipated, TST updates on segments with many intersections are more expensive.

References

1. Pankaj K. Agarwal, Julien Basch, Mark de Berg, Leonidas J. Guibas, and John Hershberger. Lower bounds for kinetic planar subdivisions. *Discrete & Computational Geometry*, 24(4):721–733, 2000. doi:10.1007/s004540010060.
2. Pankaj K. Agarwal, Jeff Erickson, and Leonidas J. Guibas. Kinetic binary space partitions for intersecting segments and disjoint triangles. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 107–116, 1998. URL: http://dl.acm.org/citation.cfm?id=314613.
3. Lars Arge, Gerth Stølting Brodal, and Loukas Georgiadis. Improved dynamic planar point location. In *Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 305–314, 2006. doi:10.1109/FOCS.2006.40.
4. Daniel Bahrdt and Martin P. Seybold. Rational points on the unit sphere: Approximation complexity and practical constructions. In *Proceedings of the 2017 ACM on International Symposium on Symbolic and Algebraic Computation (ISSAC)*, pages 29–36, 2017. doi:10.1145/3087604.3087639.
5. Hanna Baumgarten, Hermann Jung, and Kurt Mehlhorn. Dynamic point location in general subdivisions. *Journal of Algorithms*, 17(3):342–380, 1994. doi:10.1006/jagm.1994.1040.
6. Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, Tsvi Kopelowitz, and Pablo Montes. File maintenance: When in doubt, change the layout! In *Proceedings of the 28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1503–1522, 2017. doi:10.1137/1.9781611974782.98.

[3] We use the standard least-squares Marquardt-Levenberg algorithm of GNUPlot 5.2.

7. Timothy M. Chan and Yakov Nekrich. Towards an optimal method for dynamic planar point location. In *Proceedings of the 56th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 390–409, 2015. doi:10.1109/FOCS.2015.31.
8. Siu-Wing Cheng and Ravi Janardan. New results on dynamic planar point location. *SIAM Journal on Computing*, 21(5):972–999, 1992. doi:10.1137/0221057.
9. Y. Chiang and R. Tamassia. Dynamic algorithms in computational geometry. *Proceedings of the IEEE*, 80(9):1412–1434, 1992. doi:10.1109/5.163409.
10. Yi-Jen Chiang and Roberto Tamassia. Dynamization of the trapezoid method for planar point location in monotone subdivisions. *International Journal of Computational Geometry & Applications*, 2(3):311–333, 1992. doi:10.1142/S0218195992000184.
11. Mark de Berg, Otfried Cheong, Marc J. van Kreveld, and Mark H. Overmars. *Computational Geometry: Algorithms and Applications, 3rd Edition.* Springer, 2008. doi:10.1007/978-3-540-77974-2.
12. Michael T. Goodrich and Roberto Tamassia. Dynamic trees and dynamic point location. *SIAM Journal on Computing*, 28(2):612–636, 1998. doi:10.1137/S0097539793254376.
13. Michael Hemmer, Michal Kleinbort, and Dan Halperin. Optimal randomized incremental construction for guaranteed logarithmic planar point location. *Computational Geometry: Theory and Applications*, 58:110–123, 2016. doi:10.1016/j.comgeo.2016.07.006.
14. Alex Hobé, Daniel Vogler, Martin P. Seybold, Anozie Ebigbo, Randolph R. Settgast, and Martin O. Saar. Estimating fluid flow rates through fracture networks using combinatorial optimization. *Advances in Water Resources*, 122:85–97, 2018. doi:10.1016/j.advwatres.2018.10.002.
15. Ketan Mulmuley. A fast planar partition algorithm, I. *Journal of Symbolic Computation*, 10(3-4):253–280, 1990. doi:10.1016/S0747-7171(08)80064-8.
16. Ketan Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms.* Prentice Hall, 1994.
17. J. Ian Munro and Yakov Nekrich. Dynamic planar point location in external memory. In *Proceedings of the 35th International Symposium on Computational Geometry (SoCG)*, pages 52:1–52:15, 2019. doi:10.4230/LIPIcs.SoCG.2019.52.
18. Eunjin Oh and Hee-Kap Ahn. Point location in dynamic planar subdivisions. In *Proceedings of the 34th International Symposium on Computational Geometry (SoCG)*, pages 63:1–63:14, 2018. doi:10.4230/LIPIcs.SoCG.2018.63.
19. Otfried Schwarzkopf. Dynamic maintenance of geometric structures made easy. In *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 197–206, 1991. doi:10.1109/SFCS.1991.185369.
20. Raimund Seidel. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Computational Geometry: Theory and Applications*, 1:51–64, 1991. doi:10.1016/0925-7721(91)90012-4.
21. Raimund Seidel and Cecilia R. Aragon. Randomized search trees. *Algorithmica*, 16(4-5):464–497, 1996. doi:10.1007/BF01940876.
22. Martin P. Seybold. Robust map matching for heterogeneous data via dominance decompositions. In *Proceedings of the 2017 SIAM International Conference on Data Mining (SDM)*, pages 813–821, 2017. doi:10.1137/1.9781611974973.91.
23. The CGAL Project. *CGAL User and Reference Manual.* CGAL Editorial Board, 4.14.1 edition, 2019. URL: https://doc.cgal.org/4.14.1/Manual/packages.html.
24. Athanasios K. Tsakalidis. Maintaining order in a generalized linked list. *Acta Informatica*, 21:101–112, 1984. doi:10.1007/BF00289142.