Untangled Monotonic Chains and Adaptive Range Search[☆]

Diego Arroyuelo[a,1], Francisco Claude[b], Reza Dorrigiv[b], Stephane Durocher[c], Meng He[b], Alejandro López-Ortiz[b], J. Ian Munro[b], Patrick K. Nicholson[b], Alejandro Salinger[b], Matthew Skala[c,*]

[a] *Yahoo! Research Latin America, Chile*
[b] *Cheriton School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada*
[c] *Department of Computer Science, University of Manitoba, Winnipeg, Manitoba, Canada*

Abstract

We present the first adaptive data structure for two-dimensional orthogonal range search. Our data structure is adaptive in the sense that it gives improved search performance for data that is better than the worst case [8]; in this case, data with more inherent sortedness.

Given n points on the plane, the linear-space data structure can answer range queries in $O(\log n + k + m)$ time, where m is the number of points in the output and k is the minimum number of monotonic chains into which the point set can be decomposed, which is $O(\sqrt{n})$ in the worst case. Our result matches the worst-case performance of other optimal-time linear-space data structures, or surpasses them when $k = o(\sqrt{n})$. Our data structure can be made implicit, requiring no extra space beyond that of the data points themselves [16], in which case the query time becomes $O(k \log n + m)$. We also present a novel algorithm of independent interest to decompose a point set into a minimum number of untangled, similarly directed monotonic chains in $O(k^2 n)$ time.

[☆] This research was supported through the NSERC Discovery Grants Program, the Canada Research Chairs Program, and an NSERC Strategic Grant on Optimal Data Structures for Organization and Retrieval of Spatial Data. A preliminary version of these results appeared at the International Symposium on Algorithms and Computation [2].

[*] Corresponding author.

Email addresses: darroyue@dcc.uchile.cl (Diego Arroyuelo), fclaude@cs.uwaterloo.ca (Francisco Claude), rdorrigiv@cs.uwaterloo.ca (Reza Dorrigiv), durocher@cs.umanitoba.ca (Stephane Durocher), mhe@cs.uwaterloo.ca (Meng He), alopez-o@cs.uwaterloo.ca (Alejandro López-Ortiz), imunro@cs.uwaterloo.ca (J. Ian Munro), p3nichol@cs.uwaterloo.ca (Patrick K. Nicholson), ajsalinger@cs.uwaterloo.ca (Alejandro Salinger), mskala@ansuz.sooke.bc.ca (Matthew Skala)

[1] Much of this work took place while this author was a visitor at the University of Waterloo.

1. Introduction

Applications in geographic information systems, among others, require structures that can store and retrieve spatial data efficiently in both space and time. In this work we describe a data structure and corresponding algorithm for two-dimensional orthogonal range search, a commonly encountered spatial data retrieval problem. Our data structure is *adaptive*, giving improved query performance for data with more inherent sortedness; and can be *implicit*, requiring no added storage space beyond that of the data points themselves. Along the way we present an algorithm of independent interest for decomposing a set of points into untangled monotonic chains.

The problem of *two-dimensional orthogonal range search* can be defined as follows: let $P = \{p_1, p_2, \ldots, p_n\}$ be a set of n points in the plane, and let $r = [x_1, x_2] \times [y_1, y_2]$ be a *query range*. The orthogonal range search problem asks for all points in $P \cap r$, that is, all $p_i \in P$ such that $x_1 \leq x(p_i) \leq x_2$ and $y_1 \leq y(p_i) \leq y_2$, where $x(p_i)$ and $y(p_i)$ denote the x and y coordinate values of point p_i respectively. An orthogonal range search data structure preprocesses the set P in order to efficiently answer arbitrary range queries; a natural goal is to balance the conflicting objectives of minimizing both the space required by the data structure and the time required to answer queries.

Our data structure is inspired by the range-trees of Lueker [14], which achieve fast queries at the cost of superlinear storage space for two dimensions by indexing along one dimension with a balanced binary tree, and then making each node of that tree the root of another tree that indexes the second dimension. The important insight is that if our data were monotonic, with the same ordering along both dimensions, then we could support fast query time like that of range trees while only requiring a single linear-space tree. Our data might not be monotonic in general, but we can always partition it into monotonic chains. The resulting data structure consumes $O(n)$ space and can answer queries in worst-case time $O(k \log n + m)$, where n is the number of points in the data set, m is the number of points returned, and k is the number of chains in a minimal decomposition, which is $O(\sqrt{n})$ in the worst case. The data structure can be made implicit, requiring no storage space beyond that necessary to store the point coordinates while keeping the query time of $O(k \log n + m)$; or, in the alternative, we can apply the fractional cascading technique of Chazelle and Guibas [6] to reduce the query time to $O(\log n + k + m)$ with $O(n)$ space.

For optimal query performance it is preferable that the monotonic chains should be untangled. That is, when successive vertices are connected by line segments, the chains should not intersect each other. This requirement does not increase the minimal number of chains. We present a novel algorithm for finding a minimal set of untangled chains (all monotonic in the same direction) in $O(k^2 n)$ time; this solution for the untangling problem is of independent interest.

2. Previous Work

Because of its importance, the two-dimensional orthogonal range search problem has received significant attention in the literature, and many data structures are known, providing different trade-offs of time and space. Decomposition of points into monotonic chains has also been well studied. We review here the main related existing work.

2.1. Minimal monotonic chain decompositions

Any set of n points can be partitioned into some number k of chains such that for each chain the y coordinate is monotonically increasing or decreasing as the x coordinate increases. When all chains must be ascending (or all descending), the problem of finding a minimal chain decomposition is well studied. With worst-case data the minimal number of chains all in the same direction may be $\Theta(n)$, even given a choice of the direction. Supowit gives an algorithm for minimizing the number of chains in one direction with worst-case running time $\Theta(n \log n)$ [18], which is optimal [5].

If both ascending and descending chains are allowed simultaneously, then the minimal number of chains is $O(\sqrt{n})$, and finding a decomposition into the minimal number of chains is NP-hard [9]. However, an algorithm of Fomin, Kratsch, and Novelli achieves a constant-factor approximation of the minimal number of chains in $O(n^3)$ time [10]. An algorithm of Yang, Chen, Lu, and Zheng generates a decomposition into at most $\lfloor\sqrt{2n + 1/4} - 1/2\rfloor$ chains of both types (which is the minimal number for worst-case data) in $O(n^{3/2})$ time [19], using techniques developed by Bar-Yehuda and Fogel [3]. They do not prove any guaranteed approximation factor when the minimal number of chains is $o(\sqrt{n})$, but comment that in practical experiments their algorithm often achieves very close to the constant-factor approximation value.

2.2. Data structures for orthogonal range search

Many efficient data structures exist for the two-dimensional orthogonal range search problem. For instance, R-trees [11] are a multidimensional extension of B-trees. An R-tree is a height-balanced tree where each node represents a rectangular region of the underlying space. Thus, the data structure divides the space with hierarchically nested (and possibly overlapping) minimum bounding rectangles. The search algorithm descends the tree, recursing into every subtree whose bounding rectangle overlaps the query. In the worst case a search could be forced to examine the entire tree in $O(n)$ time, even when the query rectangle is empty. However, R-trees are simple to implement, use linear space, tend to perform much better in practice than the theoretical worst case, and are popular as a result.

Range trees [14] support multidimensional range queries by generalizing balanced binary search trees to multiple dimensions. The data points are indexed along one dimension in a standard balanced binary search tree. At each node v of that tree, we collect all the descendants of v and store a new balanced binary search tree of all those points indexed along the second dimension. A

Table 1: Summary of orthogonal range query results; n is the number of points in the database, m is the number of points returned, and k is the number of chains.

Data structure	Ref.	Worst-case search time	Space
R-trees	[11]	$O(n)$	$O(n)$
kd-trees	[4, 15]	$O(\sqrt{n}+m)$	implicit
PR-trees	[1]	$O(\sqrt{n}+m)$	$O(n)$
Range trees	[14]	$O(\log n + m)$	$O(n \log n)$
Nekrich	[17]	$O(\log n + m \log^{\epsilon} n)$	$O(n)$
This paper		$O(\log n + k + m)$	$O(n)$
This paper		$O(k \log n + m)$	implicit

rectangle query descends the first tree to do a one-dimensional range search in $O(\log n)$ time, then searches along the other dimension for an overall time of $O(\log^2 n+m)$. More advanced techniques, like fractional cascading [6], allow the two-dimensional search time to be reduced to $O(\log n + m)$; and the technique can also be extended to higher dimensions at some cost in search time.

Alternative solutions exist that require linear space like R-trees but improve on the worst-case search time. Kanth and Singh show that $O(\sqrt{n}+m)$ worst-case search time is optimal for non-replicating (or linear-space) data structures [12]. Bentley achieves it with kd-trees [4], which recursively divide a k-dimensional space with hyperplanes. Munro describes an *implicit* kd-tree, with optimal search time and no storage used beyond that of the points themselves [15]. Arge *et al.* describe priority R-trees, or PR-trees [1], also with $O(\sqrt{n} + m)$ worst-case search time. In a recent result, Nekrich [17] presents a data structure that uses linear space with search time $O(\log n + m \log^{\epsilon} n)$, trading suboptimal performance in m for better performance in n. See Table 1 for a comparison of methods.

To summarize, R-trees are practical, but do not have proven good worst-case search times, and range trees have an impractical $O(n \log n)$ space requirement. There are alternative solutions requiring linear space and providing better search time. However, none of these can profit from "easy" data. Here we present an *adaptive* data structure. When the data can be decomposed into a small number of monotonic chains, our search performance improves. If the number of chains $k = o(\sqrt{n})$, we surpass the performance of optimal-time linear-space data structures [1, 4, 12, 15].

3. Finding Untangled Chains

In the next section we describe an adaptive algorithm and data structure for two-dimensional orthogonal range search on data decomposed into a union of monotonic chains. The data structure performs better when there are fewer chains. Furthermore, although the worst-case asymptotic time does not depend on this, we can search more efficiently by assuming that the chains are untangled: successive data points can be connected with line segments with no segments

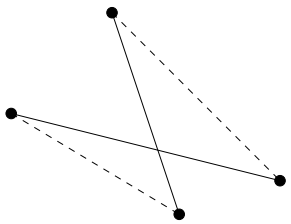


Figure 1: Untangling a pair of segments.

intersecting. That raises the question of how to find an optimal untangled chain decomposition, which we resolve in this section.

Although our data structure asks for an optimal decomposition into chains with both ascending and descending monotonic chains allowed, it actually functions by splitting the points into the two directions as a preprocessing step and then considering the two directions separately; chains are only required to be untangled with respect to other chains of the same type. The untangling problem of interest to us, then, is how to decompose a set of points into a minimal number of untangled chains all in one direction (without loss of generality, descending). We assume that points in the input set are in general position.

As shown in Fig. 1, we can remove any single intersection from a chain decomposition by replacing two intersecting segments (represented by solid lines in the figure) with two that do not intersect (represented by dashed lines). The number of chains remains unchanged, and the operation strictly reduces the total Euclidean length of all chains. Only a finite number of distinct values are possible for the total length, so it follows that any set of chains can be transformed in a finite time into an untangled set of the same number of chains, and the minimum number of untangled chains is the same as the minimum number of possibly-tangled chains.

However, that argument proves only that the time to find the untangled chains is finite. Finding tangles to remove requires a search, and each untangling move could introduce many new tangles as the new segments intersect other existing segments, making the untangling procedure expensive. Van Leeuwen and Schoone show that such a process must terminate after $O(n^3)$ moves [13]. They describe an $O(n^2)$ exhaustive search to find each tangle, for an overall time of $O(n^5)$. Their work is on postprocessing of Travelling Salesman Problem solutions, for which a polynomial-time solution suffices. We describe an algorithm for finding a minimal number of untangled chains in $O(k^2 n)$ time, where k is the number of chains.

3.1. Untangling Monotonic Chains

Given two points $p_i, p_j \in P$, we say that the edge or line segment $\langle p_i, p_j \rangle$ is *valid* if $x(p_i) \leq x(p_j)$ and $y(p_j) \leq y(p_i)$. We also say that points p_i and p_j are *compatible* if either $\langle p_i, p_j \rangle$ or $\langle p_j, p_i \rangle$ is valid. A *chain* is a sequence of edges $C = \{\langle p_1, p_2 \rangle, \langle p_2, p_3 \rangle, \ldots, \langle p_{m-1}, p_m \rangle\}$ where each one is valid. Define $pts(C)$

to be the set of all endpoints of edges in C. A *subchain* S of C is a contiguous subset of the edges $\{\langle p_i, p_{i+1}\rangle, \ldots, \langle p_{i+\ell-1}, p_{i+\ell}\rangle\}$, where $i + \ell \leq m$. We call ℓ the length of S. We refer to the start and end points of a subchain as *terminals*.

Now we can define the basic concept of tangling.

Definition 1. If two chains C_1 and C_2 contain edges $e \in C_1$ and $f \in C_2$ such that e intersects f, such an intersection is called a *tangle*. A pair of chains is *tangled* if one or more tangles exist between those two chains. Often we refer to a single chain as tangled, and in this context the existence of another chain with which it is tangled is implied.

Our goal is to generate the minimum number of descending and untangled chains. We start with the algorithm proposed by Supowit [18] for finding a minimal number of same-direction monotonic chains (possibly tangled). Let A be a chain and $\text{miny}(A) = \min\{y|(x, y) \in pts(A)\}$. Let $P = \{p_1, p_2, \ldots, p_n\}$ be the data points sorted by increasing x-coordinate. Supowit's algorithm does a left-to-right pass over P, either adding each point to an existing chain, or creating a new chain for the point. Upon processing a point p_i, all points p_j with $j < i$ are already part of some chain. Among all existing chains whose right endpoint is above p_i, the algorithm adds p_i to the chain whose right endpoint is lowest. If no such chain exists, a new chain is created with p_i as the only point. The pseudocode of Supowit's algorithm is shown in Algorithm 1.

Algorithm 1 – Supowit($\mathbf{p_1} \ldots \mathbf{p_n}$)

```
1: S ← ∅
2: for i = 1...n, where x(p_i) < x(p_j) ∀i < j ≤ n do
3:    let S' = {A ∈ S, miny(A) ≥ y(p_i)}
4:    if S' ≠ ∅ then
5:       let A_0 = argmin_A{miny(A), A ∈ S'}
6:       append p_i to pts(A_0)
7:    else
8:       add p_i as a chain to S
9: return S
```

We will show that any tangles produced by Algorithm 1 are of a special form that enables us to perform the untangling efficiently. For convenience we define notation for the sets of edges that could ever exist, edges that come from Algorithm 1, and some geometric sets used in the proofs.

Definition 2. For a given set of points P, let $\mathcal{L}(P)$ contain every valid monotonic descending edge between two points in P. That is, given two points $p_i, p_j \in P$, $\langle p_i, p_j\rangle \in \mathcal{L}(P)$ if and only if $x(p_i) \leq x(p_j)$ and $y(p_j) \leq y(p_i)$. Furthermore, for a given set of points P, define $\mathcal{L}^*(P)$ to contain every edge of every chain created by running Algorithm 1 on P.

Definition 3. An edge $\langle p_i, p_j\rangle$ with endpoints p_i and p_j induces two open half-planes. Define $H^+(\langle p_i, p_j\rangle)$ as the open half-plane that contains the point

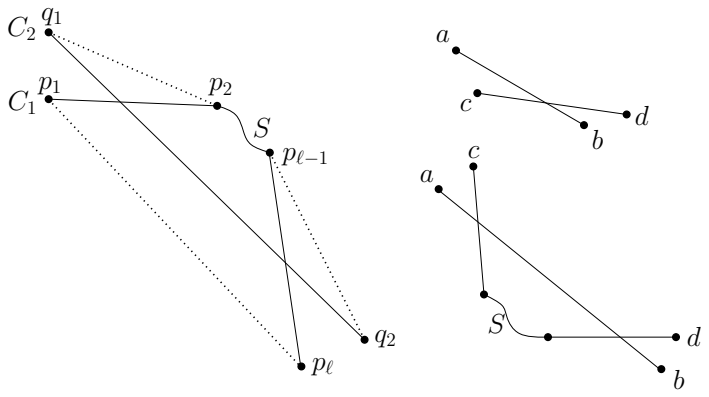


Figure 2: (Left) Valid tangles (v-tangles) generated by Algorithm 1. (Right) Two examples of invalid tangles, which cannot be generated by Algorithm 1. The segment S represents an arbitrary subchain of C_1.

$(x(p_i)+1, y(p_i)+1)$. Similarly, define $H^-(\langle p_i, p_j\rangle)$ as the open half-plane that contains the point $(x(p_i)-1, y(p_i)-1)$.

Now we define the special form of well-behaved tangles as follows.

Definition 4. Suppose we have two chains C_1 and C_2 with edges $\langle q_1, q_2\rangle \in C_2$ and $\langle p_1, p_2\rangle, \ldots, \langle p_{\ell-1}, p_\ell\rangle \in C_1$ such that $p_1 \in H^-(\langle q_1, q_2\rangle)$, $p_\ell \in H^-(\langle q_1, q_2\rangle)$, and $p_i \in H^+(\langle q_1, q_2\rangle)$ for all $1 < i < \ell$. We call such a tangle a *valid* tangle, abbreviated as *v-tangle*. Fig. 2 shows examples of valid and invalid tangles. In the figure, S stands for a subchain and the dotted lines show the new edges that would be added by untangling the v-tangle. We call $\langle q_1, q_2\rangle$ the *upper part* of the v-tangle, and $\langle p_1, p_2\rangle, \ldots, \langle p_{\ell-1}, p_\ell\rangle$ the *lower part*.

Now we can prove the following lemma.

Lemma 1. *All tangles created by Algorithm 1 are v-tangles.*

Proof. Suppose Algorithm 1 on input point set P generated chains C_1 and C_2 with a tangle between edges $\langle p_i, p_{i+1}\rangle \in C_1$ and $\langle p_j, p_{j+1}\rangle \in C_2$. We will show that for every possible ordering of these points the created tangle is a v-tangle; otherwise we reach a contradiction. For this purpose, we will fix $\langle p_i, p_{i+1}\rangle$ and consider the cases where p_j and p_{j+1} are located in each of the quadrants defined by p_i and p_{i+1}, respectively. We will name each case a-b, where a and b are the quadrants where p_j and p_{j+1} are located, respectively (see Fig. 3).

- 1-1 (and, symmetrically, 3-3): In these cases no tangle exists; $\langle p_i, p_{i+1}\rangle$ does not intersect with $\langle p_j, p_{j+1}\rangle$.
- 2-2 (and 2-3, 1-2, 1-3): Upon processing p_{j+1}, Algorithm 1 would have connected this point to p_i, since p_i is lower than p_j and would have not

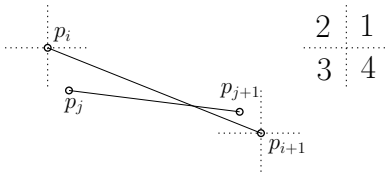


Figure 3: Possible cases for Lemma 1. The configuration in this example is 4-2.

yet been connected to a point to its right. Therefore these cases cannot occur in the output of Algorithm 1.

- 2-1 (and symmetrically 4-3): Since $\langle p_i, p_j \rangle \notin \mathcal{L}^*(P)$, there must exist edges $\langle p_{i-\ell}, p_{i-\ell+1} \rangle, \ldots, \langle p_{i-1}, p_i \rangle \in C_1$ for some $\ell \geq 1$ such that $\langle p_j, p_{i-\ell+1} \rangle \in \mathcal{L}(P)$ and $\langle p_j, p_{i-\ell} \rangle \notin \mathcal{L}(P)$. Such a point $p_{i-\ell}$ must exist and it must be the case in which $x(p_{i-\ell}) \leq x(p_j)$, because otherwise, Algorithm 1 would have added p_j to C_1. Hence, the edges $\langle p_{i-\ell}, p_{i-\ell+1} \rangle, \ldots, \langle p_{i-1}, p_i \rangle$ form the lower part of a v-tangle.

- 2-4 (and 1-4, symmetrically 4-2 and 3-2): Since $\langle p_{i+1}, p_{j+1} \rangle \in \mathcal{L}(P)$ but $\langle p_{i+1}, p_{j+1} \rangle \notin \mathcal{L}^*(P)$, there exist edges $\langle p_{i+1}, p_{i+2} \rangle, \ldots, \langle p_{i+\ell}, p_{i+\ell+1} \rangle \in C_1$ for some $\ell \geq 1$ such that $\langle p_{i+\ell}, p_{j+1} \rangle \in \mathcal{L}(P)$ and $\langle p_{i+\ell+1}, p_{j+1} \rangle \notin \mathcal{L}(P)$. Such a point $p_{i+\ell+1}$ must exist and it must be the case that $x(p_{i+\ell+1}) \leq x(p_j)$, because otherwise, Algorithm 1 would have added p_j to C_1. Therefore, $\langle p_{i+1}, p_{i+2} \rangle, \ldots, \langle p_{i+\ell}, p_{i+\ell+1} \rangle$ is the lower part of a v-tangle.

- 4-1 (and 4-4, 3-1, 3-4): Upon processing p_{i+1}, Algorithm 1 would have connected this point to p_j instead of p_i, since p_j is lower than p_i and would have not yet been connected to a point to its right. □

Since only v-tangles are possible in the output of Algorithm 1, there is an intuitive ordering on the set of chains. Loosely speaking, if chain C_1 starts on one side of C_2, then it must end on that side as well; since all tangles are v-tangles, the number of times C_1 crosses C_2 must be even. To formalize this ordering, suppose we run Algorithm 1 on P and it generates k chains. We can create a set of k points $Q = \{q_1, \ldots, q_k\}$ such that $x(q_i) < x(q_{i+1})$, no two points in Q are compatible with each other, but every point in Q is compatible with every point in P. Formally, $\langle q_i, q_j \rangle \notin \mathcal{L}(P \cup Q)$ for all $1 \leq i, j \leq k$ where $i \neq j$, and for all $q \in Q$ and $p \in P$, $\langle q, p \rangle \in \mathcal{L}(P \cup Q)$. Then, if we execute Algorithm 1 again on $P \cup Q$, each q_i will be added to a single chain C_i, and we can order the chains based on these points. Thus, C_i and C_j, for $1 \leq j < i \leq k$, are referred to as the *upper chain* and *lower chain*, respectively. We will assume we have such a set at the beginning of the chains and another at the end in order to avoid special boundary cases. Note that because $k \leq n$, adding these $2k$ extra points does not affect the asymptotic running time of the algorithm.

With this ordering in mind, we now discuss how to untangle a v-tangle.

Remark 1. Given a v-tangle, as shown at left in in Fig. 2, we can untangle it by using the dotted lines as edges. Essentially, this is moving S to be part of C_2. It does not matter how the points move among chains.

We use this idea in Algorithm 2, which will become a building block for the final untangling algorithm. We call it the *untangling pass*.

Algorithm 2 – Untangling-Pass(P)

1: Run *Supowit*(P) to get chains $C_1, \ldots, C_k$ where C_k is the uppermost chain
2: **for** $i = k$ down to 1 **do**
3: **for** $j = i - 1$ down to 1 **do**
4: Find and untangle all v-tangles between C_i and C_j
5: Return $C_1, \ldots, C_k$

Since we are removing tangles in a specific way, we have to argue that all the tangles we encounter by running an Untangling-Pass are in fact v-tangles. The following lemma makes that argument.

Lemma 2. *Suppose a v-tangle between C_i and C_j is untangled by Algorithm 2, where C_i is the upper chain. Any tangles between C_i and C_ℓ where $\ell < j$ may have been altered. However, the remaining tangles are still v-tangles.*

Proof. Before the tangle is removed, there is a v-tangle t_{ij} between C_i and C_j. Suppose there exists another v-tangle $t_{i\ell}$ between C_i and C_ℓ, where $\ell < j$, which is altered by untangling t_{ij}. Then there must also be a v-tangle $t_{j\ell}$ between C_j and C_ℓ (unless t_{ij} and $t_{j\ell}$ are nested, in which case it is easy to see that $t_{i\ell}$ does not exist after untangling). Since $\ell < j$, there is an edge $e \in C_j$ which is the upper part of $t_{j\ell}$ (recall Definition 4). The edge e must also be involved in the tangle t_{ij} as one of the edges in the lower part of that tangle. Let edge f be the upper part of t_{ij}. If both endpoints of e are in $H^+(f)$, then $t_{i\ell}$ becomes $t_{j\ell}$. Otherwise, e must also be one of the two intersecting lower edges in t_{ij}. By untangling t_{ij} we are adding a new edge e' to C_i, where e' shares one of the endpoints of e. This leads to one of two cases:

1. $e' \in C_i$ is now the upper part of a v-tangle with C_ℓ; or
2. e' is not involved in a tangle with C_ℓ, and therefore it is not a problem. □

In order to argue that all the tangles existing when we untangle the upper chain from the rest are v-tangles, we need to prove a slightly stronger statement that will help with the induction required in the following steps of the proof.

Lemma 3. *Consider the set of points P' in chains $C_1, \ldots, C_{k-1}$ after running the first iteration of Untangling-Pass (i.e., $i = k$ on line 2 of Algorithm 2). If we run Algorithm 1 with input P', the resulting set of chains is exactly $C_1, \ldots, C_{k-1}$.*

PROOF. Consider the uppermost chain C_k and any v-tangle formed in part by the edges $\langle p_1, p_2\rangle, \langle p_2, p_3\rangle, \ldots, \langle p_{\ell-1}, p_\ell\rangle$ in chain C_j, $j < k$, at the moment we untangle it from C_k. The untangling process will add $p_2, \ldots, p_{\ell-1}$ to C_k and create the edge $\langle p_1, p_\ell\rangle$. We need to prove that this edge would have been created by Algorithm 1 if the points $R = \{p_2, \ldots, p_{\ell-1}\}$ had been removed.

Assume p_ℓ had been connected to a different point $p_r \neq p_1$. In that case we know that $\langle p_r, p_\ell\rangle \in \mathcal{L}^*(P \setminus R)$. Thus, $y(p_r) \leq y(p_1)$, and both points must be available when p_ℓ is added. This implies that p_r was available when we processed $p_2, \ldots, p_\ell$ originally. Let r', $2 \leq r' \leq \ell$, be the minimum such that $y(p_{r'}) \leq y(p_r)$. Since p_r was available, $p_{r'}$ would have been connected to p_r originally, contradicting the existence of C_j. □

Ideally, after running the untangling pass algorithm on point set P, we would like all tangles to be removed. Unfortunately, it could be the case that when untangling a pair of chains C_i and C_j (where C_j is the upper chain), a new tangle between C_j and a chain C_u with $u > j$ is created.[2] However, these tangles can only be of a special kind, which we call *reverse v-tangles*.

Definition 5. Suppose we have two chains C_1 and C_2 with edges $\langle q_1, q_2\rangle \in C_1$ and $\langle p_1, p_2\rangle, \ldots, \langle p_{\ell-1}, p_\ell\rangle \in C_2$ such that $p_1 \in H^+(\langle q_1, q_2\rangle)$, $p_\ell \in H^+(\langle q_1, q_2\rangle)$, and $p_i \in H^-(\langle q_1, q_2\rangle)$ for all $1 < i < \ell$. We call such a tangle a *reverse v-tangle*. The right bottom tangle in Fig. 2 is a reverse v-tangle.

The following lemmas show that an untangling pass can only create reverse v-tangles.

Lemma 4. *When $i = k - 2$ in Algorithm 2, only reverse v-tangles can exist between chains C_k and C_{k-1}.*

PROOF. Untangling C_k with C_i, $1 \leq i < k$, will never create a tangle between the intermediate chains $C_j$and C_k, where $i < j < k$, since every untangling operation only adds points to the upper chain that are higher than existing edges. Combining this fact with Lemmas 2 and 3, the upper chain k is untangled from all the other chains just before we start untangling chain $k - 1$.

Now, the only possibility remaining is that the upper chain gets tangled as a consequence of untangling C_{k-1} with C_i, $1 \leq i < k - 1$, in Algorithm 2. We now argue that only reverse v-tangles can be created between C_k and C_{k-1}.

Consider a v-tangle like the one depicted in Fig. 4. Recall that an untangling operation would create the edges shown in dotted lines, plus the edge $\langle c, d\rangle$. In order for this operation to create a tangle between C_{k-1} and C_k, either a point $q \in pts(C_k)$ has to be in the triangle A created by the left dotted line, or a point $r \in pts(C_k)$ must be in the triangle B created by the right dotted line.

Since the upper chain is untangled from the rest of the chains, if it contains a point in either of the two triangles, then the chain must enter and exit through

[2]The previous version of this work [2] overlooked this situation, hence the different strategy described in the present version.

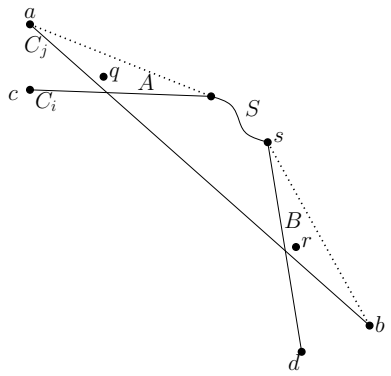


Figure 4: Illustration of cases considered in Lemma 4.

the dotted line, forming a reverse v-tangle, otherwise it would have been tangled. Note that this argument extends to the case when q or r are subchains. □

Lemma 5. *If any tangles exist after one pass of Untangling-Pass, then they must be reverse v-tangles.*

Proof. Consider $i = k$ in Algorithm 2. By Lemma 4, C_k can only participate in reverse v-tangles with chain C_{k-1} after iteration $i = k - 1$. If we remove C_{k-1} when running $i = k - 2$, Lemma 4 holds for C_k and C_{k-2}, so all tangles between C_k and lower chains after Untangling-Pass are reverse v-tangles.

For every $i \in \{1, \ldots, k-1\}$, the arguments hold because we have a set of chains possessing the same properties as that returned by Algorithm 1 (by Lemma 3); in other words, the same situation as if the untangled upper chain had not existed. □

We can now state our untangling algorithm, shown in Algorithm 3. The algorithm executes k passes of Algorithm 2, extracting the current lowest chain at the end of each pass. We prove the correctness of the untangling algorithm by induction. The base case is that after the first pass of the algorithm, the first (lowest) chain has been untangled.

We use the notation C_q^p to denote chain q output by Supowit's algorithm in pass p, and $\bar{C}_q^p$ denotes chain q after Algorithm 3 has completed pass p. Occasionally, we use $\hat{C}_q^p$ to refer to chain q at some intermediate stage during untangling pass p. Finally, let $E(p)$ denote the set of all edges that existed during pass p. Note that $\bigcup_{q=1}^{k} \left(C_q^p \cup \bar{C}_q^p\right) \subseteq E(p)$, but other edges may be created or destroyed during pass p.

Note that Algorithm 1 is equivalent to Bar-Yehuda and Fogel's algorithm for computing layers of minima [3]. Therefore, chains generated by Algorithm 1 have the following useful property [3]:

Definition 6. For points a and b we say a is dominated by b, b dominates a, or $a \prec b$, if $x(a) \leq x(b)$ and $y(a) \leq y(b)$. Supowit's algorithm creates chains that

Algorithm 3 – Untangled-Chains(P)

Let $k \leftarrow$ minimum number of chains to cover P
Add k dummy points at the left such that they are pairwise incompatible with each other, but each one is compatible with every point in P.
Add k dummy points at the right such that they are pairwise incompatible with each other, but each one is compatible with every point in P.
Let $F = \varnothing$
for $p = 1$ to k **do**
    Obtain $C_p, \ldots, C_k$ using Untangling-Pass(P)
    $P \leftarrow P \setminus pts(C_p)$
    $F = F \cup \{C_p\}$
Return F

have the following recursively-defined property: for $1 \leq i \leq k-1$, C_i is created such that for all $b \in pts(C_i)$ there exists no point $a \in \bigcup_{j=i+1}^{k} pts(C_j)$ such that $a \prec b$, and C_i is maximal. C_k contains all remaining points. Thus, C_1 contains all of the points which dominate no other points, C_2 contains all of the points which dominate points from C_1, and so on. We refer to this property as the *dominance property.*

Lemma 6. *After one Untangling-Pass, $\bar{C}_1^1$ has no tangles.*

Proof. By Lemma 5, if the lower chain is tangled after an Untangling-Pass then it is a reverse v-tangle. Consider the edge $e \in \bar{C}_1^1$ for which there is a point $a \in H^-(e)$. If many such points exist, choose a to be one which does not dominate any other. By the dominance property, $a \in pts(C_1^1)$, which means that an edge $e' \in E(p)$ must have existed such that $a \in H^+(e')$ and e' was the upper part of a v-tangle involving a. However, by Lemmas 1 and 2, we arrive at a contradiction: one of the endpoints of e must be in $H^+(e')$ and would in the lower part of the v-tangle involving a. □

Now we show that no tangles will be created involving the p-th chain after pass p. Showing this will prove the correctness of the untangling algorithm. Assume that $\bar{C}_1^p, \ldots, \bar{C}_p^p$ are all mutually untangled, and $\bar{C}_p^p$ is untangled with all chains above. Note that $\bar{C}_p^p = \bar{C}_p^q = C_p^q$ for $p < q \leq k$, since the algorithm does not touch $\bar{C}_p^p$ after pass p.

Lemma 7. *If $\bar{C}_p^q$ is involved in a tangle for $p < q \leq k$, then the tangle must be a v-tangle.*

Proof. If a tangle forms that is not a v-tangle, it implies that a point from $P \setminus (pts(\bar{C}_1^p) \cup \cdots \cup pts(\bar{C}_p^p))$ is located below $\bar{C}_p^q$, which contradicts the assumption that $\bar{C}_p^q$ is untangled with all chains above. □

Using Lemma 7, we can now rule out the possibility of v-tangles occurring with $\bar{C}_p^p$ in subsequent passes through a series of lemmas. The following definitions are required for Lemmas 8 to 11:

Definition 7. A point b is called a *displaced point* if $b \in pts(C_i^p)$, and $b \in pts(C_{i+1}^{p+1})$, where $1 \leq p < k$ and $p \leq i < k$. If a point is not displaced between passes p and $p+1$ we refer to it as *original*.

Lemma 8. *If an edge $\langle b_1, b_2 \rangle$ output by Supowit's algorithm at the beginning of pass $p+1$ was not output by Supowit's algorithm at the beginning of pass p, then either b_1 or b_2 is a displaced point.*

Proof. The proof follows by an invariant property on the edges in the chains, starting from C_{p+1}^{p+1}. When we remove $\bar{C}_p^p$ after the untangling pass in pass p, consider the set $R = pts(C_p^p) \setminus pts(\bar{C}_p^p)$. Since C_{p+1}^{p+1} is maximal by the dominance property, R will be inserted into $pts(C_{p+1}^{p+1})$. The set of points in $S = pts(C_{p+1}^p) \setminus pts(C_{p+1}^{p+1})$ will be points such that for all $s \in S$, there exists an $r \in R$ such that $r \prec s$. All of the edges in C_{p+1}^p which do not have endpoints in S, and which have not been replaced by edges containing endpoints from R, remain in the output at the beginning of pass $p+1$. Thus the property holds for C_{p+1}^{p+1}. By setting $R = pts(C_{p+i}^p) \setminus pts(C_{p+i}^{p+1})$ and $S = pts(C_{p+i+1}^p) \setminus pts(C_{p+i+1}^{p+1})$ for $1 \leq i < k-p$ we can continue this argument, proving the lemma. □

Next we have two lemmas to show that points displaced between pass p and $p+1$ were involved in an untangling operation during pass p. The first shows the existence of sequences of points based on the dominance property, and the second uses these sequences to show that the v-tangles must have existed. The second lemma is the key lemma used in the proof of correctness.

Lemma 9. *If a_q is a displaced point from C_q^p to C_{q+1}^{p+1}, then there must exist a sequence of points $a_p, \ldots, a_{q-1}$ such that for $p \leq i \leq q-1 < k$, a_i is a displaced point from C_i^p to C_{i+1}^{p+1} and $a_i \prec a_{i+1}$.*

Proof. Follows from the dominance property: since the chains are maximal, a_q would not move to a higher chain unless another point dominated by a_q took its place. □

Lemma 10. *If a_q is a displaced point from C_q^p to C_{q+1}^{p+1}, then there must have been an edge $e \in E(p)$ such that $a_q \in H^+(e)$, and e was the upper part of a v-tangle involving a_q during pass p.*

Proof. We give a proof by induction. In the base case, a point which moves from C_p^p to C_{p+1}^{p+1} must have been involved in a v-tangle, since we remove $\bar{C}_p^p$ from consideration in pass $p+1$. Inductive step: Consider a sequence of points $a_p, \ldots, a_q$ from Lemma 9, such that for $p \leq i \leq q-1 < k$, a_i is a displaced point from C_i^p to C_{i+1}^{p+1} and $a_i \prec a_{i+1}$. Assume that there exists edges $e_i \in E(p)$ such that $a_i \in H^+(e_i)$ and a_i is in the lower part of a v-tangle that has upper part e_i. Consider a_q, and assume that no edge $e_q \in E(p)$ exists such that $a_q \in H^+(e_q)$ and a_q is part of a v-tangle that has upper part e_q. Since $a_{q-1} \prec a_q$, e_{q-1} comes from a chain $C_{q+j'}^p$ for $j' \geq 1$; such an edge cannot come from C_q^p due to the

orientation of a_q and a_{q-1}. However, this implies that $a_q \in H^+(e_{q-1})$. Since $C_{q+j'}$ is untangled with C_q before C_{q-1}, a_q would be untangled with e_{q-1}, and we have arrived at a contradiction. □

We now make use of Lemma 10 to complete the inductive step of the proof of correctness.

Lemma 11. *Any sequence of untangling operations occurring in pass $p+1$ cannot form a v-tangle with C_p^{p+1}.*

PROOF. Suppose a v-tangle v_1 is created in pass $p+1$, and the lower part of v_1 consists of a subchain of C_p^{p+1}. For such a tangle to have been created, either v_1 was output by Supowit's algorithm at the beginning of pass $p+1$, or a sequence of untangling operations occurred, starting with v-tangle v_t and ending with the creation of v_1. This follows from Lemmas 1 and 7, and the untangling pass (Algorithm 2). It also must be the case that at least one of the tangles in the sequence $v_t, \ldots, v_1$ contains displaced points. Otherwise, all of the edges participating in the sequence would have been present in pass p by Lemma 8.

Before continuing, we require the following definition. Consider an edge $e \in E(p+1)$, such that e is the upper part of a v-tangle v, and the lower part of v is a subchain S. We define an edge $e' \in E(p)$ to be *equivalent* to e if for each $a \in pts(S)$ where $a \in H^+(e)$, $a \in H^+(e')$ and S forms a valid v-tangle with e'.

Consider the v-tangle with the largest index t', $1 \leq t' \leq t$, such that $v_{t'}$ consists of displaced points which *cause* $v_{t'}$ (i.e., $v_{t'}$ would not exist if the displaced points in $v_{t'}$ were not present). If the displaced points are in the upper part of $v_{t'}$, $t' \geq 1$, we will show that an edge equivalent to the upper part of $v_{t'}$ existed in pass p. Similarly, if the displaced points are in the lower part of $v_{t'}$, $t' > 1$, we show that an edge equivalent to the upper part of $v_{t'-1}$ existed in pass p by assuming the existence of an edge equivalent to the upper part of $v_{t'}$. As an invariant property, this is sufficient to prove the lemma. We now describe these two cases:

1. The upper part of v-tangle $v_{t'}$ contains at least one displaced point. Call the upper part of the tangle, from left to right, $\langle u_1, u_2 \rangle$ and the points in the lower part $\{l_1, \ldots, l_\ell\}$ where l_1 and l_ℓ are the points which are not in $H^+(\langle u_1, u_2 \rangle)$. Without loss of generality, assume that u_1 is a displaced point. By Lemma 10, there exists an edge $e \in E(p)$ such that u_1 was in the lower part of a v-tangle with e. If $u_2 \in H^+(e)$ we are done. If not, then consider the first original point b_1 to the left of u_1, such that $b_1 \in H^-(e)$. By Lemma 2, and the existence of e, any original points to the right of b_1 and to the left of u_1 must have been involved in the lower part of a v-tangle during pass p. If u_2 is an original point then we have shown the existence of an edge $e' \in E(p)$ equivalent to $\langle b_1, u_2 \rangle$; thus, for each $l \in \{l_2, \ldots, l_{\ell-1}\}$, $l \in H^+(e')$. Otherwise, u_2 is a displaced point, and we can locate b_2, the first original point to the right of u_2, such that $\langle b_1, b_2 \rangle$ satisfies the invariant.

2. The lower part of v-tangle $v_{t'}$ contains a subchain A of displaced points: use $\hat{C}_q^{p+1}$ to denote the lower chain, and $\hat{C}_r^{p+1}$ the upper chain, for some $p < q < r \leq k$. Let $e = \langle u_1, u_2 \rangle$ be the upper part of $v_{t'-1}$ after $v_{t'}$ is untangled. Since $t' > 1$, if u_1 or u_2 is a displaced point, we can apply the argument from case 1. Therefore, we assume that both u_1 and u_2 are original points. Consider the points b_1 and b_2, connected to the left and right terminals of A, respectively. Also consider the subchain A' of points displaced from C_q^{p+1} by A:
   (a) If $A' = \emptyset$ then $e' = \langle b_1, b_2 \rangle \in E(p)$. If $u_1 = b_1$ and $u_2 = b_2$ we are done since e' is equivalent to e. Otherwise, note that either $u_1 = b_1$ or $u_2 = b_2$ by the assumption that A causes $v_{t'}$ and that no endpoint in e is a displaced point. In either case, we can apply the invariant that an edge equivalent to the upper part of $v_{t'}$ existed in pass p, which in turn applies the existence of e', an edge equivalent to either $\langle u_1, b_2 \rangle$ or $\langle b_1, u_2 \rangle$, depending on which point differs.
   (b) Likewise, if $A' \neq \emptyset$, then by Lemma 10 there existed a series of untangling operations by which all of the points in A' were removed from C_q^p during pass p. As in the previous subcase, this resulted in the formation of an edge equivalent to $\langle u_1, u_2 \rangle$.

Since we can maintain the invariant property in both cases, we arrive at a contradiction, completing the lemma. □

Lemma 11 implies that no further v-tangles will be formed with C_p^p in any pass after pass p. Since, by Lemma 7, only v-tangles could be formed, this completes the proof of the correctness of the algorithm.

Theorem 1. *We can untangle k chains using at most k passes of the untangling algorithm, requiring $O(k^2 n)$ time.*

Proof. Lemmas 1–11 guarantee that Algorithm 3 generates a minimal set of untangled chains, so it remains only to establish its running time.

It is clear that untangling two chains C_i and C_j takes time proportional to the sum of the chain lengths. Let ℓ_i^f represent the final length of C_i after the algorithm terminates, and ℓ_j^s represent the length of C_j at the start of the algorithm. Then the running time for one untangling pass can be expressed as:

$$\begin{aligned}\sum_{i=1}^{k}\sum_{j=1}^{i-1}(\ell_i^f + \ell_j^s) &= \sum_{i=1}^{k}(i-1)\ell_i^f + \sum_{i=1}^{k}\sum_{j=1}^{i-1}\ell_j^s \\ &\leq k\left(\sum_{i=1}^{k}\ell_i^f + \sum_{j=1}^{k-1}\ell_j^s\right) \\ &\leq 2kn \ .\end{aligned}$$

Initially running Algorithm 1 requires $O(n \log n)$ time to sort the points. However, observe that running Algorithm 1 k times where each pass generates

at most k chains takes $O(kn \log k + n \log n)$ time, since we only need to sort the points once [7]. Therefore, the cost of running k passes of Algorithm 1 is absorbed by the k passes of 2, and the running time of Algorithm 3 is $O(nk^2)$ time. In the worst case, when $k = \Theta(\sqrt{n})$, this becomes $O(n^2)$. □

4. Adaptive Orthogonal Range Search

If the data points form a single monotonic chain, then the answer to any orthogonal range query must be a contiguous interval of the ordered list of points, and we can find it with two binary searches to identify its start and end. We can store such a data set in $O(n)$ space (for instance, in an array) and answer queries in $O(\log n + m)$ time, where n is the number of data points and m is the number of points returned by the query. Now assume that as a preprocessing step the data points have been decomposed into a minimal number k of monotonic chains. If we store each chain in sorted order and search them all, the query time becomes $O(k \log n + m)$. That is the basic concept underlying our data structure.

A truly optimal decomposition into monotonic chains in both directions would require solving the NP-hard problem of optimally determining whether to assign each point to an ascending or descending chain, but we can come within a constant factor in $O(n^3)$ time with the algorithm of Fomin, Kratsch, and Novelli [10], and that is sufficient to preserve the asymptotic search time of our data structure. The $O(n^{3/2})$ algorithm of Yang, Chen, Lu, and Zheng offers no guarantee of a minimal decomposition, but appears to come close in practice and may be preferable in real applications [19]. In either case, once we have a decomposition of the data points into chains, we separate the ascending and descending chains, and treat the two directions separately, building a data structure for each and running every query on both.

The two-direction minimization algorithms are used only to decide for each point whether it will go into the ascending or descending structure. Having made that decision, we run the algorithm of the previous section to find a minimal set of untangled chains for each direction; doing so cannot increase the number of chains further.

Without loss of generality, we describe the data structure for descending chains here. The ascending case is symmetric. Let $\{C_1, C_2, \ldots, C_k\}$ be the set of untangled descending chains, and let ℓ_i be the length of C_i. Let $r = [x_1, x_2] \times [y_1, y_2]$ be the query range. The points (x_1, y_1) and (x_2, y_2) are the lower left and upper right corners of the query range.

We first find the set of chains that intersect r. Since the chains are untangled and we store them ordered from left to right as described in the previous section, we can find the first chain to pass above the point (x_1, y_1) and the last chain to pass below the point (x_2, y_2), and know that all chains intersecting the query range must be between those two chains in the ordering. Evaluating whether a point is above or below a chain can be accomplished by a simple binary search over the chain in $O(\log n)$ time, so with two binary searches over the chains

we can find the start and end of the range of chains that might intersect r, in $O(\log k \log n)$ time. Let $k' \leq k$ be the number of chains in that subset.

For each of the k' chains that might intersect r, we can perform two more binary searches to find the start and end of the interval of data points within the chain that are actually included in the query range. Note that because of the monotonicity of the chain, this must be a contiguous interval. The time to perform these searches is $O(\log \ell_i)$ for each of the k' chains, and since $\sum_{i=1}^{k} \ell_i = n$, the time for this step is $O(k' \log(n/k'))$.

The number of points m returned by the query also places a lower bound of $\Omega(m)$ on the running time. Summing the times gives the following lemma:

Lemma 12. *Given a set of n points which can be decomposed into k monotonic chains, we can in $O(n^3)$ time construct a linear-space data structure answering two-dimensional orthogonal range search queries in $O(\log k \log n + k' \log(n/k') + m)$ time, where m is the number of points returned and $k' \leq k$ depends on the query.*

Our algorithm is adaptive in the sense that if we have a good set of points, then we will have a small k and this leads to a better search time. The value of m depends on the given query; k' depends on the query and the details of how the chain decomposition was done. The $O(n^3)$ preprocessing time may be improved to $O(n^2)$ in practical cases when the partitioning algorithm of Yang, Chen, Lu, and Zheng gives acceptable results [19]. We can also improve the data structure in one of two other ways: by applying fractional cascading, or by storing it implicitly.

4.1. Fractional cascading

Observe that the basic algorithm performs binary searches for the same keys in separate ordered lists (namely, the chains). Thus, we can use the technique of fractional cascading [6] to speed up the query time and achieve the following result.

Theorem 2. *Given a set of n points which can be decomposed into k monotonic chains, we can in $O(n^3)$ time construct a linear-space data structure answering two-dimensional orthogonal range search queries in either $O(\log n + k + m)$ time or $O(\log k \log n + k' + m)$ time, where m is the number of points returned and $k' \leq k$ depends on the query.*

Proof. To check whether the query rectangle $[x_1, x_2] \times [y_1, y_2]$ intersects a given chain C_i, it is sufficient to perform binary searches on the list of x-coordinates (or y-coordinates) of the points on C_i using x_1 and x_2 (or y_1 and y_2) as search keys. We can then determine whether C_i intersects any of the four edges of the query rectangle using the results of the above four binary searches. This also finds which edge, if any, of C_i intersects each edge of the query rectangle. Therefore, we can report the points on C_i that are located in the query range in $O(\log n + k_i)$ time, where k_i is the number of such points.

To answer orthogonal range search queries using our data structure, we can perform two binary searches on the list of x-coordinates of the points on each chain, and two binary searches on the list of y-coordinates for each chain. Thus, we can store the sorted lists of x-coordinates and y-coordinates corresponding to the monotonically increasing chains separately, and use the technique of fractional cascading [6] to speed up the query time without increasing the asymptotic space cost of our data structure. We augment the data structure for the monotonically decreasing chains using the same approach. This yields a data structure of linear space that supports orthogonal range search in $O(\log n + k + m)$ time.

The bound of $O(\log k \log n + k' + m)$ time can be achieved by locating the start and the end of the range of chains that might intersect the query rectangle, and then using fractional cascading to compute the answer starting from the uppermost chain in this range. □

4.2. Implicit storage

A data structure is *implicit* if it uses no additional storage beyond the space required to encode the data and a constant number of parameters [16]. We now show that our data structure can also be made implicit.

Corollary 1. *A set of n points in the plane can be arranged as an array of n coordinate pairs so that any orthogonal range query over this point set can be answered in $O(\log k \log n + k' \log(n/k') + m)$ time with $O(1)$ working space.*

Proof. Our adaptive algorithm assumes that the coordinates of the points are stored in such a manner that, given two integers i and j, the coordinates of the j-th leftmost point in the i-th monotonic chain can be retrieved in constant time. An obvious way to achieve this is to store the coordinate pairs of the points in the same monotonic chain from left to right as a sub-array, and then concatenate all such sub-arrays into a single array. The number, k, of chains and the indices of the entries storing the coordinate pairs of the leftmost points in each chain are also stored. Thus, $(k+1)\lceil \log n \rceil$ additional bits of storage are required to store the point coordinates. We now show how to encode such information by permuting the coordinate array.

Let ℓ_i be the number of points in the i-th monotonic chain, C_i, for $i = 1, 2, \ldots, k$. Then $\sum_{i=1}^{k} \ell_i = n$. We first construct an array $A[1..n]$ storing the coordinate pairs of all the points as follows: We start with the first chain, and if it has an even number of points, we store the coordinate pair of the j-th leftmost point of C_i in $A[j]$, for $j = 1, 2, \ldots, \ell_1$. Otherwise, we store all but the rightmost point in $A[1..j-1]$ in the same order. We use the next ℓ_2 entries of A to store the points of C_2 from left to right if ℓ_2 is even, and otherwise, we use the next $\ell_2 - 1$ entries to store all but the rightmost point in C_2. We repeat this process until the points in C_k (with the possible exception of its rightmost point) are stored. After completing the above process, all the points in a monotonic chain with an even number of points are stored in A, but the rightmost points in chains with odd numbers of points are not stored. We store

these missing points in the last v entries of A, where v is the number of chains with odd numbers of points, sorted by the number of the chain each such point is in. This way all the points are stored in A, and the following property of A is immediate:

Property 1. Given an odd integer i, where $1 \leq i < n-k$, the two points whose coordinate pairs are stored in $A[i]$ and $A[j]$ are in the same monotonic chain. This first point is also to the left of the second such point.

To perform our adaptive range search algorithm on a point set whose coordinates are stored in A, we require the following additional information: the integer k; an integer, s_i, for each chain C_i that stores the index of the entry that stores the coordinates of the leftmost point in this chain if $\ell_i > 1$ (if $\ell_i = 1$, set s_i to be s_{i+1}); an integer, r_i, for each chain C_i that stores 1 if ℓ_i is even; and the index of the entry in A that stores the rightmost point in C_i if ℓ_i is odd. The above information occupies $(2k+1)\lceil \log n \rceil = O(\sqrt{n} \log n)$ bits.

We next encode the above information by permuting the array A. Property 1 guarantees that, if we swap the pair $(A[i], A[i+1])$, where i is an odd number and $1 \leq i < n-k$, we can still retrieve the coordinates of the point originally stored in $A[i]$ in constant time, as it is to the left of the point stored in $A[i+1]$. Thus, our adaptive range search algorithm works on A with the same additional information if we swap elements of A using the method described above. By permuting each pair of data points in $A[1..n-k]$ using this approach, we can encode $\lfloor (n-k)/2 \rfloor$ bits (this is because we can encode a 1 bit by swapping a pair in the above way, and a 0 bit by leaving the pair as it is). Therefore, we can encode the additional information of $O(\sqrt{n} \log n)$ bits by permuting A, for sufficiently large n, and we denote the permuted array of A by A'. With A', we can decode k in $O(\log n)$ time and the entry in A that stores the leftmost point of C_i in $O(\log n)$ time. After we get the index of the entry storing the leftmost point of C_i, we can also retrieve any other point (with perhaps the exception of the rightmost point) of C_i in $O(1)$ additional time. If ℓ_i is odd, the rightmost point of C_i can be retrieved in $O(\log n)$ time. All this enables us to perform our adaptive range search algorithm using A' without storing additional information, and the claim of this corollary follows immediately. □

5. Conclusions

We have presented a new data structure, for two-dimensional orthogonal range search, that is adaptive to the minimum number of monotonic chains into which the input points can be partitioned. For data which is considered easy in this sense, our data structure outperforms existing alternatives, either in query time or space requirements. Furthermore, we show that our structure can be made implicit, requiring only constant space in addition to the space required to encode the input points.

A recent study [7] compares our data structure in practice against available implementations of *kd*-trees and range trees, showing not only that our adaptive

data structure is competitive in terms of query time against these well known structures, but that it also outperforms them for various practical data sets, even without fractional cascading.

As a contribution of independent interest, we show how to partition a set of two-dimensional points into a minimal number of untangled monotonic chains. This decomposition is a key element of our data structure, and could also be useful in other geometric applications.

Natural directions for future work include extending our structure to higher dimensions—which would involve adapting our untangling algorithm to higher dimensions as well—and adapting our structure to answer other kinds of queries than orthogonal.

[1] L. Arge, M. de Berg, H.J. Haverkort, K. Yi, The priority R-tree: A practically efficient and worst-case optimal R-tree, ACM Transactions on Algorithms 4 (2008) 9:1–9:30.

[2] D. Arroyuelo, F. Claude, R. Dorrigiv, S. Durocher, M. He, A. López-Ortiz, J.I. Munro, P.K. Nicholson, A. Salinger, M. Skala, Untangled monotonic chains and adaptive range search, in: Y. Dong, D.Z. Du, O. Ibarra (Eds.), 20th International Symposium on Algorithms and Computation (ISAAC 2009), Honolulu, Hawaii, USA, December 16–18, 2009, volume 5878 of *LNCS*, Springer, 2009, pp. 203–212.

[3] R. Bar-Yehuda, S. Fogel, Partitioning a sequence into few monotone subsequences, Acta Informatica 35 (1998) 421–440.

[4] J.L. Bentley, Multidimensional binary search trees used for associative searching, Communications of the ACM 18 (1975) 509–517.

[5] P.A. Bloniarz, S.S. Ravi, An $\Omega(n \log n)$ lower bound for decomposing a set of points into chains, Information Processing Letters 31 (1989) 319–322.

[6] B. Chazelle, L.J. Guibas, Fractional cascading: I. a data structuring technique, Algorithmica 1 (1986) 133–162.

[7] F. Claude, J.I. Munro, P.K. Nicholson, Range queries over untangled chains, in: E. Chávez, S. Lonardi (Eds.), Proceedings of the 17th Symposium on String Processing and Information Retrieval (SPIRE 2010), volume 6393 of *LNCS*, Springer, 2010, pp. 82–93.

[8] E.D. Demaine, A. López-Ortiz, J.I. Munro, Adaptive set intersections, unions, and differences, in: SODA, pp. 743–752.

[9] G. Di Stefano, S. Krause, M.E. Lübbecke, U.T. Zimmermann, On minimum k-modal partitions of permutations, Journal of Discrete Algorithms 6 (2008) 381–392.

[10] F.V. Fomin, D. Kratsch, J.C. Novelli, Approximating minimum cocolorings, Information Processing Letters 84 (2002) 285–290.

[11] A. Guttman, *R*-trees: a dynamic index structure for spatial searching, SIGMOD Record (ACM Special Interest Group on Management of Data) 14 (1984) 47–57.

[12] K.V.R. Kanth, A. Singh, Optimal dynamic range searching in non-replicating index structures, in: 7th International Conference on Database Theory (ICDT '99), Proceedings, volume 1540 of *LNCS*, Springer, 1999, pp. 257–276.

[13] J. van Leeuwen, A.A. Schoone, Untangling a traveling salesman tour in the plane, in: Proceedings of the 7th Conference on Graphtheoretic Concepts in Computer Science (WG 81), Hanser Verlag, München, Germany, 1981, pp. 87–98.

[14] G.S. Lueker, A data structure for orthogonal range queries, in: 19th Annual Symposium on Foundations of Computer Science (FOCS '78), IEEE Computer Society Press, Long Beach, Ca., USA, 1978, pp. 28–34.

[15] J.I. Munro, A multikey search problem, in: Proceedings of the 17th Allerton Conference on Communication, Control and Computing, University of Illinois, pp. 241–244.

[16] J.I. Munro, H. Suwanda, Implicit data structures for fast search and update, Journal of Computer Systems Sciences 21 (1980) 236–250.

[17] Y. Nekrich, Orthogonal range searching in linear and almost-linear space, Computational Geometry 42 (2009) 342–351.

[18] K.J. Supowit, Decomposing a set of points into chains, with applications to permutation and circle graphs, Information Processing Letters 21 (1985) 249–252.

[19] B. Yang, J. Chen, E. Lu, S.Q. Zheng, A comparative study of efficient algorithms for partitioning a sequence into monotone subsequences, in: TAMC 2007, volume 4484 of *LNCS*, Springer, 2007, pp. 46–57.