Agglomerative Clustering of Growing Squares

Algorithmica (2022) 84:216–233
https://doi.org/10.1007/s00453-021-00873-0

Agglomerative Clustering of Growing Squares

Thom Castermans[1] · Bettina Speckmann[1] · Frank Staals[2] · Kevin Verbeek[1]

Received: 16 May 2019 / Accepted: 31 August 2021 / Published online: 20 November 2021

Abstract

We study an agglomerative clustering problem motivated by interactive glyphs in geovisualization. Consider a set of disjoint square glyphs on an interactive map. When the user zooms out, the glyphs grow in size relative to the map, possibly with different speeds. When two glyphs intersect, we wish to replace them by a new glyph that captures the information of the intersecting glyphs. We present a fully dynamic kinetic data structure that maintains a set of n disjoint growing squares. Our data structure uses $O\left(n \log n \log \log n\right)$ space, supports queries in worst case $O\left(\log^2 n\right)$ time, and updates in $O\left(\log^5 n\right)$ amortized time. This leads to an $O\left(n\,\alpha(n) \log^5 n\right)$ time algorithm to solve the agglomerative clustering problem. This is a significant improvement over the current best $O\left(n^2\right)$ time algorithms.

Keywords Computational geometry · Kinetic data structures · Range trees

The Netherlands Organisation for Scientific Research (NWO) is supporting T. Castermans (Project Number 314.99.117), B. Speckmann (Project Number 639.023.208), F. Staals (Project Number 612.001.651), and K. Verbeek (Project Number 639.021.541).
A preliminary version of this work appeared in the proceedings of the 13th Latin American Theoretical Informatics Symposium (LATIN), pages 260–274, 2018.

✉ Bettina Speckmann
b.speckmann@tue.nl

Thom Castermans
t.h.a.castermans@tue.nl

Frank Staals
f.staals@uu.nl

Kevin Verbeek
k.a.b.verbeek@tue.nl

1 TU Eindhoven, Eindhoven, The Netherlands

2 Utrecht University, Utrecht, The Netherlands

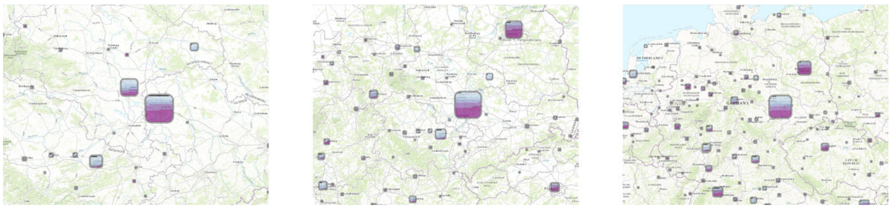

Fig. 1 Zooming out in GlamMap will merge overlapping squares. This figure shows a sequence of three steps zooming out from the surroundings of Leipzig

1 Introduction

We study an agglomerative clustering problem motivated by interactive glyphs in geo-visualization. Our specific use case stems from the eHumanities, but similar visualizations are used in a variety of application areas. *GlamMap* [6][1] is a visual analytics tool which allows the user to interactively explore datasets which contain (at least) the following metadata of a book collection: author, title, publisher, year of publication, and location (city) of publisher. Each book is depicted by a square, color-coded by publication year, and placed on a map according to the location of its publisher. Overlapping squares (many books are published in Leipzig, for example) are recursively aggregated into a larger glyph until all glyphs are disjoint (see Fig. 1). As the user zooms out, the glyphs "grow" relative to the map to remain legible. As a result, glyphs start to overlap and need to be merged into larger glyphs to keep the map clear and uncluttered. It is straightforward to compute the resulting agglomerative clustering whenever a data set is loaded and to serve it to the user as needed by the current zoom level. However, GlamMap allows the user to filter by author, title, year of publication, or other applicable meta data. It is impossible to pre-compute the clustering for any conceivable combination of filter values. To allow the user to browse at interactive speeds, we hence need an efficient agglomerative clustering algorithm for growing squares (glyphs). Interesting bibliographic data sets (such as the catalogue of WorldCat, which contains more than 321 million library records at hundreds of thousands of distinct locations) are too large by a significant margin to be clustered fast enough with the current state-of-the-art $O(n^2)$ time algorithms (here n is the number of squares or glyphs).

In this paper we formally analyze the problem and present a fully dynamic data structure that uses $O(n \log n \log \log n)$ space, supports updates in $O(\log^5 n)$ amortized time, and queries in $O(\log^2 n)$ time, which allows us to compute the agglomerative clustering for n glyphs in $O(n\,\alpha(n) \log^5 n)$ time. Here, α is the extremely slowly growing inverse Ackermann function. To the best of our knowledge, this is the first fully dynamic clustering algorithm which beats the classic $O(n^2)$ time bound.

Formal problem statement. Let P be a set of points in $\mathbb{R}^2$ (the locations of publishers from our example). Each point $p \in P$ has a positive weight p_w (number of books published in this city). Given a "time" parameter t, we interpret the points in P as

[1] http://glammap.net/glamdev/maps/1, best viewed in Chrome.

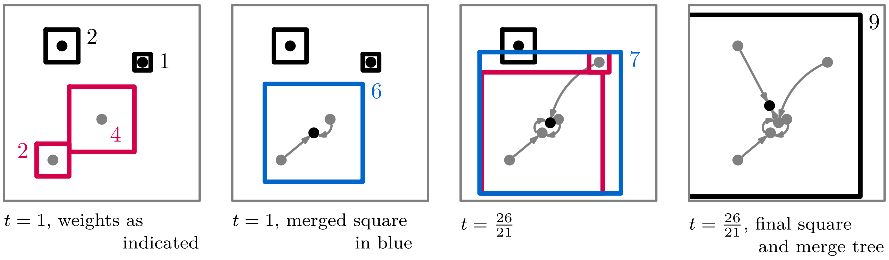
2
1
4
2
6
7
9
$t = 1$, weights as indicated
$t = 1$, merged square in blue
$t = \frac{26}{21}$
$t = \frac{26}{21}$, final square and merge tree

Fig. 2 The timeline of squares that grow and merge as they touch

squares. More specifically, let $\square_p(t)$ be the square centered at p with width tp_w. For ease of exposition we assume all x and y coordinates to be unique. With some abuse of notation we may refer to P as a set of squares rather than the set of center points of squares. Observe that initially, i.e. at $t = 0$, all squares in P are disjoint. As t increases, the squares in P grow, and hence they may start to intersect. When two squares $\square_p(t)$ and $\square_q(t)$ intersect at time t, we remove both p and q and replace them by a new point z, which is located at the weighted average of p and q and has the sum of their weights as its weight. More formally, $z = \omega p + (1 - \omega)q$, with $\omega = p_w/(p_w + q_w)$, and has weight $z_w = p_w + q_w$ (see Fig. 2).

Related Work. Funke, Krumpe, and Storandt [7] introduced so-called "ball tournaments", a related, but simpler, problem, which is motivated by map labeling. Their input is a set of balls in $\mathbb{R}^d$ with an associated set of priorities. The balls grow linearly and whenever two balls touch, the ball with the lower priority is eliminated. The goal is to compute the elimination sequence efficiently. Bahrdt et al. [4] and Funke and Storandt [8] improved upon the initial results and presented bounds which depend on the ratio Δ of the largest to the smallest radius. Specifically, Funke and Storandt [8] show how to compute an elimination sequence for n balls in $O\big(n \log \Delta(\log + \Delta^{d-1})\big)$ time in arbitrary dimensions and in $O\big(Cn \operatorname{polylog} n\big)$ time for $d = 2$, where C denotes the number of different radii. In our setting eliminations are not sufficient, since merged glyphs need to be re-inserted. Furthermore, as opposed to typical map labeling problems where labels come in a fixed range of sizes, the sizes of our glyphs can vary by a factor of 10.000 or more (Amsterdam with its many well-established publishers vs. Kaldenkirchen with one obscure one).

Ahn et al. [2] recently and independently developed the first sub-quadratic algorithms to compute elimination orders for ball tournaments. Their results apply to balls and boxes in two or higher dimensions. Specifically, for squares in two dimensions they can compute an elimination order in $O\big(n \log^4 n\big)$ time. Their results critically depend on the fact that they know the elimination priorities at the start of their algorithm and that they only have to handle deletions. Hence they do not have to run an explicit simulation of the growth process and can achieve their results by the clever use of advanced data structures. In contrast, we are handling the fully dynamic setting with both insertions and deletions, and without a specified set of priorities. In fact, our algorithm can easily be adapted to solve the ball tournament problem for squares without increasing the running time.

Our clustering problem combines both dynamic and kinetic aspects: squares grow, which is a restricted form of movement, and squares are both inserted and deleted. There are comparatively few papers which tackle dynamic kinetic problems. Alexandron et al. [3] present a dynamic and kinetic data structure for maintaining the convex hull of points (or analogously, the lower envelope of lines) moving in $\mathbb{R}^2$. Their data structure processes (in expectation) $O\left(n^2\beta_{s+2}(n)\log n\right)$ events in $O\left(\log^2 n\right)$ time each. Here, $\beta_s(n) = \lambda_s(n)/n$, and $\lambda_s(n)$ is the maximum length of a Davenport-Schinzel sequence on n symbols of order s. Agarwal et al. [1] present dynamic and kinetic data structures for maintaining the closest pair and all nearest neighbors. The expected number of events processed is again roughly $O\left(n^2\beta_{s+2}(n)\operatorname{polylog} n\right)$, each of which can be handled in $O\left(\operatorname{polylog} n\right)$ expected time. We are using some ideas and constructions which are similar in flavor to the structures presented in their paper.

Results. We present a fully dynamic data structure that can maintain a set P of disjoint growing squares. At any time, it supports inserting a new square that is disjoint from the squares in P, or removing an existing square from P. Our data structure will produce an *intersection event* at every time t when two squares $\square_p$ and $\square_q$, with $p, q \in P$, start to intersect (i.e. at any time before t, all squares in P remain disjoint). At such a time, we then have to delete some of the squares, to make sure that the squares in P are again disjoint. Our data structure can handle a sequence of $m \geq n$ updates in a total of $O\left(m\,\alpha(n)\log^5 n\right)$ time, each update is performed in $O\left(\log^5 n\right)$ amortized time.

Our Approach. In order to efficiently keep track of which squares in our set P are about to intersect, we combine the following two ideas.

The first is that we can focus on specific kinds of intersections. In particular, if we can, for a given square $\square_q$, track which square $\square_p$ will intersect the right side of $\square_q$ first, then we can analogously track similar intersection events for the left, top and bottom sides of $\square_q$. Moreover, we only need to explicitly consider two sides of $\square_q$, say the top and right sides: if, e.g., the left side of $\square_q$ intersects some square $\square_r$, then symmetrically the right side of $\square_r$ intersects $\square_q$ and thus we already track that event. We describe our data structure, built on two-layered range trees, for intersections with the right side of $\square_q$, and then combine two copies to make sure that all squares in P remain disjoint. We formalize this in Sect. 2.

The second idea is that, instead of having each square $\square_q$ independently track which $\square_p$ will intersect it first, we can intelligently group squares that track (roughly) the same subset of P. We do so efficiently using the layers of search trees. This saves on space used by our data structure, in the form of certificates. It also reduces the running time of handling a sequence of updates, simply because fewer certicates fail or need to be updated. We concretize this in Sect. 3.

While our data structure is conceptually simple, the exact implementation is somewhat intricate, and the details are numerous. Our initial analysis shows that our data structure maintains $O\left(\log^2 n\right)$ certificates for every secondary node in our range trees, so $O\left(n\log^3 n\right)$ in total, and supports dynamic updates in $O\left(\log^5 n\right)$ amortized time. This allows us to simulate the process of growing the squares in P –and thus solve the agglomerative glyph clustering problem– in $O\big(n\,\alpha(n)\log^5 n\big)$ time using $O\big(n\log^3 n\big)$ space. In Sect. 4 we analyze the relation between canonical subsets in dominance queries. We show that for two range trees T^R and T^B in $\mathbb{R}^d$, the number of pairs of nodes $r \in T^R$ and $b \in T^B$ for which r occurs in the canonical subset of a dominance query defined by b and vice versa is only $O\big(n(\log n \log\log n)^{d-1}\big)$, where n is the total size of T^R and T^B. This implies that the number of linking certificates that our data structure maintains, as well as the total space used, is actually only $O\big(n\log n\log\log n\big)$. Since the linking certificates provide an efficient representation of all dominance relations between two point sets (or within a point set), we believe that this result is of independent interest as well.

In the preliminary version of this paper, we partitioned space in a different way [5]. This gave us an $O\big(n\,\alpha(n)\log^7 n\big)$ time algorithm that used $O\big(n(\log n\log\log n)^2\big)$ space. The improvements we made, which we describe in this paper, lead to a reduction of two log factors in the running time, and a $\log n\log\log n$ factor of space usage.

2 Geometric Properties

Our approach, as described above, will be to focus on specific types of intersections. In particular, for any given square $\square_q$ we are interested in tracking intersections with the right side of $\square_q$. We observe that the first square $\square_p$ that is to intersect the right side of $\square_q$ will have (i) its left side touch $\square_q$ first, and (ii) its center p in a cone east of q. We refer to all squares having their center in this cone as $E(q)$. See Fig. 3 for an example. Let us introduce some notation and formally capture these two observations.

We are interested, for all points $q \in P$, in a cone east of them, which is the set of points $\{(x, y) \in \mathbb{R}^2 \mid x - x(q) \geq |y - y(q)|\}$. North, south and west cones can be defined analogously. Let ℓ_q denote the midpoint of the left edge of a square $\square_q$, and let r_q denote the midpoint of the right edge of $\square_q$. Similarly, let the midpoints of the top and bottom edges of $\square_q$ be denoted by $\uparrow_q$ and $\downarrow_q$, respectively. Furthermore, let $E(q)$ denote the subset of points of P lying in the cone east of q, and let $L(q) = \{\ell_p \mid p \in E(q)\}$ denote the set of midpoints of left edges of the squares of those points.

We remark that points $p \in P$ can simultaneously lie in, e.g., the east and south cones induced by any $q \in P$. This means that the top left corner of $\square_p$ will first intersect with the bottom right corner of $\square_q$. In such cases it is not necessary to track the square

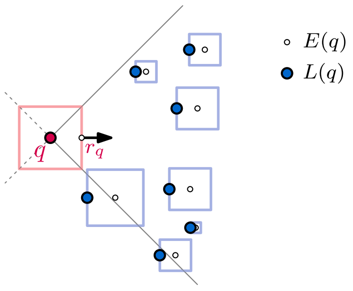

Fig. 3 Squares $E(q)$ east of $\square_q$. $\square_p \in E(q)$ intersects $\square_q$ when $x(r_q) \geq x(\ell_p)$

in both copies of our data structure. An arbitrary choice to break this degeneracy can be made. The same holds for the other pairs of neighboring cones.

Observation 1 *Let $p \in E(q)$ be a point east of point q. The squares $\square_q(t)$ and $\square_p(t)$ intersect at time t if and only if $x(r_q(t)) \geq x(\ell_p(t))$ at time t.*

Proof Clearly, when $x(r_q(t)) < x(\ell_p(t))$, then since the rightmost point of $\square_q$ is left of the leftmost point of $\square_p$, the squares do not intersect. For the other direction, observe that because p is in the cone east of q, it holds that $x(p)-x(q) \geq |y(p)-y(q)|$. Since $x(r_q(t)) \geq x(\ell_p(t))$, it holds that $x(r_q(t))-x(\ell_p(t))+x(p)-x(q) \geq x(p)-x(q) \geq |y(p)-y(q)|$. Because $\square_q$ and $\square_p$ are squares, either $y(\downarrow_q(t)) \leq y(\uparrow_p(t))$ or $y(\uparrow_q(t)) \geq y(\downarrow_p(t))$. In either case, both the horizontal and vertical intervals of $\square_q$ and $\square_p$ overlap, and hence they intersect. □

Following the above observations, we observe another important detail:

Observation 2 *Let t^* be the first time at which a square $\square_p$ of a point $p \in E(q)$ intersects $\square_q$. We then have that $x(r_q(t^*)) = x(\ell_p(t^*))$, and $\ell_p(t^*)$ is the point with minimum x-coordinate among the points in $L(q)$ at time t^*.*

Proof It is easy to see that $x(r_q(t^*)) = x(\ell_p(t^*))$. To see that $\ell_p(t^*)$ has minimum x-coordinate, we assume it is not for a contradiction. Let r be the point that does have minimum x-coordinate, thus $x(\ell_r(t^*)) < x(\ell_p(t^*))$. Since $x(r_q(t^*)) = x(\ell_p(t^*))$, we get $x(\ell_r(t^*)) < x(r_q(t^*))$. But r is east of q, so $x(r) > x(q)$ and thus, following a same type of argument as for Observation 1, $\square_q$ and $\square_r$ intersect. Because all squares grow linearly, there must be some $t' < t^*$ when $x(r_q(t')) = x(\ell_r(t'))$, and $\square_q$ first intersected $\square_r$. But then t^* is not the first time at which a square east of q intersects $\square_q$. Contradiction, thus $\ell_p(t^*)$ has minimum x-coordinate in $L(q)$ at time t^*. □

3 A Kinetic Data Structure for Growing Squares

In this section we present a data structure that can detect the first intersection among a dynamic set of disjoint growing squares. In particular, we describe a data structure that can detect intersections between all pairs of squares $\square_p$, $\square_q$ in P such that $p \in E(q)$. We then use two copies of this data structure, one for east/west and one for north/south intersections, to detect the first intersection among all pairs of squares.

We describe the data structure itself in Sect. 3.1, and we briefly describe how to query it in Sect. 3.2. We deal with updates –inserting a new square into P or deleting an existing square from P– in Sect. 3.3. In Sect. 3.4 we analyze the total number of events that we have to process, and the time required to do so, when we simulate growing the squares.

3.1 The Data Structure

Our data structure consists of two trees T^L and T^R, both with two layers, and a set of certificates linking nodes in T^L to nodes in T^R. These trees essentially form two 2D range trees on the centers of the squares in P, using rotated axes (see Sect. 2), so that we can easily query the tree for $E(q)$, for any $q \in P$. The second layer of T^L will double as a kinetic tournament tracking the left midpoints of the squares. Similarly, T^R will track the right midpoints of the squares.

The Layered Trees. The tree T^L is a 2D-range tree storing the center points in P: in the rotated plane suggested in Sect. 2, the first layer indexes the first coordinate x', while the second layer indexes the second coordinate y'. Each layer is implemented by a weight-balanced binary search tree (BB[α] tree) [10], and each node μ corresponds to a canonical subset P_μ of points stored in the leaves of the subtree rooted at μ. Let L_μ denote the set of left midpoints of squares corresponding to the set P_μ.

Consider the associated structure X_u^L of some primary node $u \in T^L$. We consider X_u^L as a kinetic tournament on the x-coordinates of the points L_u [1]. More specifically, every node $v \in X_u^L$ stores the midpoint ℓ_p in L_v with minimum x-coordinate, and will maintain certificates that guarantee this [1].

The tree T^R has the same structure as T^L, but the associated structure X_u^R, for some primary node u, –which again doubles as a kinetic tournament– maintains not the minimum, but the maximum x-coordinate of the points in R_u. Analogous to L_u, R_u is the set of right midpoints of the squares (with center points) in P_u. Hence, every secondary node $v \in X_u^R$ stores the midpoint r_q in R_v with maximum x-coordinate.

Linking the Trees. Kinetic data structures maintain certicates that guarantee that some property holds at least until some point in the (nearby) future. Maintaining a kinetic data structure revolves for a large part around updating certificates as they reach that point in time and break. We want to maintain the property that all squares in P are disjoint, and thus we next describe how to add *linking certificates* between the kinetic tournament nodes in the trees T^L and T^R that guarantee that the squares are disjoint. More specifically, a secondary node $v \in T^L$ represents a set $P_v \subseteq P$ of squares (the canonical subset of node v). We describe how we add certificates that guarantee that any $\square_q$, with $q \in P_v$, is disjoint from all $\square_p$, with $p \in E(q)$. These points q are represented by secondary nodes $w \in T^R$, and so we can discuss linking nodes instead of squares.

Consider a point q. There are $O\left(\log^2 n\right)$ nodes in the secondary tree of T^L whose canonical subsets together represent exactly $E(q)$. These nodes, in their secondary function as kinetic tournament nodes, represent the points in $L(q)$. So, in total q is *interested in* a set $Q^L(q)$ of $O\left(\log^2 n\right)$ kinetic tournament nodes. It now follows from Observation 2 that if we were to add certificates certifying that r_q is left of the point stored at the nodes in $Q^L(q)$, we can detect when $\square_q$ intersects with a square of a point in $E(q)$. However, as there may be many points q interested in a particular kinetic tournament node v (an example is given in Fig. 4), we cannot afford to maintain all of these certificates. The main idea is to represent all of these points q by a number of canonical subsets of nodes in T^R, and add certificates to only these nodes.

Consider a point p. Symmetric to the above construction, there are $O\left(\log^2 n\right)$ nodes in kinetic tournaments associated with T^R that together exactly represent the (right sides of) squares $\square_q$ west of p, for which $p \in E(q)$. Let $Q^R(p)$ denote this set.

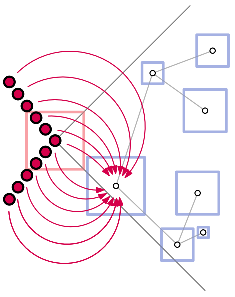

Fig. 4 The red points, on two convex chains, are all interested in the same set $Q^L(q)$. In general a linear number of points may be interested in a given kinetic tournament node

Next, we extend the definitions of Q^L and Q^R to kinetic tournament nodes. To this end, we first associate each kinetic tournament node with a (query) point in $\mathbb{R}^2$. Consider a kinetic tournament node v in a tournament X^L_u, associated with node u in the primary T^L. We denote the coordinates of a point p in the rotated coordinate system using $x'(p)$ and $y'(p)$. Let, in coordinates of the rotated system, $\underline{m}^v = (\min_{a\in P_u} x'(a), \min_{b\in P_v} y'(b))$ be the point associated with v (note that we take the minimum over different sets P_u and P_v for the different coordinates), and define $Q^R(v) = Q^R(\underline{m}^v)$. Symmetrically, for a node w in a tournament X^R_u, with $u \in T^R$, we define $\overline{m}^w = (\max_{a\in P_u} x'(a), \max_{b\in P_v} y'(b))$ and $Q^L(w) = Q^L(\overline{m}^w)$ (see Fig. 5).

We now add a linking certificate between every pair of secondary nodes $v \in T^L$ and $w \in T^R$ for which (i) v is a node in the canonical subset of w, that is $v \in Q^L(w)$, *and* (ii) w is a node in the canonical subset of v, $w \in Q^R(v)$. Such a certificate will guarantee that the midpoint r_q currently stored at w lies left of the point ℓ_p stored at v.

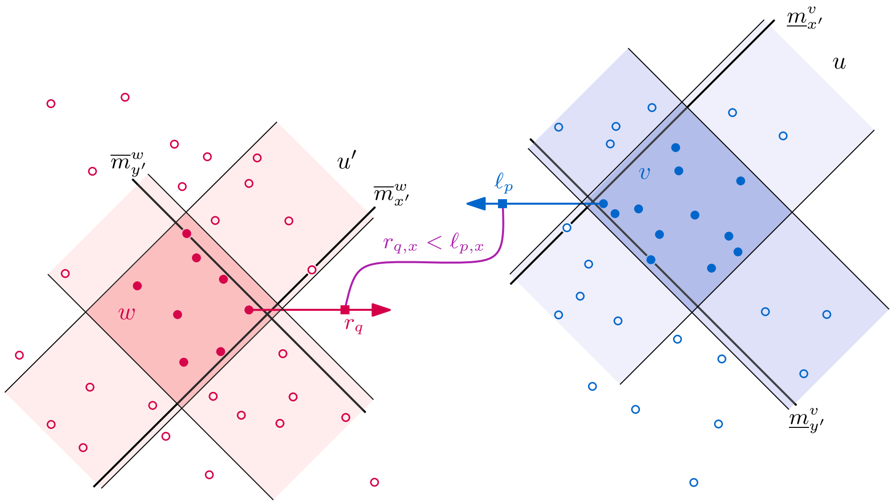

Fig. 5 The points $\overline{m}^w$ and $\underline{m}^v$ are defined by a pair of secondary nodes $w \in T^R_{u'}$, and $v \in T^L_u$. If $v \in Q^L(\overline{m}^w)$ and $w \in Q(\underline{m}^v)$, then we add a linking certificate between the rightmost midpoint r_q, $q \in P_w$, and the leftmost midpoint ℓ_p, $p \in P_v$, certifying that the squares in P_w are disjoint from those in P_v

Lemma 3 *Every kinetic tournament node is involved in $O\big(\log^2 n\big)$ linking certificates, and thus every point p is associated with at most $O\big(\log^4 n\big)$ certificates.*

Proof We start with the first part of the lemma statement. Every secondary node $v \in T^L$ can be associated with at most $O\big(\log^2 n\big)$ linking certificates: one with each node in $Q^R(v)$. Analogously, every secondary node $w \in T^R$ can be associated with at most $O\big(\log^2 n\big)$ linking certificates: one for each node in $Q^L(w)$.

Every point p occurs in the canonical subset of at most $O\big(\log^2 n\big)$ kinetic tournament nodes in the second layers of both T^L and T^R: p is stored in $O\big(\log n\big)$ leaves of the kinetic tournaments, and in each such a tournament it can participate in $O\big(\log n\big)$ certificates (at most two tournament certificates in $O\big(\log n\big)$ nodes). As we argued above, each such a node itself occurs in at most $O\big(\log^2 n\big)$ certificates. □

What remains to argue is that we can still detect the first upcoming intersection.

Lemma 4 *Consider two sets of elements, say blue elements B and red elements R, stored in the leaves of two binary search trees T^B and T^R, respectively, and let $p \in B$ and $q \in R$, with $q < p$, be leaves in trees T^B and T^R, respectively. There is a pair of nodes $b \in T^B$ and $r \in T^R$, such that (i) $p \in P_b$ and $b \in C(T^B, [x', \infty))$, and (ii) $q \in P_r$ and $r \in C(T^R, (-\infty, x])$, where $x' = \max P_r$, $x = \min P_b$, and $C(T^S, I)$ denotes the minimal set of nodes in T^S whose canonical subsets together represent exactly the elements of $S \cap I$.*

Proof Let b be the first node on the path from the root of T^B to p such that the canonical subset P_b of b is contained in the interval $[q, \infty)$, but the canonical subset of the parent of b is not. We define b to be the root of T^B if no such node exists. We define r to be the first node on the path from the root of T^R to q for which P_r is contained in $(-\infty, x]$ but the canonical subset of the parent is not. We again define r as the root of T^R if no such node exists. See Fig. 6. Clearly, we now directly have that r is one of the nodes whose canonical subsets form $R \cap (-\infty, x]$, and that $q \in P_r$ (as r lies on the search path to q). It is also easy to see that $p \in P_b$, as b lies on the

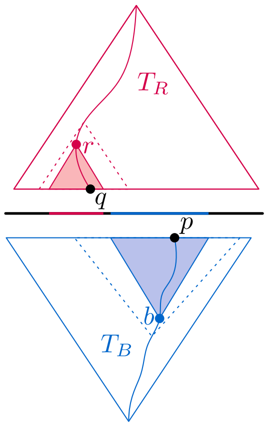

Fig. 6 The nodes b and r in the trees T^B and T^R

search path to p. All that remains is to show that b is one of the canonical subsets that together form $B \cap [x', \infty)$. This follows from the fact that $q \le x' < x \le p$ –and thus P_b is indeed a subset of $[x', \infty)$– and the fact that the subset of the parent v of b contains an element smaller than q, and can thus not be a subset of $[x', \infty)$. □

Lemma 5 *Let $\square_p$ and $\square_q$, with $p \in E(q)$, be the first pair of squares to intersect, at some time t^*. There is a pair of nodes v, w with a linking certificate that fails at time t^*.*

Proof Consider the leaves representing p and q in T^L and T^R, respectively. Applying Lemma 4, we get that there is a pair of nodes $u \in T^L$ and $u' \in T^R$ that, among other properties, have $p \in P_u$ and $q \in P_{u'}$. Hence, we can apply Lemma 4 again on the associated trees of u and u', giving us nodes $v \in X^L_u$ and $w \in X^R_{u'}$ with $p \in P_v$ and $q \in P_w$. In addition, these two applications of Lemma 4 give us two points (x, y) and (x', y') –with coordinates in the rotated system– such that:

- P_u occurs as a canonical subset representing $P \cap ([x', \infty) \times \mathbb{R})$, and
- P_v occurs as a canonical subset representing $P_u \cap (\mathbb{R} \times [y', \infty))$,

and such that

- $P_{u'}$ occurs as a canonical subset representing $P \cap ((-\infty, x] \times \mathbb{R})$, and
- P_w occurs as a canonical subset representing $P_{u'} \cap (\mathbb{R} \times (-\infty, y])$.

We combine these first two facts and observe that $\overline{m}^w = (x', y')$. This gives us that P_v occurs as a canonical subset representing $P \cap ([x', \infty) \times [y', \infty)) = E((x', y'))$, and hence $v \in Q^L(\overline{m}^w) = Q^L(w)$. Analogously, combining the latter two facts and $\underline{m}^v = (x, y)$ gives us $w \in Q^R(v)$. Therefore, v and w have a linking certificate. This linking certificate involves the leftmost left edge midpoint ℓ_a for some point $a \in P_v$ and the rightmost right edge midpoint r_b for some point $b \in P_w$. Since $p \in P_v$ and $q \in P_w$, we have that $r_q \le r_b$ and $\ell_a \le \ell_p$, and thus we detect their intersection at time t^*. □

From Lemma 5 it follows that we can now detect the first intersection between a pair of squares $\square_p$ and $\square_q$, with $p \in E(q)$, so any east/west intersection. We define an analogous data structure for detecting north/south intersections.

Space Usage. Our trees T^L and T^R are range trees in $\mathbb{R}^2$, and thus use $O(n \log n)$ space. However, it is easy to see that this is dominated by the space required to store the certificates. For all $O(n \log n)$ kinetic tournament nodes we store at most $O(\log^2 n)$ certificates (Lemma 3), and thus the total space used by our data structure is $O(n \log^3 n)$. In Sect. 4 we will show that the number of certificates we maintain, and thus the space our data structure uses, is actually only $O(n \log n \log \log n)$.

3.2 Answering Queries

The basic query that our data structure supports is testing if a query square $\square_q$ currently intersects with a square $\square_p$ in P, with $p \in E(q)$. To this end, we simply select the $O(\log^2 n)$ kinetic tournament nodes from T^L whose canonical subsets together represent $E(q)$. For each such a node v we check if the x-coordinate of the left midpoint ℓ_p stored at that node (which has minimum x-coordinate among L_v) is smaller than the x-coordinate of r_q. If so, the squares intersect. The correctness of our query algorithm directly follows from Observation 1. The total time required for a query is equivalent to the query time of a 2D range tree, which is $O(\log^2 n)$. We can similarly test if a given query point q is contained in a square $\square_p$, with $p \in E(q)$.

To check for intersections with squares p west of q, we can use T^R in a way symmetrical to the procedure described above. Our full data structure will contain trees analogous to T^L and T^R that can be used to check if there is a square $\square_p \in P$, with p either north or south of q, that intersects $\square_q$. Again, a similar query for containment of a query point q in a square $\square_p$ is also supported.

In summary, we can at any point in time test if a given query square $\square_q$ currently intersects any $\square_p$ in P, in $O(\log^2 n)$ time. We can in the same running time test if a query point q is currently contained in any square $\square_p$ in P.

3.3 Inserting or Deleting a Square

At an insertion or deletion of a square $\square_p$ we proceed in three steps. First, we update the individual trees T^L and T^R, making sure that they once again represent 2D range trees of all center points in P, and that the secondary data structures are, by themselves, correct kinetic tournaments. For each kinetic tournament node in T^L affected by the update, we then query T^R to find a new set of linking certificates. We update the affected kinetic tournament nodes in T^R analogously. Finally, we update the global event queue that stores all certificates.

Lemma 6 *Inserting a square into T^L or deleting a square from T^L takes $O(\log^2 n)$ amortized time.*

Proof We use the following standard procedure for updating the two-level BB[α] trees T^L in $O(\log^2 n)$ amortized time. An update (insertion or deletion) in a secondary data structure can easily be handled in $O(\log n)$ time. When we insert into or delete an element x in a BB[α] tree that has associated data structures, we add or remove the leaf that contains x, rebalance the tree by rotations, and finally add or remove x from the associated data structures. When we do a left rotation around an edge (μ, ν) we have to build a new associated data structure for node μ from scratch. See Fig. 7. Right rotations are handled analogously. It is well known that if building the associated data structure at node μ takes $O(|P_\mu| \log^c |P_\mu|)$ time, for some $c \geq 0$, then the costs of all rebalancing operations in a sequence of m insertions and deletions takes a total of $O(m \log^{c+1} n)$ time, where n is the maximum size of the tree at any time [9]. We can build a new kinetic tournament X_u^L for node u (using the associated data structures at its children) in linear time. Note that this cost excludes updating the global event queue.

It then follows that the cost of our rebalancing operations is at most $O(m \log n)$. This is dominated by the total number of nodes created and deleted, $O(m \log^2 n)$, during these operations. Hence, we can insert or delete a point (square) in T^L in $O(\log^2 n)$ amortized time. □

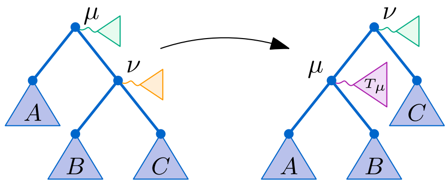

Fig. 7 After a left rotation around an edge (μ, ν), the associated data structure T_μ of node μ (pink) has to be rebuilt from scratch as its canonical subset has changed. For node ν we can simply use the old associated data of node μ. No other nodes are affected

Analogous to Lemma 6 we can update T^R in $O(\log^2 n)$ amortized time. Next, we update the linking certificates. We say that a kinetic tournament node v in T^L is *affected by* an update if (i) the update added or removed a leaf node in the subtree rooted at v, (ii) node v was involved in a tree rotation, or (iii) v occurs in a newly built associated tree X_u^L (for some node u). Let $\mathcal{X}_i^L$ denote the set of nodes affected by update i. Analogously, we define the set of nodes $\mathcal{X}_i^R$ of T^R affected by the update. For each node $v \in \mathcal{X}_i^L$, we query T^R to find the set of $O(\log^2 n)$ nodes whose canonical subsets represent $Q^R(v)$. For each node w in this set, we test if we have to add a linking certificate between v and w. As we show next, this takes constant time for each node w, and thus $O(\sum_i |\mathcal{X}_i^L| \log^2 n)$ time in total, for all nodes v. We update the linking certificates for all nodes in $\mathcal{X}_i^R$ analogously.

We have to add a link between a node $w \in Q^R(v)$ and v if and only if we also have $v \in Q^L(w)$. We test this as follows. Let u be the node whose associated tree X_u^L contains v. We have that, using the notation introduced in Lemma 4 and coordinates in the rotated system, $v \in Q^L(w)$ if and only if $u \in C(T^L, [\overline{m}_x^w, \infty))$, and $v \in C(X_u^L, [\overline{m}_y^w, \infty))$. We can test each of these conditions in constant time:

Observation 7 *Let q be a query point in $\mathbb{R}^1$, let v be a node in a binary search tree T, and let $x_p = \min P_p$ of the parent p of v in T, or $x_p = -\infty$ if no such node exists. We have that $v \in C(T, [q, \infty))$ if and only if $q \leq \min P_v$ and $q > x_p$.*

Proof We start with the if direction. Since $q > x_p$, it holds that $P_p \not\subset [q, \infty)$ and hence, $p \notin C(T, [q, \infty))$. However $q \leq \min P_v$ and so $P_v \subset [q, \infty)$. Clearly $v \in C(T, [q, \infty))$.

For the other direction, observe that $v \in C(T, [q, \infty))$ implies that $P_v \subset [q, \infty)$ and so, $q \leq \min P_v$. It also implies that $p \notin C(T, [q, \infty))$, hence, $P_p \setminus P_v \not\subset [q, \infty)$. It therefore must hold that $q > x_p$. □

Finally, we delete all certificates involving no longer existing nodes from our global event queue, and replace them by all newly created certificates. This takes $O(\log n)$ time per certificate. We charge the cost of deleting a certificate to when it gets created. Since every node v affected creates at most $O(\log^2 n)$ new certificates, all that remains is to bound the total number of affected nodes. We can show this using basically the same argument as we used to bound the update time. This leads to the following result.

Lemma 8 *Inserting a disjoint square into P, or deleting a square from P takes $O(\log^5 n)$ amortized time.*

Proof An update visits at most $O\left(\log^2 n\right)$ nodes itself (i.e. leaf nodes and nodes on the search path). All other affected nodes occur as newly built trees due to rebalancing operations. As in Lemma 6, the total number of nodes created due to rotations in a sequence of m updates is $O\left(m \log^2 n\right)$. It follows that the total number of affected nodes in such a sequence is $O\left(m \log^2 n\right)$. Therefore, we create $O\left(m \log^4 n\right)$ linking certificates in total, and we can compute them in $O\left(m \log^4 n\right)$ time. Updating the event global queue therefore takes $O\left(m \log^5 n\right)$ time. □

3.4 Running the Simulation

All that remains is to analyze the number of events processed. We show that in a sequence of m operations, our data structure processes at most $O\left(m\,\alpha(n) \log^2 n\right)$ events. This leads to the following result.

Theorem 9 *We can maintain a set P of n disjoint growing squares in a fully dynamic data structure such that we can detect the first time that a square $\square_q$ intersects with a square $\square_p$, with $p \in E(q)$. Our data structure uses $O\left(n \log n \log\log n\right)$ space, supports updates in $O\left(\log^5 n\right)$ amortized time, and queries in $O\left(\log^2 n\right)$ time. For a sequence of m operations, the structure processes a total of $O\left(m\,\alpha(n) \log^2 n\right)$ events in a total of $O\left(m\,\alpha(n) \log^5 n\right)$ time.*

Proof We argued the bounds on the query and the update times before. We argue the space bounds in Sect. 4. All that remains is to bound the number of events processed, and the time to do so.

We start by the observation that each failure of a linking certificate produces an intersection, and thus a subsequent update. It follows that, since we process m updates by definition, the number of such events is at most m.

To bound the number of events created by the tournament trees we extend the argument of Agarwal et al. [1]. For any kinetic tournament node v in T^L, the minimum x-coordinate corresponds to a lower envelope of line-segments in the t, x-space. This envelope has complexity $O\left(|P_v^*|\,\alpha(|P_v^*|)\right) = O\left(|P_v^*|\,\alpha(n)\right)$, where P_v^* is the multiset of points that ever occur in P_v, i.e. that are stored in a leaf of the subtree rooted at v at some time t. Hence, the number of tournament events involving node v is also at most $O\left(|P_v^*|\,\alpha(n)\right)$. It then follows that the total number of events is proportional to the size of these sets P_v^*, over all v in our tree. As in Lemma 6, every update directly contributes one point to $O\left(\log^2 n\right)$ nodes. The remaining contribution is due to rebalancing operations, and this cost is again bounded by $O\left(m \log^2 n\right)$. Thus, the total number of events processed is $O\left(m\,\alpha(n) \log^2 n\right)$.

At every event, we have to update the $O\left(\log^2 n\right)$ linking certificates of v. Because all events incur not only an ExtractMin operation (which takes constant time in some priority queue implementations), but also an insertion into or deletion from the priority queue, every event incurs a $O\left(\log n\right)$ time cost. We can thus update the linking certicates in $O\left(\log^3 n\right)$ time (including the time to update the global event queue). Thus, the total time for processing all kinetic tournament events in T^L is $O\left(m\,\alpha(n) \log^5 n\right)$. The analysis for the tournament nodes in T^R is analogous. □

To simulate the process of growing the squares in P, we now maintain two copies of the data structure from Theorem 9: one for east/west intersections and one for north/south intersections. We thus obtain the following result.

Theorem 10 *We can maintain a set P of n disjoint growing squares in a fully dynamic data structure such that we can detect the first time that two squares in P intersect. Our data structure uses $O\big(n \log n \log\log n\big)$ space, supports updates in $O\big(\log^5 n\big)$ amortized time, and queries in $O\big(\log^2 n\big)$ time. For a sequence of m operations, the structure processes $O\big(m\,\alpha(n)\log^2 n\big)$ events in a total of $O\big(m\,\alpha(n)\log^5 n\big)$ time.*

And so we can solve the agglomerative glyph clustering problem as follows.

Theorem 11 *Given a set of n initial square glyphs, we can compute an agglomerative clustering of the squares in $O\big(n\,\alpha(n)\log^5 n\big)$ time using $O\big(n \log n \log\log n\big)$ space.*

4 Efficient Representation of Dominance Relations

Our cones for each cardinal direction around a square $\square_q$ (defined in Sect. 2) closely resemble dominance relations: essentially they are a dominance relation in a rotated plane. The linking certificates of our data structure, which are built on these cones, actually comprise an efficient representation of all dominance relations between two point sets. We therefore think that this representation, and in particular the tighter analysis in this section, is of independent interest.

Let R and B be two point sets in $\mathbb{R}^d$ with $|R| = n$ and $|B| = m$, and let T^R and T^B be range trees built on R and B, respectively. We assume that each layer of T^R and T^B consists of a BB[α]-tree, although similar analyses can be performed for other types of balanced binary search trees. By definition, every node u on the lowest layer of T^R or T^B has an associated d-dimensional range Q_u (the hyper-box, not the subset of points). For a node $u \in T^R$, we consider the subset of points in B that dominate all points in Q_u, which can be comprised of $O\big(\log^d m\big)$ canonical subsets of B, represented by nodes in T^B. Similarly, for a node $v \in T^B$, we consider the subset of points in R that are dominated by all points in Q_v, which can be comprised of $O\big(\log^d n\big)$ canonical subsets of R, represented by nodes in T^R. We now link a node $u \in T^R$ and a node $v \in T^B$ if and only if v represents such a canonical subset for u and vice versa. By repeatedly applying Lemma 4 for each dimension, it can easily be shown that these links represent all dominance relations between R and B.

As a d-dimensional range tree consists of $O\big(n \log^{d-1} n\big)$ nodes, a trivial bound on the number of links is $O\big(m \log^{2d-1} n\big)$ (assuming $n \geq m$). Below we show that the number of links can be bounded by $O\big(n(\log n \log\log n)^{d-1}\big)$. We first consider the case for $d = 1$.

4.1 Analyzing the Number of Links in 1D

Let R and B be point sets in $\mathbb{R}$ with $|R| = n$, $|B| = m$, and $n \geq m$. Now, every associated range of a node u in T^R or T^B is an interval I_u. We can extend the interval to infinity in one direction; to the left for $u \in T^R$, and to the right for $u \in T^B$. For analysis purposes we construct another range tree T on $R \cup B$, where T is not a BB[α]-tree, but instead a perfectly balanced tree with height $\lceil \log(n+m) \rceil$. For convenience we assume that the associated intervals of T are slightly expanded so that all points in $R \cup B$ are always interior to the associated intervals. We associate a node u in T^R or T^B with a node v in T if the endpoint of I_u is contained in the associated interval I_v of v.

Observation 12 *Every node of T^R or T^B is associated with at most one node per level of T.*

For two intervals $I_u = (-\infty, a]$ and $I_v = [b, \infty)$, corresponding to a node $u \in T^R$ and a node $v \in T^B$, let $[a, b]$ be the *spanning interval* of u and v. We now want to charge spanning intervals of links to nodes of T. We charge a spanning interval $I_{uv} = [a, b]$ to a node w of T if and only if $[a, b]$ is a subset of I_w, and $[a, b]$ is cut by the splitting coordinate of w. Clearly, every spanning interval can be charged to exactly one node of T.

Now, for a node u of T, let $h_R(u)$ be the height of the highest node of T^R associated with u, and let $h_B(u)$ be the height of the highest node of T^B associated with u.

Lemma 13 *The number of spanning intervals charged to a node u of T is $O\big(h_R(u) \cdot h_B(u)\big)$.*

Proof Let x be the splitting coordinate of u and let $r \in T^R$ and $b \in T^B$ form a spanning interval that is charged to u. We claim that, using the notation introduced in Lemma 4, $r \in C(T^R, (-\infty, x])$ (and symmetrically, $b \in C(T^B, [x, \infty))$). Let $I_b = [x', \infty)$ be the associated interval of b, where $x' > x$. By definition, $r \in C(T^R, (-\infty, x'])$. If $r \notin C(T^R, (-\infty, x])$, then the right endpoint of I_r must lie between x and x'. But then the spanning interval of r and b would not be charged to u. As a result, we can only charge spanning intervals between $h_R(u)$ nodes of T^R and $h_B(u)$ nodes of T^B, of which there are at most $O\big(h_R(u) \cdot h_B(u)\big)$. $\square$

Using Lemma 13, we count the total number of charged spanning intervals and hence, links between T^R and T^B. We refer to this number as *numLinks* (T^R, T^B). This is simply $\sum_{u \in T} O\big(h_R(u) \cdot h_B(u)\big) \leq \sum_{u \in T} O\big(h_R(u)^2 + h_B(u)^2\big)$. We can split the sum and assume w.l.o.g. that *numLinks* $(T^R, T^B) \leq 2 \sum_{u \in T} O\big(h_R(u)^2\big)$. Rewriting the sum based on heights in T^R gives

$$\textit{numLinks}\,(T^R, T^B) \leq \sum_{h_R = 0}^{\textit{height}(T^R)} n_T(h_R) \cdot O\big(h_R^2\big),$$

where $n_T(h_R)$ is the number of nodes of T that have a node of height h_R associated with it.

To bound $n_T(h)$ we use Observation 12 and the fact that T^R is a BB[α] tree. Let $c = \frac{1}{1-\alpha}$, then we get that $\textit{height}(T^R) \leq \log_c(n)$ from properties of BB[α] trees. Therefore, the number of nodes in T^R that have height h is at most $O\big(\frac{n}{c^h}\big)$.

Lemma 14 $n_T(h) = O\big(\frac{(n+m)h}{c^h}\big)$.

Proof As argued, there are at most $O\big(n/c^h\big)$ nodes in T^R of height h. Consider cutting the tree T at level $\log(n/c^h)$. This results in a top tree of size $O\big(n/c^h\big)$, and $O\big(n/c^h\big)$ bottom trees. Clearly, the top tree contributes at most its size to $n_T(h)$. All bottom trees have height at most $\lceil\log(n+m)\rceil - \log(n/c^h) = O\big(\log(c^h)+\log(1+m/n)\big) = O\big(h+m/n\big)$. Every node in T^R of height h can, in the worst case, be associated with one distinct node per level in the bottom trees by Observation 12. Hence, the bottom trees contribute at most $O\big(n(h+m/n)/c^h\big) = O\big((nh+m)/c^h\big) = O\big((n+m)h/c^h\big)$ to $n_T(h)$. □

Using this bound on $n_T(h)$ in the sum we previously obtained gives:

$$numLinks\,(T^R, T^B) \leq \sum_{h_R=0}^{height(T^R)} O\left(\frac{(n+m)\,h_R^3}{c^{h_R}}\right) \leq O\big(n+m\big)\sum_{h=0}^{\infty}\frac{h^3}{c^h} = O\big(n+m\big).$$

Where indeed, $\sum_{h=0}^{\infty}\frac{h^3}{c^h} = O\big(1\big)$ because $c > 1$. Thus, we conclude:

Theorem 15 *The number of links between two 1-dimensional range trees T^R and T^B containing n and m points, respectively, is bounded by $O\big(n+m\big)$.*

4.2 Extending to Higher Dimensions

We now extend the bound to d dimensions. The idea is very simple. We first determine the links for the top-layer of the range trees. This results in links between associated range trees of $d-1$ dimensions (see Fig. 8). We then determine the links within the linked associated trees, which number can be bounded by induction on d.

Theorem 16 *The number of links between two d-dimensional range trees T^R and T^B containing respectively n and m $(n \geq m)$ points is bounded by $O\big(n(\log n \log\log n)^{d-1}\big)$.*

Proof We show by induction on d that the number of links is bounded by the minimum of $O\big(n(\log n \log\log n)^{d-1}\big)$ and $O\big(m\log^{2d-1} n\big)$. The second bound is simply the trivial bound given at the start of Sect. 4. The base case for $d = 1$ is provided by Theorem 15. Now consider the case for $d > 1$. We first determine the links for the top-layer of T^R and T^B. Now consider the links between an associated tree T_u in T^R containing k points and other associated trees $T_0, \ldots, T_r$ that contain at most k points.

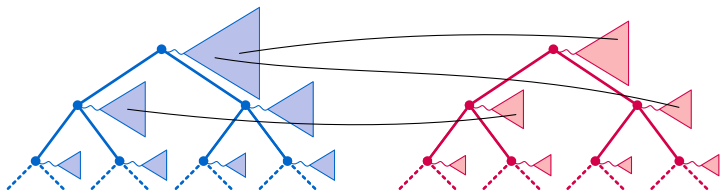

Fig. 8 Two layered trees with two layers, and the links between them (sketched in black). We are interested in bounding the number of such links

Since T_u can be linked with only one associated tree per level, and because both range trees use BB$[\alpha]$ trees, the number of points $m_0, \dots, m_r$ in $T_0, \dots, T_r$ satisfy $m_i \le k/c^i$ $(0 \le i \le r)$ where $c = \frac{1}{1-\alpha}$. By induction, the number of links between T_u and T_i is bounded by the minimum of $O\big(k(\log n \log\log n)^{d-2}\big)$ and $O\big(m_i \log^{2d-3} n\big)$. Now let $i^* = \log_c(\log^{d-1} n) = O\big(\log\log n\big)$. Then, for $i \ge i^*$, we get that $O\big(m_i \log^{2d-3} n\big) = O\big(k \log^{d-2} n\big)$. Since the sizes of the associated trees decrease geometrically, the total number of links between T_u and T_i for $i \ge i^*$ is bounded by $O\big(k \log^{d-2} n\big)$. The links with the remaining trees can be bounded by $O\big(k \log^{d-2} n (\log\log n)^{d-1}\big)$. Finally note that the top-layer of each range tree has $O\big(\log n\big)$ levels, and that each level contains n points in total. Thus, we obtain $O\big(n \log^{d-1} n (\log\log n)^{d-1}\big)$ links in total. The remaining links for which the associated tree in T^B is larger than in T^R can be bounded in the same way. □

It follows from Theorem 16 that our data structure from Sect. 3 actually maintains only $O\big(n \log n \log\log n\big)$ certificates. That number is significantly lower than our initial analysis showed: at the end of Sect. 3.1 we analyzed the space usage to be $O\big(n \log^3 n\big)$. The number of certicates, even under the tighter analysis, however still slightly dominates the space usage of the 2D range trees and kinetic tournaments, which is $O\big(n \log n\big)$. Our data structure thus uses only $O\big(n \log n \log\log n\big)$ space.

5 Conclusion and Future Work

We presented an efficient fully dynamic data structure for maintaining a set of disjoint growing squares. This leads to an efficient algorithm for agglomerative glyph clustering. The main future challenge is to improve the analysis of the running time. Our analysis from Sect. 4 shows that at any time, we need only few linking certificates. However, we would like to bound the total number of linking certificates used throughout the entire sequence of operations. An interesting question is if we can extend our argument to this case. This may also lead to a more efficient algorithm for maintaining the linking certificates during updates.

References

1. Agarwal, P.K., Kaplan, H., Sharir, M.: Kinetic and dynamic data structures for closest pair and all nearest neighbors. ACM Trans. Algorith **5**(1), 4:1–4:37 (2008)

2. Ahn, H.K., Bae, S.W., Choi, J., Korman, M., Mulzer, W., Oh, E., Park, J.W., van Renssen, A., Vigneron, A.: Faster algorithms for growing prioritized disks and rectangles. Comput. Geom. **80**, 23–39 (2019)
3. Alexandron, G., Kaplan, H., Sharir, M.: Kinetic and dynamic data structures for convex hulls and upper envelopes. Comput. Geom. **36**(2), 144–1158 (2007)
4. Bahrdt, D., Becher, M., Funke, S., Krumpe, F., Nusser, A., Seybold, M., Storandt, S.: Growing Balls in R^d. In: Proceedings of the 19th Workshop on Algorithm Engineering and Experiments (ALENEX), pp. 247–258. SIAM (2017)
5. Castermans, T., Speckmann, B., Staals, F., Verbeek, K.: Agglomerative clustering of growing squares. In: Latin American Symposium on Theoretical Informatics, pp. 260–274. Springer (2018)
6. Castermans, T.H.A., Speckmann, B., Verbeek, K.A.B., Westenberg, M.A., Betti, A., van den Berg, H.: GlamMap: Geovisualization for e-Humanities. In: Workshop on Visualization for the Digital Humanities (Vis4DH) (2016)
7. Funke, S., Krumpe, F., Storandt, S.: Crushing disks efficiently. In: International Workshop on Combinatorial Algorithms (IWOCA), pp. 43–54. Springer (2016)
8. Funke, S., Storandt, S.: Parametrized runtimes for ball tournaments. In: European Workshop on Computational Geometry (EuroCG), pp. 221–224 (2017)
9. Mehlhorn, K.: Data Structures and Algorithms 1: Sorting and Searching. Springer, Berlin (1984)
10. Nievergelt, J., Reingold, E.M.: Binary Search Trees of Bounded Balance. SIAM J. Comput. (SICOMP) **2**(1), 33–43 (1973)