A Practical Algorithm for Spatial Agglomerative Clustering

Thom Castermans†  Bettina Speckmann†  Kevin Verbeek†

Abstract
We study an agglomerative clustering problem motivated by visualizing disjoint glyphs (represented by geometric shapes) centered at specific locations on a geographic map. As we zoom out, the glyphs grow and start to overlap. We replace overlapping glyphs by one larger merged glyph to maintain disjointness. Our goal is to compute the resulting hierarchical clustering efficiently in practice.

A straightforward algorithm for such spatial agglomerative clustering runs in \(O(n^2 \log n)\) time, where \(n\) is the number of glyphs. This is not efficient enough for many real-world datasets which contain up to tens or hundreds of thousands of glyphs. Recently the theoretical upper bound was improved to \(O(n \alpha(n) \log^7 n)\) time [10], where \(\alpha(n)\) is the extremely slow growing inverse Ackermann function. Although this new algorithm is asymptotically much faster than the naïve algorithm, from a practical point of view, it does not perform better for \(n \leq 10^6\).

In this paper we present a new agglomerative clustering algorithm which works efficiently in practice. Our algorithm relies on the use of quadtrees to speed up spatial computations. Interestingly, even in non-pathological datasets we can encounter large glyphs that intersect many quadtree cells and that are involved in many clustering events. We therefore devise a special strategy to handle such large glyphs. We test our algorithm on several synthetic and real-world datasets and show that it performs well in practice.

∗The Netherlands Organization for Scientific Research (NWO) is supporting B. Speckmann under project no. 639.023.208, K. Verbeek under project no. 639.021.541, and T. Castermans under project no. 314.99.117.
†TU Eindhoven.
1 Introduction

We study an agglomerative clustering problem which is motivated by interactive geo-visualization applications, such as the two eHumanities tools GlamMap\(^1\) [7, 8] and GlottoVis\(^2\) [9, 20]. GlamMap, for example, shows the data of extremely large library metadata aggregations such as WorldCat\(^3\) or Trove\(^4\) on a geographic map (see Fig. 1). These aggregations contain extensive metadata for essentially every book in every library of the world (WorldCat alone contains more than 321 million library records). Such metadata of course includes author and title, but also relevant geographic data, such as the location of the library, the location of the publisher, or the location of the author (WorldCat lists hundreds of thousands of different locations). The last two fields are of particular importance to humanities scholars who work with books and authors from pre-internet times.

GlamMap represents each book by a square glyph centered at a location chosen by the user, for example, the location of the publisher. To increase visual clarity of the presentation and show concentration and clusters, GlamMap merges overlapping glyphs into larger glyphs until all overlap is removed. A merge replaces two existing glyphs by a new glyph which represents the joint information of the two merged glyphs. As the user zooms out the glyphs “grow” with respect to the underlying map and might hence overlap again, necessitating further merges. Eventually these merges result in a hierarchical clustering of the input glyphs.

GlamMap is an interactive tool which allows the user to search and filter. For example, the user can view the books by a particular set of authors or all books published during a particular time interval, on a particular topic, or in a particular language. Hence the agglomerative hierarchical clustering of the glyphs cannot be computed and stored ahead of time, but needs to be computed on the fly as a response to a user query. This is a crucial difference to other geo-visualization applications such as map labeling, where the data to be aggregated does not generally change. Since the user is waiting to see the answer to his or her query (which might contain tens or hundreds of thousands of locations) displayed on the map, the clustering algorithm needs to be very efficient so as not to frustrate the user.

GlottoVis follows the same general design as GlamMap, but it uses circular glyphs to encode language status and documentation for all languages of the world (see Fig. 2). In general one can envision a variety of related tools that use different glyph shapes as appropriate for the data the tool handles. However, any suitable glyph is of low complexity, as to not overload the visual space and leave room for other visual variables [23]. Furthermore, the rate at which glyphs grow as the user zooms out of the map (the so-called perceptual scaling) depends on the glyph type and the density of the data [14].

Problem description. Our input is a set of low complexity, disjoint weighted glyphs with geographic locations as well as a growing function (modeling zooming in the above applications). As the glyphs grow according to the growing function we merge glyphs as soon as they overlap. Our goal is to compute the resulting hierarchical clustering efficiently in practice.

Results and organization. In Section 2 we introduce some notation, define the problem and its variants in detail, and present a straightforward (slow) algorithm

---

\(^1\)http://glammap.net/glambdev/maps/5, Trove data set, best viewed in Chrome.
\(^2\)http://www.glammap.net/glottovis/
\(^3\)https://www.worldcat.org/
\(^4\)https://trove.nla.gov.au/

Figure 1: GlamMap displaying the Trove dataset.

Figure 2: GlottoVis displaying the Glottolog dataset.

Copyright © 2019 by SIAM
Unauthorized reproduction of this article is prohibited
to solve the problem. In Section 3 we present our new quadtree-based algorithm. We also argue that large glyphs may negatively affect the running time of our algorithm. Therefore, in Section 4 we present an augmentation to the algorithm that can efficiently handle large glyphs. In Section 5 we present the results of extensive experiments that show that our algorithm is efficient in practice for tens to hundreds of thousands of glyphs. We finish with some final thoughts and potential future work in Section 6.

Related work. Agglomerative clustering is one of the fundamental building blocks of data analysis and there are a multitude of methods and data structures devoted to it (see, for example, [21]). Most commonly, such clustering is performed on point data. In our setting, however, we are clustering low complexity glyphs which grow according to a (cartographic) scaling function. As such, our work is most closely related to map labeling for interactive maps.

In recent years, so called “ball tournaments”, introduced by Funke, Krumm, and Storandt [18], have been studied as an abstraction of such map labeling questions. Here the input is a set of balls in \( \mathbb{R}^d \) together with a set of priorities and a linear growing function. Whenever two growing balls touch, the ball with the lower priority is eliminated. Bahrdt et al. [6] and following up on their work Funke and Storandt [19] show how to compute an elimination sequence for \( n \) balls in \( O(n \log \Delta (\log n + \Delta^{d-1})) \) time in arbitrary dimensions and in \( O(Cn \text{polylog } n) \) time for \( d = 2 \), where \( C \) denotes the number of different radii and \( \Delta \) the ratio of the largest to the smallest radius. Quite recently Ahn et al. [3] showed how to compute elimination orders for ball tournaments in sub-quadratic time, for balls and boxes in two or higher dimensions. All of these results only handle deletions. As explained above, deletions are not sufficient on our setting, we also need to insert the new merged glyphs.

A straightforward algorithm for spatial agglomerative clustering in the fully dynamic setting with both insertions and deletions runs in \( O(n^2 \log n) \) time (see Section 2). Recently the theoretical upper bound was improved to \( O(n \alpha(n) \log^d n) \) time [10], where \( \alpha(n) \) is the extremely slow growing inverse Ackermann function. Although this new algorithm is asymptotically much faster than the naive algorithm, from a practical point of view, it does not perform better for \( n \leq 10^6 \).

The agglomerative clustering problem we consider in this paper combines both dynamic (insertion and deletion) and kinetic (growing functions) aspects. Our new practical algorithm relies on quadtrees to speed up spatial computations. There are comparatively few papers which handle both dynamic and kinetic aspects simultaneously. In particular, there are several results on kinetic kd-trees, quadtrees, and range spaces (see, e.g., [1, 13, 12]), and also on dynamic quadtrees (see [24] for a very recent example). We are not aware of dynamic and kinetic data structures for spatial partitioning problems. However, Alexandron et al. [4] describe a dynamic and kinetic data structure for maintaining the convex hull of points and Agarwal et al. [2] describe dynamic and kinetic data structures for maintaining the closest pair and all nearest neighbors.

2 Preliminaries

The algorithm we describe in Section 3 makes extensive use of a quadtree [16]. This data structure is among the first developed in the area of computational geometry and has been improved and researched in countless works. We briefly introduce it here for the sake of completeness.

A quadtree is a tree data structure \( T \) used to store points \( p_1, \ldots, p_n \) in \( \mathbb{R}^2 \) (in three dimensions the corresponding data structure is called an octree). Quadtree queries are typically used to efficiently compute proximity queries. Every node \( u \) in \( T \) represents a square cell \( \sigma(u) \) in \( \mathbb{R}^2 \). In particular, the root of \( T \) represents a square cell in \( \mathbb{R}^2 \) containing all the input points. Whenever a cell \( \sigma(u) \) contains more points than a given threshold \( \Lambda \), it is split into its four square quadrants, which correspond to the four children nodes of \( u \). Finally, the points are stored in the leaf nodes of \( T \), where each leaf can contain at most \( \Lambda \) points.

Quadtrees can efficiently report all points that lie in a query rectangle: starting at the root of \( T \) we check how the query rectangle overlaps with the cell of a node. If there is no overlap, then we report nothing. If the cell is contained in the query rectangle, then we report all points stored in the subtree of the node. Otherwise we recursively check all the children of the node. In a leaf node we can simply check all stored points individually.

From a theoretical point of view, quadtrees have nowadays been superseded by more advanced data structures, as quadtrees may perform poorly in the worst case. In particular, a quadtree may have unbounded depth when two points are very close together. This can be resolved by using compressed quadtrees [5], which ensure that the depth of the quadtree is bounded by \( O(n) \). A further improvement are skip quadtrees [15], which allow logarithmic queries. Although standard quadtrees may perform poorly in the worst case, they have shown to be efficient in practical applications [17, 22], and therefore are perfectly suitable for our specific application, even without any of the theoretical improvements.

In our algorithm we are not storing points, but we
are storing glyphs in the quadtree (see Fig. 3). This is also a common use of quadtrees. We store a glyph in every leaf node \( u \) for which \( \sigma(u) \) overlaps with the glyph. Every leaf cell can then be overlapped by at most \( \Lambda \) glyphs. We denote this set of glyphs for a leaf node \( u \in \mathcal{T} \) as \( G(u) \). Furthermore, we need to be able to trace through neighboring leaf cells. We define the set of neighbors \( N_{\text{dir}}(u) \) \( \{ \text{dir} \in \{N,E,S,W\} \} \) of a leaf node \( u \) as all nodes \( v \in \mathcal{T} \) for which \( \sigma(v) \) borders \( \sigma(u) \) in direction \( \text{dir} \).

**Detailed problem statement.** Our input consists of a set of \( n \) glyphs \( g_1, \ldots, g_n \). Each glyph \( g_i \) is centered at a point \( p_i \in \mathbb{R}^2 \) and has a weight \( w_i \in \mathbb{R}_{>0} \). These weights directly represent the data we want to visualize. We further define \( p_i \) as the growing rate of a glyph \( g_i \), which depends solely on the weight \( w_i \). We simulate growing the glyphs with a time parameter \( t \), where at \( t = 0 \) a glyph \( g_i \) covers only its center \( p_i \). The shape of a glyph is a closed ball in a particular normed space. More specifically, \( B_r(p) = \{ x \in \mathbb{R}^2 \mid \| x - p \| \leq r \} \) is the closed ball of radius \( r \) centered at \( p \), where \( \| \cdot \| \) is the used norm. We typically use either the Euclidean norm for circular glyphs or the Chebyshev (\( \ell_{\infty} \)) norm for square glyphs. The radius of a glyph \( g_i \) at time \( t \) is simply given by the function \( \rho_i(t) \). In this paper we sometimes use \( g_i(t) \) to refer to the glyph’s geometric representation at time \( t \).

For technical reasons we restrict the glyph shapes to use monotone norms\(^5\). A norm \( \| \cdot \| \) on a fixed coordinate system is monotone if for any two points \( p = (x, y) \) and \( p' = (x', y') \) with \( |x| \leq |x'| \) and \( |y| \leq |y'| \) we get that \( \| p \| \leq \| p' \| \). Note that all \( \ell_p \)-norms are monotone.

The goal is to compute the clustering that arises from growing the glyphs by increasing time \( t \) and merging two glyphs once they intersect. Now assume that \( g_i \) and \( g_j \) intersect at a time \( t_{ij} \). We then remove \( g_i \) and \( g_j \) and insert a new glyph \( g' \) with center \( p' = (w_ip_i + w_jp_j)/(w_i + w_j) \) and weight \( w' = w_i + w_j \) (see Fig. 4). This new glyph \( g' \) may directly intersect with one or multiple glyphs. We iteratively merge these overlapping glyphs with \( g' \), as these merges would have occurred before \( t_{ij} \), until there is no longer an overlap at the current time \( t_{ij} \). We repeat this process until all glyphs have been merged into a single glyph, at which point we have also computed a clustering tree.

**Variants.** We consider several variants of the problem, varying both the shape of the glyph and the computation of the growing rate. As already described above, the shape of a glyph can be any ball defined by some monotone norm \( \| \cdot \| \). In this paper we will only consider circular glyphs and square glyphs.

We further consider three growing functions: (linear) \( \rho_i = w_i \), (area) \( \rho_i = \sqrt{w_i} \), and (logarithmic) \( \rho_i = \log(w_i) \). Note that in the linear growing function the data is represented by the size (radius) of a glyph, and in the area growing function the data is represented by the area of the glyph. In the logarithmic growing function there is no such clear visual correspondence.

For glyphs that become very large compared to other glyphs, we can also utilize compression levels for glyphs. If the weight exceeds a particular threshold (of which there can be several), then we multiply the weight by some predefined constant before using it to compute \( \rho_i \). The compression level is visually indicated by drawing the glyph with a thicker border\(^6\). In this paper we only consider the linear and logarithmic growing functions without compression levels, and the area growing function with compression levels (as this matches the setup in the tool GlamMap).

**Naive algorithm.** A straightforward algorithm to solve our problem works as follows. Let \( t_{ij} \) denote the time at which glyph \( g_i \) and \( g_j \) start intersecting. We first compute \( t_{ij} \) for every pair of glyphs in the input and add the merge events to a priority queue \( Q \). While there are at least two glyphs, we extract the merge event with the lowest time value \( t_{ij} \) from \( Q \). If \( g_i \) or \( g_j \) is no

\(^5\)If the norm is not monotone, then only minor changes are needed to make our algorithm work.

\(^6\)Technically, we should incorporate this border in the glyph shape, and add a small constant to the radius of the glyph. We can easily do so in our algorithm, but for the sake of simplicity we ignore this minor technicality in this paper.

---

**Figure 3:** A quadtree containing square glyphs.

**Figure 4:** A merge between \( g_i \) and \( g_j \) results in \( g' \).
longer active (since it has been merged with another glyph already), we discard the event. Otherwise we make $g_i$ and $g_j$ inactive and add the new merged glyph $g'$. We also compute the merge events between $g'$ and all other active glyphs, and add these events to $Q$.

We can initialize $Q$ in $O(n^2)$ time, assuming that we can compute $t_{ij}$ for any pair of glyphs in $O(1)$ time. While handling a merge event we add at most $n$ events to $Q$, which can be done in $O(n \log n)$ time. Note that there can be at most $n - 1$ merge events. Furthermore, at most $O(n^2)$ events are added to $Q$ throughout the algorithm, so we can empty $Q$ in $O(n^2 \log n)$ time. Thus this naive algorithm can compute the complete clustering tree in $O(n^2 \log n)$ time.

Although the hidden constant in the $O$-notation for this simple algorithm is quite small, the algorithm does not scale well to $n \geq 10^4$. Nonetheless we will compare our algorithm with this naive algorithm in our experiments in Section 5.

3 Main Algorithm

The main problem with the naive algorithm is that it considers merge events between all pairs of glyphs. The main strategy is thus to compute merge events only for pairs that may potentially merge during the execution of the algorithm. Intuitively we should consider merge events between a glyph $g_i$ and another glyph $g_j$ for which $p_i$ is close to $p_j$ (perhaps one of its closest neighbors). However, the time of a merge event not only depends on the distance between two glyphs, but also on the growing rate of the two glyphs. Hence, two glyphs that are not that close to each other may actually be the first two glyphs to merge.

Therefore our approach is to consider distances between the glyph shapes instead of the centers, and to maintain close neighbors kinetically as the glyphs grow. Since at every merge event we delete two glyphs and insert a new one, our data structure needs to be both kinetic and dynamic. Indeed, the first subquadratic algorithm for this problem by Castermans et al. [10] also utilizes a kinetic and dynamic data structure. However, instead of the complex data structure in [10] that has good asymptotic worst-case behavior, we use a data structure that performs well in practice: a kinetic dynamic quadtree.

3.1 Kinetic dynamic quadtree In this section we describe a quadtree that contains disjoint growing glyphs (using time parameter $t$) and that supports the following operations:

- Find the first time that two glyphs will start to overlap (kinetic).
- Delete a glyph from the quadtree (dynamic).
- Insert a glyph into the quadtree (dynamic).

To make this data structure work correctly, we simulate growing the glyphs (increasing time $t$) and maintain for every cell in the quadtree which glyphs are overlapping it. During the simulation we split a quadtree node if it is overlapped by more than $\Lambda$ glyphs. On the other hand, if a parent node $u$ is overlapped by at most $\Lambda/2$ glyphs, then we delete the children of $u$ and make $u$ a leaf node. In our implementation we use $\Lambda = 10$.

We first describe the kinetic events we handle to maintain the above invariants and to compute the next time two glyphs start to overlap. Afterwards we describe how we can delete and insert glyphs into this quadtree. Finally we describe how to split or merge quadtree cells and update events accordingly. By keeping track of all processed merge events we directly obtain the clustering tree that is the solution to our problem.

**Kinetic events.** In our data structure we only deal with two types of events: *out-of-cell events* and *merge events*. Merge events occur when two glyphs start to overlap. Out-of-cell events occur when a glyph boundary leaves a quadtree cell, or more accurately, when the glyph starts intersecting the boundary of the cell. All events are stored in a single priority queue $Q$ and are handled in increasing order in time.

Initially every glyph $g_i$ consists of a single point and is stored in a single quadtree node $u$. We compute merge events between $g_i$ and all other glyphs in $G(u)$ and add those events to $Q$. Furthermore, we add 4 out-of-cell events to $Q$, one for each side of $\sigma(u)$.

Processing a merge event between two glyphs $g_i$ and $g_j$ is straightforward: we delete $g_i$ and $g_j$ and insert a new glyph $g'$ as is described in Section 2. The deletion and insertion operations are explained below.

Now assume we need to process an out-of-cell event of a glyph $g_i$ intersecting the top (northern) side of a quadtree cell $\sigma(u)$ (other directions are analogous). We then consider all neighboring leaf nodes in $N_N(u)$ and check which cells are intersected by $g_i$ at the time of the out-of-cell event. For each leaf node $v \in N_N(u)$ for which $\sigma(v)$ is intersected by $g_i$, we first add $g_i$ to $G(v)$. If $g_i$ was already in $G(v)$, we are done with $v$. Otherwise, we first check if $|G(v)| > \Lambda$, in which case we split $v$. Next we compute merge events between $g_i$ and all other glyphs in $G(v)$ and add those to $Q$. Finally we compute 3 out-of-cell events for $g_i$ and $v$, one for each direction except the direction from which $g_i$ entered $v$.

Finally note that, for every event, we first need to check if the event is still valid. An event can become invalid if an involved glyph has already been deleted.
Dynamic operations. Before we delete or insert a glyph \( g \), we first determine all quadtree leaf cells overlapped by \( g \). This can easily be computed recursively, starting from the root of \( T \): for a quadtree node \( u \), if \( \sigma(u) \) is disjoint from \( g \), then we return nothing. If \( \sigma(u) \) is contained in \( g \), then we report all leaf nodes that are descendants of \( u \). Otherwise, we recursively check all children of \( u \), or return \( u \) if it is a leaf. Let \( A \) be the resulting set of leaf nodes.

To delete a glyph \( g \), we simply remove \( g \) from \( G(u) \) for all nodes \( u \in A \). We then check the parent of \( u \) and merge it if necessary. Note that we can check this efficiently during the computation of the set \( A \).

Inserting a glyph \( g \) is more involved, as it may directly violate the disjointness of glyphs. We therefore first compute merge events between \( g \) and all glyphs in \( \bigcup_{u \in A} G(u) \), and determine the first event among these. If this event occurs before the current time, then we immediately remove \( g \) from the glyph set as well. We repeat this process until \( g \) is disjoint from all glyphs in the quadtree. Next we add \( g \) to \( G(u) \) for all \( u \in A \). Note that at that point we already computed all merge events with respect to glyphs in \( G(u) \), so we simply add all of those events to \( Q \). Next we determine if \( u \) must be split, and do so if necessary. Finally, we add out-of-cell events for \( g \) and every node \( u \in A \) for which \( g \) partially overlaps \( \sigma(u) \). All such out-of-cell events that occur before the current time can safely be discarded.

Quadtree operations. For both split and merge operations on a quadtree node \( u \), we need to take care to update the neighbor set of not only (the children of) \( u \) but also the neighbor set of the neighbors of (the children of) \( u \).

When we merge a node \( u \) with children \( v_1, \ldots, v_4 \), we first construct \( G(u) = \bigcup_{k=1}^4 G(v_k) \). We then compute merge events for every pair \( g_i, g_j \in G(u) \) that was not present in any \( G(v_k) \) for some \( k \), and add those events to \( Q \). Finally we add new out-of-cell events for all glyphs in \( G(u) \), as long as these events happen after the current time \( t \).

When we split a node \( u \) into children \( v_1, \ldots, v_4 \), we check for every glyph in \( G(u) \) which children it overlaps and add the glyph to the corresponding glyph sets. Note that we do not need to compute new merge events. However, we do need to compute new out-of-cell events for all children of \( u \) and add these events to \( Q \) if they occur after the current time \( t \).

Correctness. To argue the correctness of this algorithm, we simply need to show that we detect every merge event as it happens. This is clearly the case if we correctly maintain \( G(u) \) for every leaf node \( u \in T \), and if we compute new merge events whenever we add a glyph to \( G(u) \). The latter can easily be verified directly. For the former, it is straightforward to verify that the insert, delete, node split, and node merge operations correctly maintain \( G(u) \). We just need to prove that \( G(u) \) is correctly maintained kinetically, that is, the out-of-cell events work correctly.

Lemma 3.1. If a glyph \( g \) starts overlapping a cell \( \sigma(u) \) at some time \( t \), then there exists an out-of-cell event in \( Q \) at time \( t \) such that \( g \) will be added to \( G(u) \), provided that the norm defining the glyph shape is monotone.

Proof. We prove this statement by induction on \( \sigma(u) \) ordered by the time \( t \) at which \( g \) starts overlapping the cell. The base case consists of the cell that contains the center \( p \) of \( g \), which does not require an out-of-cell event. Now assume that every cell that \( g \) started overlapping before \( t \) has spawned out-of-cell events for its sides. Assume w.l.o.g. that the center of \( \sigma(u) \) is in the upper-right quadrant with respect to \( p \). Assume w.l.o.g. that the \( \sigma(v) \) (see Fig. 5 left). Furthermore, let \( \sigma(v) \) be the other leaf cell that contains \( q \) on its boundary. By the induction hypothesis, \( \sigma(v) \) has spawned an out-of-cell event for its top side. Since the norm is monotone, \( q \) minimizes \( ||q - p|| \) among all points on the horizontal line through the top side of \( \sigma(v) \). Thus, when processing this out-of-cell event, we must detect an intersection between \( g \) and \( \sigma(u) \).

We can use an analogous argument if the bottom side of \( \sigma(u) \) is below \( p \). If neither of these cases hold, then let \( q \) be the bottom-left corner of \( \sigma(u) \), which is the point that minimizes \( ||q - p|| \) as the norm is monotone (see Fig. 5 right).

Copyright © 2019 by SIAM
Unauthorized reproduction of this article is prohibited
points on the right side of $\sigma(v)$. Thus, when processing this out-of-cell event, we must detect an intersection between $g$ and $\sigma(u)$. Finally, in the special case that overlap with $\sigma(v)$ also occurs at time $t$, then the same out-of-cell event that adds $g$ to $G(v)$ will also add $g$ to $G(u)$.

3.2 Running time analysis

In this section we perform a rough analysis of the running time of our algorithm. It is not hard to see that, in the worst case (asymptotically), our algorithm may be as slow as the naive algorithm. However, we are interested in the performance of our algorithm on inputs that could arise in practice. The main goal of this section is to identify patterns in the input that could potentially slow down our algorithm. We can then come up with strategies to efficiently handle such patterns in the input. In this analysis we look at the complexity of the quadtree, the number of events in our kinetic data structure, and the number and complexity of the dynamic operations on our data structure.

Quadtree. In general a quadtree may have unbounded complexity, but this rarely happens in practice. By using a compressed quadtree instead, the complexity can be bounded by $O(n)$, where $n$ is the number of points in the quadtree. However, we are not simply storing points in the quadtree; we are storing glyphs of a particular shape. Nonetheless, since our glyphs are disjoint, have low complexity, and have low density [11], a compressed quadtree will still have linear complexity. Since we expect the difference between the complexity of a standard quadtree and a compressed quadtree to be low in practice, we expect our quadtree to have linear complexity as well.

Furthermore note that a quadtree node $u$ can be merged or split only when the set $G(u)$ changes. Thus we can charge such events to insertions, deletions, or out-of-cell events, which we analyze below.

Events. We now consider the number of kinetic events in our algorithm, starting with the out-of-cell events. Note that every executed out-of-cell event adds at least one new glyph to a leaf in the quadtree. As argued above, the total number of (leaf) cell-glyph incidences is expected to be linear. Thus, if we ignore glyph deletions, there can be at most a linear number of out-of-cell events. The additional number of out-of-cell events depends on the number of cells overlapped by deleted or inserted glyphs. We analyze this further below. Finally note that the number of discarded out-of-cell events is also bounded, since we add at most four out-of-cell events to the event queue every time we add a glyph to a leaf cell in the quadtree.

The number of executed merge events is easily bounded by $n-1$, as after that only one glyph remains. It is therefore more interesting to consider the number of discarded merge events. In every leaf cell of $T$ we can generate at most $O(A^2)$ merge events. Note that a merge event can only be invalidated when one of the two involved glyphs is deleted (that is, merged with another glyph). For every leaf cell overlapped by the deleted glyph, we can invalidate at most $A-1$ merge events. Thus, the number of discarded merge events again depends on the number of cells overlapped by deleted glyphs, which is analyzed further below.

Insertions and deletions. The insertions and deletions of glyphs are very controlled in our algorithm: they occur only during merge events. We need to handle at most $2n-2$ deletions and $n-1$ insertions. Furthermore, the deletions and insertions during a single merge event are typically geometrically close. The time complexity of a single insertion or deletion of a glyph depends on both the depth of the quadtree and the number of quadtree cells the glyph overlaps with. Whereas we expect the depth of the quadtree to be quite small in practice (at least on average), there can be large glyphs that overlap many cells. To ensure a good running time in practice, we need to make sure that we do not delete or insert too many large glyphs.

Large glyphs. Consider a glyph that overlaps many quadtree cells. This glyph must be much larger than the surrounding glyphs, and, since the glyphs must be disjoint, there must be many small glyphs for every such large glyph (see Fig. 6). In other words, there can only be few large glyphs at any point in time during the execution of the algorithm. One may incorrectly conclude that, on average, merge events will often occur between two small glyphs, and thus insertions and deletions will not be very costly on average. Unfortunately, due to the nature of our problem and the insertions and deletions it causes, merge events may often involve a large glyph. Indeed, there will not be many merge events between two large glyphs, as that eliminates one of the few large glyphs, but there can be many merge events between a large and a small glyph. Since large glyphs grow faster than small glyphs, they are more likely to be involved in a merge event. Furthermore, when a large glyph merges with a small glyph, the result is another large glyph. This problem especially occurs when the growing function causes larger glyphs to grow much faster than smaller graphs, like with the linear growing function.

We conclude that the main problem for the running time of our algorithm are merge events with large glyphs. This does not only result in costly deletion and
3.3 Further Improvement Next to the main problem with large glyphs, our algorithm also has the problem that it may generate many events, especially invalidated events that need to be discarded. Although we expect the number of events to be roughly linear, as discussed in Section 3.2, the hidden constant might be quite large. This may result in a very large priority queue \( Q \) and many relatively costly priority queue operations.

To avoid this problem, we limit the number of events we directly add to \( Q \). In particular, we add only one merge event per glyph to \( Q \). When creating merge events between a glyph \( g_i \) and glyphs in a cell \( G(u) \), we compute all events, but only insert the first one, say with \( g_j \), into \( Q \). The rest is stored in a small priority queue with \( g_i \). At the same time we store that \( g_i \) is tracking \( g_j \). When \( g_j \) merges with another glyph, then all glyphs tracking \( g_j \) can add another merge event to \( Q \). This approach keeps the number of (invalid) events in \( Q \) low, and we can more easily eliminate duplicate events in the small priority queue of a glyph.

We use a similar improvement for out-of-cell events: only the first one to occur for a particular glyph \( g_i \) is actually added to \( Q \). The others are again stored in a separate small queue with \( g_i \), and the next one is added to \( Q \) only when the first one has been handled.

4 Dealing with Large Glyphs

As described in Section 3.2, our algorithm may be slow if there are many merges with large glyphs. In this section we describe a strategy on how to deal with such large glyphs. Note that for large glyphs the quadtree is not very effective in reducing the number of computations involving this glyph. Our general strategy is therefore to extract the large glyph from the quadtree and handle it completely separately. Although we could then simply fall back on the naive algorithm, we can actually do better. Note that when a large glyph \( g \) merges with a much smaller glyph, then the center and growing rate of the merged glyph is very similar to that of \( g \). Although the times of all merge events with the large glyph will change, they will change only slightly, and it may not be necessary to recompute all merge events in order to find the next merge event for the large glyph. We will try to exploit this property of large glyphs to develop an efficient approach to handle them.

We first describe this approach for a single large glyph, before we explain how to deal with multiple large glyphs and how to find them. To keep it simple, we consider only the linear growing function \( \rho_i = w_i \). For other growing functions we can do something similar, but then this approach is less effective, and the problem of large glyphs is also less prevalent.

4.1 Single large glyph Let \( g_1 \) be the large glyph and let \( g_2, \ldots, g_n \) be the other (small) glyphs. For the large glyph we maintain a priority queue \( Q_1 \) with a special accumulator value \( \alpha_1 \), which will be explained later. Let \( t_i \) \((2 \leq i \leq n)\) be the time at which \( g_1 \) and \( g_i \) merge. We fill \( Q_1 \) with values \( \tau_i \) \((2 \leq i \leq n)\) where initially \( \tau_i = t_i \), and we set \( \alpha_1 = 1 \). The idea is that \( Q_1 \) contains lower bounds for the times of the merge events. In particular, we keep the invariant that \( t_i \geq \alpha_1 \tau_i \). Using \( Q_1 \) along with \( \alpha_1 \) we can compute the next merge event with \( g_1 \) and process it efficiently.

Computing the next event. Since \( Q_1 \) does not contain the actual merge times \( t_i \), but instead contains lower bounds to \( t_i \), we use the following procedure to compute the next merge event. Let \( \tau_i \) be the minimum value in \( Q_1 \). (If \( g_1 \) has been deleted, discard it.) We first compute \( t_i \) and check if \( t_i = \alpha_1 \tau_i \). If that is the case, then \( t_i = \alpha_1 \tau_i \leq \alpha_1 \tau_j \leq t_j \) for all \( j \neq i \). Thus, the next merge event will be with \( g_i \) and we are done. If \( t_i > \alpha_1 \tau_i \), then we compute a new value \( \tau_i' = t_i/\alpha_1 \) and add that to \( Q_1 \) again. Note that \( t_i \geq \alpha_1 \tau_i' \), so the invariant is maintained. We then pop the next value from \( Q_1 \) and continue the process until the next merge event has been found.

Processing a merge event. Assume that the next merge event is between \( g_1 \) and \( g_i \) at time \( t_i \). We will replace \( g_1 \) with a new large glyph \( g_1' \). Since \( g_i \) is much smaller than \( g_1 \) \((\rho_i \ll \rho_1)\), we expect that the position and size of \( g_1' \) is very similar to that of \( g_1 \). Nonetheless, the new merge times \( t_i' \) will be different from \( t_i \) and the lower bounds in \( Q_1 \) may no longer be correct. To fix the lower bounds efficiently, we simply change the value of \( \alpha_1 \). To correct the lower bounds we change \( \alpha_1 \) to \( \alpha_1' = \frac{\alpha_1 \rho_i}{\rho_1 + \rho_i} \alpha_1 \).
Lemma 4.1. If the merge event between \( g_1 \) and \( g_2 \) will take place at time \( t_j \), then the merge event between \( g'_1 \) and \( g'_2 \) will take place at time \( t'_j \), where \( g'_1 \) is the result of merging \( g_1 \) and \( g_i \), and \( p_1 \) and \( p_t \) are the growing rates of \( g_1 \) and \( g_i \), respectively.

**Proof.** Let \( p \) be the center of \( g_1 \) and let \( p' \) be the center of \( g'_1 \) (see Fig. 7). We first argue that \( \|p' - p\| = \rho_1 t_i \), where the norm corresponds to the shape of the glyphs. By definition of a merged glyph we get that \( p' = \frac{\rho p + \rho p_t \rho - \rho p_1 p_t}{\rho_1 + \rho p_t} \). This implies that \( p' - p = \frac{\rho_1 p + \rho p_t \rho - \rho p_1 p_t}{\rho_1 + \rho p_t} - \rho_1 t_i \). Since \( g_1 \) and \( g_i \) intersect at \( t_i \), we get that \( \|p_1 - p\| = (\rho_1 + \rho_1) t_i \). Thus we can conclude that \( \|p' - p\| = \rho_1 \|p_1 - p\| = \rho_1 t_i \).

Now consider another glyph \( g_j \). By the triangle inequality we get that \( \|p_j - p'\| = \|p_j - p\| - \|p' - p\| = \|p_j - p\| - \rho_1 t_i \). Furthermore we know that \( \|p_j - p\| = (\rho_1 + \rho_j) t_j \) and that \( t_j \geq t_i \). Thus we get that the new merge time is \( t'_j = \frac{\|p_j - p\|'}{\rho_1 + \rho_j + \rho p_t} \geq \frac{\|p_j - p\| - \|p' - p\|}{\rho_1 + \rho_j + \rho p_t} = \frac{(\rho_1 + \rho_j) t_i - \rho_1 t_i}{\rho_1 + \rho_j + \rho p_t} = (\rho_1 + \rho_j + \rho p_t) t_j \geq (\rho_1 + \rho_1 + \rho p_t) t_j = \alpha_1 t_j + \alpha_1 t_j = \beta'_1 t_j \), and hence the invariant is maintained by replacing \( \alpha_1 \) with \( \alpha'_1 \).

4.2 Multiple large glyphs. If we have multiple large glyphs, then we simply maintain several priority queues \( Q_k \) with associated values \( \alpha_k \), one per large glyph. Note that we expect to have only a small number of large glyphs. Whenever a small glyph merges with a large glyph, we simply process the event as described above for that particular large glyph (the priority queue of other large glyphs can remain unchanged). Merges between large glyphs are handled separately; since there should be only few large glyphs, we can simply apply the naïve algorithm to them. Then, whenever two large glyphs merge, we simply build a new priority queue \( Q_k \) for the new large merged glyph from scratch. If two small glyphs merge into a new glyph \( g'_1 \), then we must add merge events between \( g'_1 \) and each of the large glyphs to their respective priority queues \( Q_k \). Note that we need to divide the time of the merge event by \( \alpha_k \) to correctly add it to a priority queue \( Q_k \).

Finally we need to determine when a glyph should be considered large. For this we use a simple rule. Let \( c \) be a large constant which can be tuned to get the desired running time. We say that a glyph \( g_i \) is large if its growing rate \( \rho_i \) is at least \( c \) times the average growing rate of a glyph. We can check this initially for all glyphs, and after a merge event for the new merged glyph. To keep it simple, we decide that once a glyph becomes large, it will remain large. In our experiments we simply use \( c = 100 \), although one can obtain better results by choosing \( c \) carefully.

5 Experiments

In this section we present the results of our experiments on the performance of our algorithm for various synthetic and real-world datasets. Before we discuss the results, we first describe our experimental setup and the problem variants, algorithms, and datasets we use in our experiment. All the code, datasets, and the complete results of the experiments can be found on GitHub.7

Experimental setup. We implemented all algorithms in Java and use the ECJ compiler bundled with Eclipse 4.7.3a to execute the code on the OpenJDK 10.0.1 JVM. We use standard library data structures (ArrayList, PriorityQueue, ArrayDeque, ...). The quadtree implementation is custom. For our evaluation, we use a HP ZBook 15 laptop with four Intel Core i7-4700MQ CPUs at 2.4GHz and 7.7 GiB of RAM. Experiments were run with a heap size of 3 GiB (~3072M) for the NAIVE algorithm, and the system default of ± 2 GiB (~31978M) for other algorithms. The system runs Debian Buster/Sid on GNU/Linux kernel 4.16.0-2-amd64. For all experimental results, a timeout of five minutes is enforced.

Problem variants. As already mentioned in Section 2, we consider 6 different problem variants. For the glyph shapes we choose either circles (□) or squares (□) (using the Euclidean or Chebyshev norm). For the growing function we choose either linear (LIN), area (AREA) (with compression and borders), or logarithmic (LOG).

Algorithms. As a baseline, we also perform experiments using the naïve algorithm (NAIVE). Next to that, we use three different variants of our quadtree-based algorithm: the basic algorithm as described in Section 3.1 (QUAD), the basic algorithm including the improvement in Section 3.3 (QUAD+), and finally the complete algorithm with the improvement in

7https://github.com/Caster/growing-glyphs
Section 3.3 and the large glyph approach of Section 4 (QUAD+BIG). We test the large glyph approach only for the linear growing function.

Note that we do not compare to the algorithm by Bahrdt et al. [6], since their problem is different. Whereas they only delete glyphs, we also need to be able to insert new glyphs. We also do not compare our practical algorithm with the algorithm by Castermans et al. [10], which does handle insertions. There are two reasons for that: (1) this algorithm can only deal with square glyphs, and (2) the algorithm is designed in such a way that the best case running time is similar to the worst-case running time. Since the $O(n\alpha(n) \log^7 n)$ running time is simply infeasible for the number of glyphs we consider, we decided that there is no point in implementing this algorithm.

Datasets. In our experiments we run the algorithms on four types of datasets, where two are synthetic and two are real-world datasets. In particular we consider the following datasets:

**Uniform:** (synthetic) This dataset contains $n$ glyphs for which the centers are sampled uniformly from a square area, and where the weights are also sampled uniformly from the integer range $[1, 10]$.

**LargeGlyph:** (synthetic) This dataset contains one glyph at the origin with a very high weight, and $n-1$ glyphs for which the centers are sampled uniformly from an annulus (using norm corresponding to glyph shape) around the origin. The latter glyphs all have unit weight. This dataset is designed to test the approach for large glyphs.

**Trove:** (real-world) A collection of about 60 million bibliographic records published at nearly 8000 different locations. The center of a glyph corresponds to the location of a publisher, and the weight of a glyph corresponds to the number of records published at the corresponding location.

**Glottolog:** (real-world) A collection of bibliographic data on the world’s languages. The centers of the slightly over 7000 glyphs/languages correspond to representative geographic locations of the language. Most glyphs simply have unit weight, but some languages share a location.

Next to the different types of datasets, we also want to consider datasets of different size. Since both real-world datasets have about 8000 glyphs, this is the smallest size of datasets we consider. We also want to consider datasets with more glyphs, to see how our algorithm scales. More precisely, we also construct datasets with up to 200k glyphs. Although this is easy for the synthetic datasets, we cannot just make the real-world datasets larger.

To increase the number of glyphs of the real-world datasets we use the following approach. For every glyph $g$ with center $p$ and weight $w$ in the original dataset, we first compute the distance $d$ to its nearest neighbor. Then we replace $g$ by $k$ glyphs, where $k$ is chosen to result in the desired total number of glyphs. The centers of these $k$ glyphs are sampled uniformly from the disc around $p$ with radius $d$. We also sample the weights of the glyphs uniformly and rescale them to ensure they sum up to $w$.

The resulting datasets are actually synthetic datasets. However, we believe that this approach mostly preserves the density of the glyph centers and the distribution of the glyph weights. We therefore think that the resulting datasets are good representatives for real-world datasets, even though they are actually not.

5.1 Results We first investigate how the different problem variants influence the running time of our algorithm. For this we run the QUAD+ algorithm for each problem variant on each dataset with 100k glyphs, except for the special LargeGlyph dataset. The results are shown in Figure 8. From these results it is immediately apparent that our algorithm is fairly agnostic to the shape of the glyph. Also the algorithm seems slowest on the dataset that is the least uniform (Trove). Furthermore the algorithm is consistently slowest when using the linear growing function. This is as expected, since this growing function is most likely to trigger the large glyph problem.

Next we compare the different versions of our algorithm and also compare them to the naive algorithm. We restrict the problem variant to use square glyphs and the linear growing function, as this variant seems to be the most challenging. This time we restrict ourselves to the smallest dataset of each type. The results are shown in Figure 9. We quickly discovered that the naive algorithm cannot even run on datasets with 10k glyphs, since it runs out of memory. The naive algorithm was just able to run on the Trove and Glottolog datasets, but it appears to be about 50 times slower than QUAD+ already on these small datasets. Furthermore we can see that QUAD+ is a significant improvement over the basic QUAD algorithm. The special strategy for large glyphs also seems to provide a small speedup compared to QUAD+. Strangely though, no speedup is observed for the Uniform dataset for both QUAD+ and QUAD+BIG.

Finally we investigate the scalability of our algorithm. Again we restrict the problem variant to use square glyphs and the linear growing function. We first test our algorithms on the Glottolog dataset with increasing number of glyphs. The results are shown in Fig-
Figure 8: Running times (s) of the QUAD+ algorithm for each problem variant on datasets with 100k glyphs.

Figure 10: Running times (s) of the different algorithms for the LIN □ problem on the Glottolog dataset with increasing number of glyphs.

Figure 9: Running times (s) of the algorithms for the LIN □ problem on the smallest datasets of each type.

Figure 11: Running times (s) of the different algorithms for the LIN □ problem on the LargeGlyph dataset with increasing number of glyphs.

6 Conclusion

We presented a new quadtree-based algorithm for a variety of spatial agglomerative clustering problems. Although the worst-case behavior of the algorithm is as bad as that of a straightforward algorithm, we have shown via experiments that it performs well on real-world and synthetic datasets in practice, and it can compute the clustering in seconds for tens to hundreds of thousands of glyphs on a standard laptop, where the largest datasets can be computed in about half a minute.

In this paper we have mostly ignored multithreading. With modern processors often having several cores, it can be of interest to see how multithreading can be used to further improve the running time of the algorithm. We leave the design of an efficient multithreaded algorithm for the spatial agglomerative clustering problem as future work.

Copyright © 2019 by SIAM
Unauthorized reproduction of this article is prohibited
References


