GSViz: Progressive Visualization of Geospatial Influences in Social Networks

Sadeem Alsudais, Qiushi Bai, Shuang Zhao, Chen Li
Department of Computer Science, UC Irvine, CA 92697, USA
{salsudai,qbai1}@uci.edu,{shz,chenli}@ics.uci.edu

ABSTRACT

With the growing popularity of social networks, it becomes increasingly important to analyze binary relationships between entities such as users or online posts. These relationships are particularly useful when the entities are location-based. The spatial dimension provides more insights on influences in social networks across different regions. In this paper, we study how to visualize geospatial relationships on a large social network for queries with ad hoc conditions (such as keyword search) that retrieves a subnetwork. We focus on a main efficiency challenge to support responsive visualization, and present a middleware-based system called GSViz that progressively answers requests and computes results incrementally. GSViz minimizes visual clutter by clustering the spatial points while considering the edges among them. It further minimizes the clutter by incrementally bundling the edges, i.e., grouping similar edges in a bundle to increase the screen's white space. The system allows user interactions such as zooming and panning. We conducted an extensive computational study on real data sets and a user study, which evaluated the system's performance and the quality of its visualization results.

CCS CONCEPTS

• **Human-centered computing → Visualization**.

KEYWORDS

Geo-social network, progressive visualization

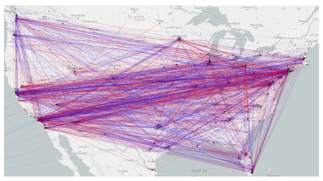
ACM Reference Format:
Sadeem Alsudais, Qiushi Bai, Shuang Zhao, Chen Li. 2022. GSViz: Progressive Visualization of Geospatial Influences in Social Networks. In *The 30th International Conference on Advances in Geographic Information Systems (SIGSPATIAL '22), November 1–4, 2022, Seattle, WA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/https://doi.org/10.1145/3557915.3560983

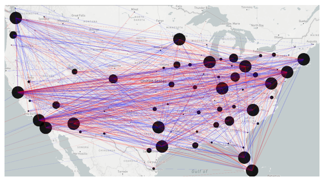
1 INTRODUCTION

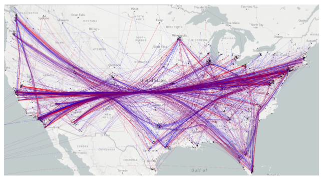
With the prevalent use of social media, it is becoming increasingly important to understand binary relations between entities, such as users or their online posts. Example relationships are "follows" between users and "retweets" between social-network posts. As many entities are location-based, naturally we want to analyze the geospatial relationships between these entities. In fact, analytics of geospatial relationships on social networks can be used in applications such as viral marketing (word-of-mouth information transfer between socially connected users) [40] or personalized location-based recommendations using social relationships [5]. Many recent efforts study the influence of geospatial relationships among entities on social networks [21, 22, 37].

Motivation. As an example, it has been observed that Covid-19 vaccine hesitancy is influenced by many factors such as geographic interactions [7] and social media interactions [26]. Analyzing tweets is a useful way to understand how vaccine information propagates across different regions. Figure 1a shows a sample network, where a node is a tweet about vaccines, and an edge between two tweets is an interaction between them, i.e., "retweet" or "reply-to." Analysts from public health are interested in examining the network to understand the geo-social influence of tweets about vaccines.

Visualization is a powerful and efficient tool to help analysts gain quick insights from data [3]. In this paper, we study how to visualize *geo-social networks*, i.e., geospatial relationships on a social network, to help domain experts that need this type of data analytics. We consider the common setting where the data is stored in a database system. As social media data has semantically rich attributes such as temporal, spatial, and textual attributes, we are particularly interested in the case where a user submits a visualization request with ad hoc conditions on the attributes. For instance, a user wants to visualize the subnetwork of tweets containing keywords such as `Covid`, `Pfizer`, or `Moderna`.


(a) All edges from replies (source blue) to original tweets (destination red)


(b) Result of edge-aware vertex clustering (§3)


(c) Result of progressive edge bundling (§4)

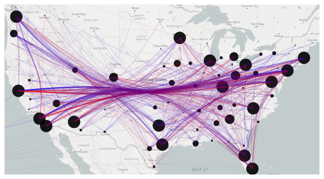

(d) Results of applying both techniques (§5)

Figure 1: A geo-social network of tweets containing the keyword `vaccine` and results of applying our techniques to support progressive visualization and simplify the network.

Challenges. Due to the ever increasing data size, visualizing large geo-social networks faces the following computational challenges:

C.1 Visualization requests with ad hoc conditions can be computationally expensive in terms of query execution in the database, network transfer, and frontend rendering. This negatively affects the responsiveness and consequently the user experience [14, 31].

C.2 Visualizing full networks without any simplification can produce results that are visually too cluttered. For instance, Figure 1a shows a cluttered result that is very difficult for the user to interpret.

To address these challenges, we develop a novel middleware-based system called "GSViz", which stands for "Geo-social network visualizer." To overcome **C1**, we leverage *progressive* computation by slicing a long-running query into multiple mini-queries, each of which has an additional slicing predicate on an attribute. In this way, each mini-query can be executed efficiently, and the results are returned in batches [20]. Similar to streaming videos, users are willing to wait until the end of a long running query as long as there are bursts of frequent updates "for keeping the user's attention focused" [14] and to help users "lose their sense of time" [31].

We address **C2** by *clustering* network vertices and *bundling* the edges. The problem of clustering spatial points has been studied extensively (e.g., [39]). These solutions cluster the points without considering the impact of edges between them. On the other hand, graph-based solutions [1, 35] do consider edges. However, they focus on optimizing the overall layout of a graph. In contrast to vertices in general graphs, vertices in geo-social networks have *geo-location*. Thus, solutions on graphs are not directly applicable in our case. We solve this problem by developing a new algorithm that clusters geo-social network vertices in an edge-aware fashion (Figure 1b). Additionally, we simplify a network using edge bundling that merges edges with similar directions and lengths (Figure 1c). Previous edge-bundling techniques [18] are not *incremental* and can take a long time to handle large input networks. To solve this problem, we develop a new technique to incrementally bundle the network edges that arrive in batches as the results of mini-queries. We show that these two techniques can be integrated to further simplify the network (Figure 1d).

To the best of our knowledge, GSViz is the first system to solve the problem of *incrementally* visualizing and *simplifying* a large geo-social network by querying a database. In this paper we make the following **contributions**:

(1) Introducing the architecture of GSViz and describing the details of its components needed to answer a visualization request with ad hoc conditions progressively (§2).
(2) Developing a new technique that clusters network vertices in an incremental and edge-aware fashion (§3).
(3) Presenting an efficient technique to bundle network edges progressively by leveraging a novel structure called PEB-tree (§4).
(4) Integrating the two techniques, addressing related challenges, and supporting zooming and panning (§5).
(5) Conducting an experimental evaluation, including a user study, on real data sets (§6). The results show that, compared to previous methods, the proposed system offers better performance without compromising the quality of visualization results.

1.1 Related Work

Big graph visualization systems. Some studies visualize the *geo-social network* as an **O-D** relation between spatial points [16]. The Gephi and Tulip systems [4, 6] load graph data into memory and process it offline, allowing interactive online filtering and exploration. These systems simplify the graphs and reduce visual clutter by performing graph clustering and layout algorithms. Tulip additionally bundles the edges. CGV [35], and ASK-GraphView [1] systems focus on graph(non-spatial) retrieval from a database and allow interactive exploration on the retrieved graph. They cluster the vertices while considering the edges by moving the vertices to any location to reduce edge crossings. CGV [35] additionally bundles the edges for a decluttered visual result. GraphVizdb [8] also allows visualizing a network based on querying the database. Tableau [32] is a commercial tool to visualize data from local files and remote databases. These techniques do not allow progressive processing of user queries.

Visualization of spatial points. One class includes visualization of spatial points such as VAS [25], Kyrix-s [33], Tabula [38], SOS [15], and HadoopViz [12] . The second class includes techniques to progressively cluster spatial points. IncrementalDBSCAN [13] clusters points by inserting each incoming point into a pre-existing nearby cluster or forming its own cluster if it is an outlier. BIRCH [39] uses a tree structure to group points into clusters based on Euclidean distance. GRIN [9] groups points in a hierarchical structure and uses the gravity theory to decide the position a new point should be inserted into the hierarchy. These methods do not consider edges between the points when clustering.

Edge bundling. Cost-based edge-bundling [18, 30] use a spatial metric to measure the closeness of the edges and move them closer. These methods produce results with a high visual quality but can be very slow when handling a large graph with many edges. Geometry-based edge bundling [11] uses geometric approaches such as Delaunay triangulation to decide which edges should be grouped. Image-based [19, 34] use Gaussian filtering to measure edge densities. Although these implementations could be applicable in our setting, they do not support incremental computation.

Progressive visualization. A section of prior works [20] focuses on progressive computation from the database perspective. Other solutions [36] focus on rendering the visual results progressively. Some sampling techniques [27] progressively improve the sample result by decreasing the error margin. These solutions are complementary to GSViz. Our focus is on how to incrementally cluster the vertices of a geo-social network while considering the edges between them and bundling the edges.

2 GSVIZ SYSTEM OVERVIEW

In this section, we explain the problem setting using an example, then describe the overall architecture of GSViz and explain the lifecycle of a visualization request in the system.

Problem formulation. Consider a typical three-tier architecture comprised of: (i) a frontend that submits visualization requests; (ii) a backend database that stores geo-social networks in tables; and (iii) a middleware layer that translates visualization requests

Table 1: Sample network data with tweets and their replies

from-id	from-date	from-coordinate	...	to-id	to-date	to-coordinate	...
667057004570	2020-08-19	(-74.0266, 40.6839)	...	669057256558	2020-08-17	(-73.9625, 40.5417)	...
669228452424	2020-08-11	(-122.4221,37.7700)	...	667131783385	2020-08-11	(-70.3463, 43.6405)	...
669057004984	2020-08-03	(-111.9217, 40.5933)	...	669225335465	2020-08-01	(-71.1915, 42.2277)	...

to database queries and forwards results to the frontend. To represent geo-social networks, we assume there is a table T, where every record is an edge connecting two geo-located points that may be associated with additional information such as text or timestamp. Table 1 shows such an example, where individual data points are tweets and a directed edge represents a reply-to relationship between an original tweet and a reply tweet.

The frontend layer allows a user to submit ad hoc visualization requests with arbitrary filtering conditions on spatial, textual, and temporal attributes. The following is an example query Q that requests for tweets and their replies posted during August 2020 and contain the keyword vaccine:

```
Original Query Q
    SELECT from-coordinate, to-coordinate
    FROM tweet-replies
    WHERE to_tsvector(from-text)@@to_tsquery('vaccine')
    AND from-date between '2020-08-01' and '2020-08-20'
    AND from-coordinate box '((-124.4, 36.5),(-70.1, 45.0))';
```

For a large table T, these visualization requests can be computationally expensive as handling them requires querying the database, transferring the results via the network, and performing frontend rendering. To allow timely feedback to the user, it is desired to have the middleware slice the original query Q into multiple *mini-queries*, each of which has an additional predicate on an attribute (which we call a *slicing predicate*). In the running example, we use "from-date" as the slicing attribute.

```
A mini-query Q_i
    SELECT from-coordinate, to-coordinate
    FROM tweet-replies
    WHERE to_tsvector(from-text)@@to_tsquery('vaccine')
    AND from-date between '2020-08-01' and '2020-08-05'
    AND from-coordinate box '((-124.4, 36.5),(-70.1, 45.0))';
```

The small date range in a mini-query makes it more selective, and an index on the slicing predicate helps fast retrieval of the result, a subset of the requested network called a *subnetwork*. Moreover, the small date range in the mini-query captures the dynamic changes on the network over time. In the running example, the mini-query Q_i includes an additional predict (in blue) to yield the subnetwork containing the keyword vaccine in the first 5 days in August 2020. The main objective of the middleware is to quickly process and visualize a subnetwork, in addition to those given by previous mini-queries, in a progressive fashion while minimizing visual clutter.

System architecture. We introduce a new system called GSViz that adopts the aforementioned three-tier architecture with a focus on the middleware layer for generating mini-queries then simplifying its results to visualize the network in a user-friendly fashion. Figure 2 depicts the system's architecture. Its QUERY MANAGER component answers long-running queries progressively. To reduce

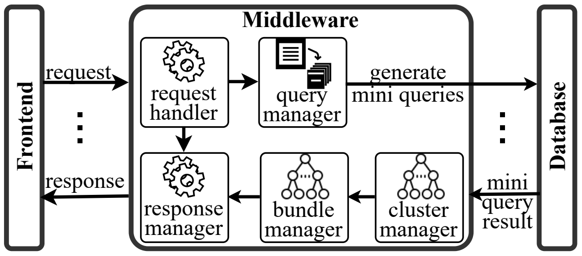


Figure 2: GSViz **system architecture.**

visual clutter, the middleware has a CLUSTER MANAGER and a BUNDLE MANAGER to incrementally cluster spatial points of a geo-social network and bundle edges, respectively.

The lifecycle of a visualization begins with the middleware slicing the original query into multiple mini-queries and forwards them to the backend database one by one. Every mini-query request and its response form a batch. Whenever the middleware receives the result of a mini-query from the database, it (i) Clusters the vertices of the subnetwork using an edge-aware spatial point clustering (§3), which produces *super edges* between the clusters; (ii) Bundles the super edges between the clusters in a progressive fashion; (iii) Forwards the simplified subnetwork to the frontend to render. We call an edge between two clusters a "super edge" because it represents many edges between individual points across the two clusters. The result of clustering the vertices and bundling the super edges of a subnetwork is called a *simplified* subnetwork.

3 INCREMENTAL EDGE-AWARE CLUSTERING OF GEO-SOCIAL NETWORK VERTICES

A key operation performed by GSViz is incremental spatial clustering of geo-social network vertices. This merge of nearby points and their associated edges reduces visual clutter of visualizing large networks. In this section, we first revisit a widely used point-clustering technique in §3.1. Building upon this technique, we introduce in §3.2 an edge-aware clustering algorithm. Lastly, we discuss in §3.3 performance optimizations of this algorithm.

3.1 Incremental Clustering of Network Vertices

We build the vertex clustering operation based on a widely used spatial point clustering algorithm called supercluster [17]. At a high level, supercluster takes as input a set of points, which map to vertices of network edges, and clusters the vertices iteratively. When a new edge arrives, the algorithms first performs a range search for each vertex of the edge over centers of existing clusters. The center of the cluster is the average of its points. The radius ρ of these searches is determined empirically based on several factors such as the field of view and screen resolution. Note that the range search can return multiple candidate clusters for each vertex. To determine which cluster a vertex should be inserted into,

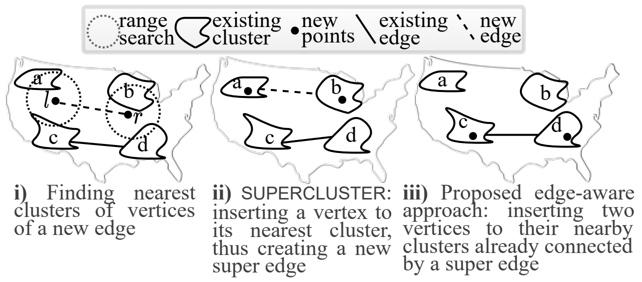

range search
existing cluster
new points
existing edge
new edge
a
b
c
d
l
r
i) Finding nearest clusters of vertices of a new edge
ii) SUPERCLUSTER: inserting a vertex to its nearest cluster, thus creating a new super edge
iii) Proposed edge-aware approach: inserting two vertices to their nearby clusters already connected by a super edge

Figure 3: Incrementally clustering a new edge (l, r). For simplicity, we omit the edge directions and cluster centers.

supercluster takes a greedy approach by selecting the cluster whose center lies closest to the vertex. Although this simple method works adequately for clustering spatial points, it neglects network edges, thus can lead to a large number of super edges after clustering.

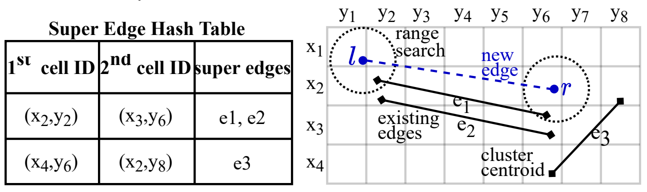
Figure 3 illustrates such an example. For simplicity, we represent an edge e as two vertices (l, r) to denote the left and right vertex respectively. Upon receiving a new edge (l, r), supercluster performs a range search around l and r, as shown in Figure 3-i. Then, points l and r are inserted in their nearest clusters, i.e., a and b respectively. This merge causes the resulting network to contain a super edge (a, b), as demonstrated in Figure 3-ii. In this example, since there already exists a super edge (c, d), inserting l into cluster c and r into d is better, as shown in Figure 3-iii. This example demonstrates that, to produce high-quality visualizations for geo-social networks, the point clustering algorithm needs to be *edge-aware*. That is, it needs to consider the information of existing super edges between the clusters during the clustering process. We present our solution to this problem next.

3.2 Achieving Edge-Awareness

For each edge $e = (l, r)$ in a batch of edges E, our goal is to insert both of its vertices into nearby clusters while minimizing the number of super edges. To this end, we start with checking if the distance between the two vertices is within ρ. If so, we merge them into one point m and insert it into its nearest cluster. The goal of this step is to filter the edges that are too short. If the distance is larger than ρ, we find nearby clusters for each vertex. For each vertex, if it does not have any nearby clusters, we create a new cluster for it and insert the new cluster to the corresponding set. We check if there exists a pair of candidate clusters that already has a super edge connecting them. If a pair exists, the edge vertices are inserted into these clusters. Revisiting the running example in Figure 3-iii, using this technique, point l will be inserted into cluster c and point r into d. If such a pair is not found, the two vertices are inserted into their nearest clusters. Then we create a super edge to connect the two clusters. Full details of the algorithm is in Appendix A

3.3 Improving Computational Efficiency

Computational challenges. The computational complexity of the edge-aware clustering algorithm is higher than that of traditional methods due to the need of examining connectivity between candidate clusters. Specifically, traditional point clustering algorithms such as supercluster always pick the nearest cluster, leading to a complexity of $O(dN)$, where d is the number of candidate clusters for a point n in a batch and N is the number of points in the batch, i.e., $N = 2E$, where E is the size of a subnetwork in the batch. The complexity of the edge-aware clustering is $O(d^2N)$, as we need to find the adjacency between every pair of candidate clusters from each corresponding vertex to check if they are connected. This high complexity negatively affects the visualization performance. To address the performance issue, we propose a grid-based technique to quickly find a nearby super edge within the range radius that is not necessarily the nearest.


Super Edge Hash Table

1st cell ID	2nd cell ID	super edges
(x2,y2)	(x3,y6)	e1, e2
(x4,y6)	(x2,y8)	e3

Figure 4: Using a grid to speed up edge-aware clustering.

Grid-based acceleration. We first divide the space into a grid, where the size of a cell is determined by ρ. Then we build an in-memory hash table to store the super edges. A key in the hash table consists of a pair of cell IDs and the value is a set of edges whose vertices are in the corresponding cells. Whenever a new super edge is formed between two clusters, we insert it to the hash table. Figure 4 shows how we use the hash table to insert an edge $e = (l, r)$ into an existing super edge. We first identify the grid cells within the search radius from l and from r. Consider the cell-ID pair $[\langle x_2, y_2\rangle, \langle x_3, y_6\rangle]$ in the hash table that includes a set of existing super edges in the cells. We choose any edge that both of its vertices lie within the search radius. i.e., edge e_1. We insert the points l and r to the corresponding clusters represented as the vertices of the edge e_1. The complexity of this approach is reduced to $O(E)$.

4 INCREMENTAL EDGE BUNDLING FOR NETWORK SIMPLIFICATION

In this section, we first describe the problem of network simplification using edge bundling (§4.1). Then, we present a new technique to enable efficient and incremental bundling of network edges (§4.2 and §4.3). For simplicity, we assume vertices of an edge in the network do not change in later batches and will relax this assumption in the next section.

4.1 Problem Specification

Given a geo-social network, we consider the problem of visually simplifying the network while preserving as much information as possible. This has been typically achieved via *edge bundling*—a process that deforms network edges so that nearby ones share similar shapes, allowing the screen space to be used more efficiently. We utilize the widely adopted *force-directed edge bundling* (FDEB) [18].

Force-directed edge bundling. We now provide a brief recap of the FDEB algorithm, starting with the following key definitions:

Definition 4.1. For two edges e_1 and e_2, their *compatibility measure*, denoted $C_e(e_1, e_2)$, is a value computed using their angle, length, position, and visibility [18].

Definition 4.2. We say two edges e_1 and e_2 are *compatible* if $C_e(e_1, e_2) \geq \delta$ for a given constant threshold δ.

Given an edge a and a set of edges $S = \{b_1, \ldots, b_n\}$ compatible with a, FDEB deforms the edge a based on S as follows. Assuming that the edge is represented as a polyline $a = (a_0, a_1, \ldots, a_m)$ with a_0 and a_m being the two fixed endpoints, the remaining points $a_1, \ldots, a_{m-1}$ are called the edge's *control points.* To deform a network edge a, FDEB applies two types of forces—*spring* and *electrostatic*—to its control points. The spring force exists between two adjacent points within the same edge. That is, each control point a_i (with $0 < i < m$) received spring forces $F_s(a_i, a_{i-1})$ and $F_s(a_i, a_{i+1})$ that are determined, respectively, by the positions of a_{i-1} and a_{i+1} [18]. The electrostatic force, on the other hand, is between control points from different compatible edges. Specifically, let $b_j = (b_{j,0}, \ldots, b_{j,m})$ be an edge from S. Then, for each $0 < i < m$, the electrostatic force acting on a_i by $b_{j,i}$ is

$$F_e(a_i, b_{j,i}) = \frac{\overrightarrow{a_i\, b_{j,i}}}{\|a_i - b_{j,i}\|}, \tag{1}$$

where $\overrightarrow{a_i\, b_{j,i}}$ denotes the unit vector pointing from a_i to $b_{j,i}$. To avoid singularities, F_e is set to zero when $\|a_i - b_{j,i}\|$ is below a predetermined threshold.

In this way, the net force acting on a_i (given S) is

$$F(a_i,\ S) = F_s(a_i, a_{i-1}) + F_s(a_i, a_{i+1}) + \sum_{b_j \in S} F_e(a_i, b_{j,i}). \tag{2}$$

All forces F, F_s, and F_e in the equation are two-dimensional vectors. By computing the net force acting on each control point a_i, we move these points accordingly, as shown in Figure 5. We call the entire process "bundling edge a using the set of edges S."

Given a set of edges $E = \{e_1, \ldots, e_n\}$, the FDEB algorithm bundles these edges as follows. For each $e_i \in E$, the algorithm first computes $S_i \subseteq E \setminus \{e_i\}$ comprised of edges compatible with e_i, and then uses S_i to bundle the edge e_i using the aforementioned process. Lastly, the bundling of each edge e_i, using the compatible ones, is repeated for a predetermined number of iterations.

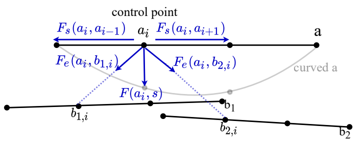


Figure 5: Dragging a control point on edge a using spring and electrostatic forces of compatible edges b_1 and b_2.

Computational challenges. To adopt FDEB in GSViz, a main challenge we need to overcome is its high computational cost. Specifically, to enable progressive visualization, we need to efficiently bundle edges of subnetworks that arrive in batches. A naïve solution is to bundle the new edges using existing subnetworks. This, however, is inefficient as computing the compatible set S_i for each new edge e_i in the batch requires examining all the edges received so far. Moreover, computing electrostatic forces using Eq. (1) requires examining all compatible edges from earlier batches. This clearly cannot meet the *responsiveness* requirement in interactive visualization. In the rest of this section, we present a novel technique for efficient incremental edge bundling. Notice that the spring forces are computed within each edge locally, so they can be computed with a low overhead. Thus we mainly focus on improving the performance of ❶ finding the compatible edges for each new edge and ❷ the computation of electrostatic forces.

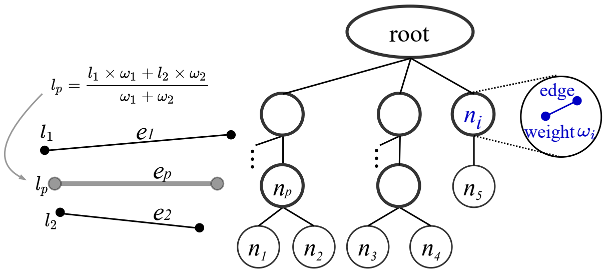


Figure 6: A PEB-tree for a set of edges.

4.2 PEB-Tree

At the core of our technique is a new hierarchical data structure—which we call the "PEB-tree" —that represents a set of edges E. ("PEB" stands for *Progressive Edge Bundling.*) As shown in Figure 6, each leaf of the PEB-tree corresponds to a "raw" edge from E. Each internal node stores: (i) an edge as a proxy of the raw edges in the descendants of this node; and (ii) a weight ω that is the number of those raw edges. Edges of the same parent siblings are initially compatible with each other, and the parent's edge is the weighted average edge computed using the child edges. Each vertex of the parent edge is a weighted average of the corresponding vertices of the child edges. Figure 6 shows the details of computing the vertex l_p of the weighted edge in the node n_p, where (x_p, y_p) is the weighted average of vertices l_1 and l_2 from the child nodes n_1 and n_2.

The tree has a pseudo root as the edges of its children are not required to be compatible with each other.

Tree construction for the first batch. We construct a PEB-tree for the first batch in progressive visualization as follows. Initially, we create a leaf node for each edge in this batch with a weight of 1. We group these edges such that all the edges in each group are compatible with each other. For each group, we construct a parent node that (i) stores an edge given by the weighted average of all edges stored in its children; and (ii) has a weight that equals the sum of the weights of its children. This process is repeated by grouping the nodes without a parent until all edges in those nodes are mutually incompatible. For each remaining leaf node that is incompatible with any other node, we construct a replica as its parent to represent a group that includes the edge of the leaf, so that future compatible edges could be merged into this group. Lastly, we construct a pseudo root as the parent of these remaining edges.

4.3 Incremental Edge Bundling Using PEB-tree

We now describe how to use the PEB-tree to efficiently and incrementally bundle a new batch of edges E. For every edge $e \in E$, we create a leaf node n with unit weight. We traverse the PEB-tree top-down and use the compatibility value between each tree node and e to guide the traversal. We find the deepest compatible non-leaf node n'. We use the same method as the one introduced in [18] to compute the compatibility value between a raw edge and a weighted edge. If there exists such an n' node, we add n as a child of n' and then traverse upward to the root to adjust the edges and their weights on the way. Notice that it is possible that the new edge e is not compatible with every child edge of node n'. In this case, after making n' the parent of n, the children of n' will no longer be mutually compatible. If we want to keep this all-pair-compatible property, we could partition the children into groups such that each group still has this property. A main downside of this approach is that there will be too many groups, and a new node can cause a cascading effect on the tree, which can be computationally expensive. We could relax this property for each group of children of the same parent node. On the other hand, if n' does not exist, we create a new parent node n' for the new leaf n, which is a replica of the new edge, then add n' as a new child under the root. The replica parent represents a group containing this singular edge to allow future compatible edges to be merged into. Note that the time complexity of traversing and maintaining the PEB-tree depends on its depth and branching factor, which can be controlled using heuristics for efficient traversal.

After inserting all the edges of the new batch E into the PEB-tree, we use the new tree to bundle these edges. For each edge $e \in E$, we use its corresponding parent node n' on the tree to bundle e using the two types of forces in the FDEB algorithm. In order to produce a similar edge bundling result of using all the child nodes under n' by using only their parent's weighted edge e', we redefine the electrostatic force on a control point a_i to include the weight information as follows:

$$F_e\left(e_i, n'(e'_i, \omega')\right) = \frac{\omega'}{\|e_i - e'_i\|} \overrightarrow{e_i e'_i}. \tag{3}$$

Deforming the new edge e using only the weighted average parent n' offers better performance compared with using all raw edges from the leaf nodes under n'. We note that the DEB algorithm [30] also considers edge weights in its revised electrostatic force function. However, their approach does not use a single edge to represent multiple edges, thus does not solve the efficiency issue. Lastly, the resulting curved edge, which contains the information about the new location of the control points, is sent to the frontend to be visualized while the edge in the tree remains the same as before the deformation. Full details of the algorithm is in Appendix 2.

5 INTEGRATING VERTEX CLUSTERING AND EDGE BUNDLING

So far we developed two progressive network-simplification techniques: one for clustering the vertices in a new batch, and one for bundling the new edges. In this section, we study how to integrate them in GSViz and address related challenges.

We integrate the two techniques in two steps. For the subnetwork in the first batch B_1, the middleware first uses the edge-aware clustering algorithm in §3.2 to group these vertices and generate a set of super edges between the clusters, where a super edge connects the centers of two clusters. It then uses the technique in §4.2 to bundle these super edges. Finally, it sends the results, including the clusters and the curved super edges, to the frontend to display. For each new batch B_i, the middleware repeats the aforementioned steps. For simplicity, we denote a super edge as e, and the curved super edge after bundling as $\hat{e}$ throughout this section.

One problem in integrating these two techniques for the batch B_i is the effect of the vertex clustering on those existing super edges computed on earlier batches $B_1, \ldots, B_{i-1}$.

5.1 Updating Edges Affected by Clustering

After adding a new vertex p in B_i to an existing cluster c, the center of c shifts as explained in details in Appendix B. As a result, the super edge e connected to this center also shifts as in Figure 7. The changes to the existing super edges should be reflected in both the PEB-tree and the displayed results on the frontend.

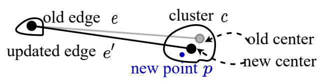


Figure 7: Adding a new point in an existing cluster causes the cluster and its related super edge to shift.

Updating PEB-tree. The changes to existing super edges in both cases include edge deletions and edge insertions. In §4.3 we already discussed how to insert edges to the tree. To delete an existing edge from the tree, we locate the leaf node that represents the old edge, and delete it from the tree. (We store a pointer for each edge to its leaf node.) Then, we propagate this deletion upwards and for each node on the path from the leaf to the root, we adjust its weight and edge. For instance, consider the example in Figure 8. Suppose e_1 is a super edge before batch B_i. After the vertex-clustering step for the new batch B_i, edge e_1 shifts to e'_1. We delete e_1 from the PEB-tree and insert the new edge e'_1 in the tree. If we were to directly update the edge e_1 to the new edge e'_1, then the new edge may not be compatible with its siblings. To address this concern, we first delete e_1, then insert e'_1 by using the compatibility score as discussed in §4 to traverse the PEB-tree. Thus e'_1 is still compatible with its new siblings. After we handle the updates of existing super edges, we start progressively inserting the newly generated super edges in the batch, e.g., e_2 in the running example.

Updating visualization results. As these updated super edges are already displayed to the user in the frontend, we also need to "hide" the outdated ones when changes occur. Consider the two approaches to rendering results in the frontend. Approach (i) that re-renders the new results from scratch is not appealing due to its low performance. For approach (ii) that renders new results as a new layer, we still need to identify the layers of those affected edges in order to delete these layers. To know which super edge belongs to which layer, the middleware stores for each super edge its batch number. We use the batch number to identify which layer the frontend has to replace. When updating those affected super edges,

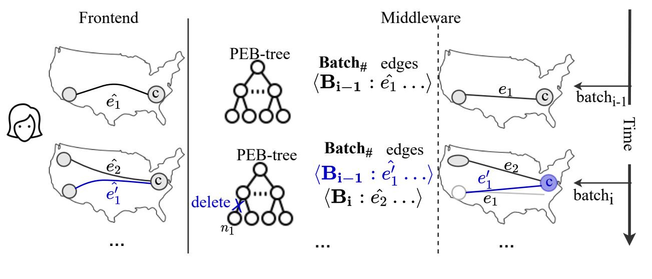


Figure 8: Maintaining updated edges affected by vertex clustering in batch B_i.

the middleware does not need to rebundle other edges. For those affected super edges, the middleware identifies their batches, then sends these batches and notifies the frontend to delete and redraw those corresponding layers. In the running example in Figure 8, after batch B_{i-1}, the hash map includes $\langle \mathbf{B_{i-1}} : \hat{e_1} \ldots \rangle$, since the super edge e_1 belongs to batch B_{i-1}. After the vertex-clustering step for batch B_i, the edge e_1 shifts. The middleware identifies its batch B_{i-1}, and notifies the frontend to delete and redraw the layer for this batch. To reduce the overhead of redrawing multiple layers because of frequent updates in super edges whenever a center of the cluster is shifted, we can optionally relax the updates on existing super edges to be performed only when a vertex of a super edge is located outside the boundary of its corresponding cluster.

5.2 Supporting Zooming and Panning

So far we discussed vertex clustering and edge bundling at one zoom level, where the result of both steps is a set of clusters and super edges between the clusters on a queried region. To support efficient zooming and panning operations, GSViz repeats the process of vertex clustering and edge bundling per batch at multiple zoom levels [17] in the background. GSViz maintains a PEB-tree at every level such that the leaves of the PEB-tree at each level represent the super edges between the clusters at that level. If a user wants to zoom or pan on the map, GSViz instantaneously retrieves the computed subnetwork for the particular region from the corresponding level.

6 EXPERIMENTS

In this section, we report an experimental evaluation of GSViz[1] using real datasets to answer the following questions. (1) How does edge-aware clustering perform (§6.2)? (2) How does the incremental edge bundling using PEB-tree perform (§6.3)? (3) How does GSViz perform when integrating the techniques and how does it compare to similar systems (§6.4)? (4) How is the quality of the final visual result perceived by users (§6.5)?.

6.1 Experiment Setting

We used three real geo-social network datasets as shown in Table 2. Gowalla [10] represents users' geotagged check-ins to places and their social friendship between early 2009 and late 2010. Foursquare [29] represents a social network between geo-tagged users collected from late 2011 till early 2012 in the US. We generated a random timestamp for every tuple and used it to specify a slicing predicate to query the data progressively. Twitter includes tweets and their replies collected from late 2015 until February 2021.

Table 2: Datasets.

Dataset	Content	Vertex #	Edge #	Size (GB)
Gowalla	Users' checkins and their social relation	99, 563	913, 660	0.12
Foursquare	Users' location and their social relation	28, 419	7, 176, 141	2.5
Twitter	Interaction between Twitter users by replies	33, 677, 670	20, 023, 731	8.6

[1]GSViz is available on Github (https://github.com/sadeemsaleh/gsviz)

We developed GSViz in Java. Additionally, to evaluate the developed algorithm of Progressive Edge-aware Clustering (PEAC) in §3.2, we implemented a greedy incremental version of Supercluster [17] called "Hierarchical Greedy Clustering" (HGC) in the middleware as "Baseline". Similarly, we implemented non-incremental FDEB [18] as the baseline to evaluate the Incremental Edge Bundling (PEB) in §4.2. We used two approaches for slicing a query into multiple mini-queries using the time predicate. The first one is called fixed-interval, which slices a query into equi-size time intervals. The other strategy is called DRUM [20], which slices a query into dynamic range intervals using a linear regression model to maintain the same running time from the database for each mini-query. Unless otherwise stated, the rhythm in DRUM was set to 500ms.

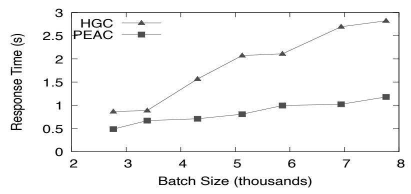
We ran the experiments on a 64-bit JVM on the Ubuntu 14 operating system on a machine with an Intel Xeon CPU, 98 GB of RAM, and a 2-TB hard disk. The data was stored in PostgreSQL 11.3 on the same machine. We built a B-tree index on the time attribute on which we specified the slicing predicate. Additionally, we built an inverted index on the text attribute on Twitter. We used keywords with different selectivity values to filter Twitter's "text". The Foursquare and Gowalla datasets did not have a text attribute, so we used the user-ID to specify range conditions. Each reported result is the average of three runs. We used a query that resulted in around $100K$ edges in total unless otherwise stated. To evaluate the quality of visualization on different zoom levels, we used levels ranging from 3 that showed north and central America to 7 that showed details of a US city. Experiments in (§6.4) and (§6.5) are only done on the largest dataset Twitter.

6.2 Progressive Vertex Clustering

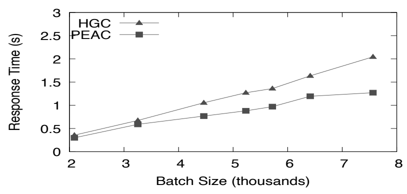
Effect of batch size on clustering performance. We evaluated the performance of PEAC against the baseline HGC. We measured the effect of varying the batch size on the performance of clustering vertices of a subnetwork in a new batch. We used the DRUM approach for slicing the query to keep a constant running time and similar batch sizes. We took the average of the batch size over all the batches from three runs. We varied the rhythm in DRUM between 1 second to 3 seconds.

Figure 9 shows the average response time of all batches for one batch size. Both HGC and PEAC had a sub-second response time when the batch had around $2K$ edges. As the batch size increased to $7K$ edges, HGC's response time increased to 2.8 seconds while PEAC's time was within 1.6 seconds. The reason PEAC's response time increased at a slower rate compared to HGC was due to the benefit of applying the grid-based technique discussed in §3.3 on PEAC to cluster the edges in the batch. Hence its performance was proportional to the batch size only, whereas HGC's performance was additionally affected by the neighboring clusters per vertex.

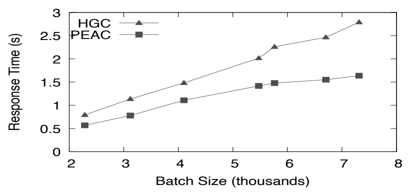
Effect of edge-aware clustering on graph density. We evaluated PEAC's reduction on the number of super edges compared to HGC. We used graph density [23], which is measured as the number of edges over the number of possible edges between the vertices in the graph. In our setting, we used the number of super edges that resulted from the clustering over the number of raw edges, i.e., $\frac{\#\text{ of super edges}}{\#\text{ of raw edges}}$. Figure 10 shows the results of the network's density for different zoom levels. For the Foursquare dataset, on zoom level



(a) Gowalla.



(b) Foursquare.



(c) Twitter.

Figure 9: Time of vertex clustering per batch (fixed batch size using DRUM**).**

4, HGC resulted in 2, 558 super edges, while PEAC significantly reduced the number to as low as 934. The graph density of PEAC was more noticeable at zoom levels 4, 5, and 6. At zoom level 3, the range radius ρ was large and it resulted in aggregating the network to include only a few clusters, which led to only a few super edges connecting them in both HGC and PEAC. Zoom level 7 had only a few clusters due to the small number of vertices in the small area of a city. As a result, both HGC and PEAC had few super edges.

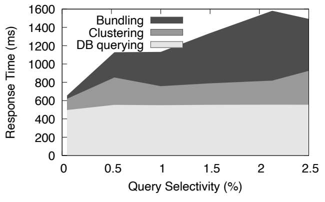
6.3 Progressive Edge Bundling

We evaluated the effect of varying the interval range size on the performance of non-incremental baseline FDEB and our incremental approach PEB. Since FDEB runs the bundling algorithm in each batch on the accumulated result of previous batches, we made sure the result of a batch is the same across different runs. In order to do that, we used the fixed-interval slicing approach. We used a query that generated around $3K$ edges in total.

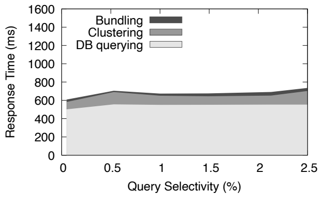
Figure 11 shows the average response time of edge bundling per batch using FDEB and PEB. When the range was two months in both Twitter and Foursquare, FDEB's response time was about 7.3 seconds, whereas PEB's response time was about 100 ms. When the slicing interval was 6 months, FDEB's response time increased to more than 10 seconds, while PEB's response time was only 1.6 seconds. FDEB's performance was better on Gowalla than its performance on Twitter and Foursquare datasets because most the edges were not compatible. We note that the response time of PEB on all datasets was mostly affected by the first batch when we used the baseline to construct the PEB-tree.

6.4 Integrating Both Techniques

We evaluated the performance of integrating both vertex clustering and edge bundling during the whole lifecycle of a visualization request in GSViz, including querying the database, clustering vertices, and bundling edges. We considered a baseline approach that used HGC for vertex clustering and FDEB for edge bundling. We then considered GSViz's approach that used PEAC for vertex clustering and PEB for edge bundling. We used the Twitter dataset and varied the keyword selectivity from 0.05% to 2.5%, resulting in a network that consisted of 10K to 500K raw edges. The number of super edges after clustering ranged from 1K to 3K. Figure 12 shows the average response time per batch for different keyword selectivity values. GSViz had a stable response time of 600 ms per batch regardless of the network size because the batch size was almost the same for each mini-query. The baseline's performance, on the other hand, increased from 655 ms to 1.6 seconds when the network size increased. This increase was because FDEB re-bundled the network edges from scratch for each batch.



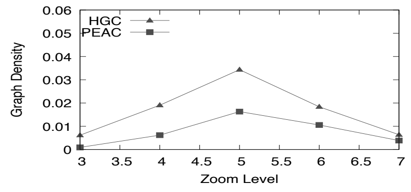
(a) Baseline.



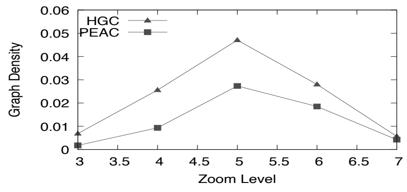
(b) GSViz.

Figure 12: Average response time per batch of all steps.

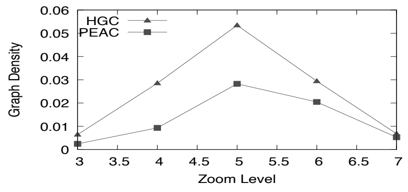
Total visualization time. To show the total time of visualizing the network across all the steps, we collected the total time it took



(a) Gowalla.

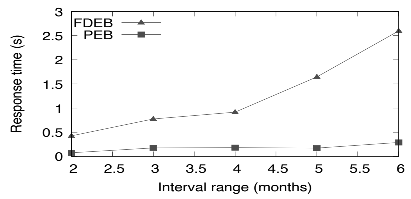


(b) Foursquare.

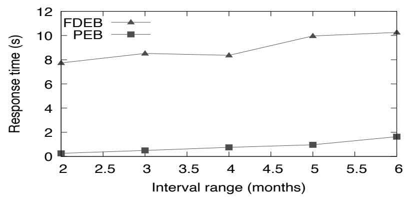


(c) Twitter.

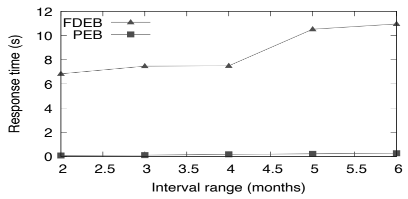
Figure 10: Graph density on different zoom levels (fixed batch size using DRUM**).**



(a) Gowalla.

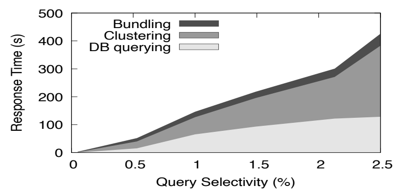


(b) Foursquare.

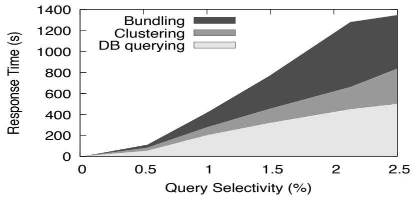


(c) Twitter.

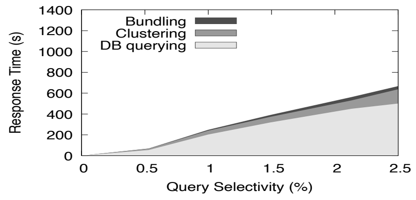
Figure 11: Bundling time per batch for different slicing intervals.



(a) non-progressive.



(b) Baseline.



(c) GSViz.

Figure 13: Total response time of all the steps.

to issue a single query to the database and process the data in a single batch. We call this method *non-progressive*. Figure 13a shows the performance for keyword conditions with different selectivity values. As the network size increased, the total time increased mostly to query the database and to cluster the vertices up to 7 minutes, where the user waits in the dark not knowing if the request was successful or not. Conversely, Figures 13b and 13c show the total response time of all the *batches* when the computation is progressive. As the network size increased, the total time increased due to the increase in the number of batches. GSViz's total time was usually half the time of the baseline to visualize the entire network. The baseline took about 23 minutes to query the database, cluster the vertices, and bundle the super edges on a network of 500K edges. GSViz took 10 minutes to show the same network.

Comparison with existing systems. We compared GSViz with two existing popular graph visualization systems, namely Tableau [32] (version 2021.2) and Tulip [2] (version 5.5.1). We chose Tableau due to its capability of doing middleware-based visualizations. We chose Tulip since it supports edge bundling. As these two systems could not be installed on Ubuntu 14 OS, we used a machine that supported all three systems. It had an Intel Core i5, 8GB RAM, and a 500GB hard disk, running MacOS 10.15.7 and PostgreSQL 12.5. Tableau and Tulip were not open-source, hence we used a stopwatch to measure the end-to-end performance of all three systems to visualize the network. We used database logging to measure the database query time. Tulip was an in-memory solution, and the largest network it could load had more than $900K$ nodes and $500K$ edges with a file size of $63MB$. It filtered the tweets using a keyword that resulted in a network of $10K$ edges, and this step took 11.55 seconds. It took additional 74.4 seconds to do edge bundling.

Tableau and GSViz support middleware-based visualization using a database, so we used the Twitter dataset. We filtered the network on a keyword condition that resulted in more than $200K$ edges. Tableau visualized the network on a map without any simplification. It took 19.49 seconds, including 15.81 seconds for querying the database. While its results were retrieved efficiently, the user had to wait for a long time before seeing any results. Moreover, the network clutter significantly hindered the user experience. On the contrary, GSViz retrieved the results progressively in 39 batches, each within 500 ms. The total time was 71.00 seconds, including 54.52 seconds for all the mini-queries, and 16.48 seconds for the steps of vertex clustering and edge bundling.

6.5 A User Study

We conducted a study to evaluate the user experience in GSViz. We mainly considered two methods: ❶ a baseline method using *non-incremental* HGC and FDEB to show its best visual quality, and ❷ GSViz using incremental PEAC and PEB. The goal of the user study is to answer the following question: "How do the two methods differ in terms of visualization quality?"

We invited 29 users, and each spent about 15 minutes to complete it. We generated 12 different sets from variations of 3 network sizes using different keywords at 4 zoom levels. The size of the network ranged from 10K to 100K raw edges. The zoom level ranged from an overview of North and Central America to a level of a few states in the US. Each set had 3 different methods, resulting in a total of 36 images. We first showed the result of the original network as is. We then showed the visual result yielded from the baseline and GSViz presented anonymously to the participants. To make the comparison fair for the baseline, we fixed the number of clusters and asked questions independently.

We used two metrics to measure the visualization quality:

(1) *Readability* [24], which indicates how easy it is to read the visualized network. To measure the readability, we asked the participants to subjectively answer a question for each simplified network: *"Q: Rate how cluttered you think the network is."* . The answer is a rating score of 1 (very cluttered) up to 5 (very sparse).
(2) *Task faithfulness* [24], which indicates how accurate the visualization of the simplified network is to correctly perform tasks. To measure the faithfulness, we asked the participants to answer analytical multiple-choice questions, each of which had 4 choices with only one correct answer. A score of 0 means the network is unfaithful and a score of 1 indicates a very faithful network [24]. All of the questions had the following template: *"Q: Which of the points, highlighted with green boxes, has more original tweets compared to reply-to tweets?"*

Figure 14 shows sample images given to the users including labels to indicate the randomly chosen clusters in the questions.

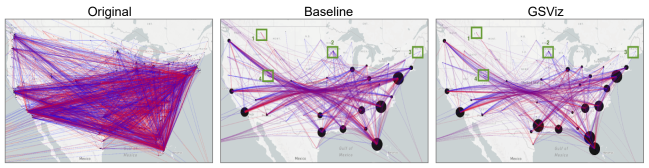


Figure 14: Example network visualizations in the user study at one zoom level.

The results are shown in Table 3. The average response time using Baseline was 4,473ms while GSViz was 631ms showing much higher performance. As the network size increased, the time difference also increased. This increase was more noticeable in the baseline. The average readability rating of Baseline was 2.3, which means the network was perceived as cluttered. GSViz's average readability rating was 2.9, which indicates that the network was perceived as not cluttered nor sparse. The average faithfulness

Table 3: User study results. The reported numbers for the visualization quality are represented as "A|B," where "A" is the average score given by all the participants and "B" is the standard deviation. Compared to the baseline, GSViz **was much more efficient and had comparable visualization quality.**

Network Size	Time per batch (ms)		Visualization Quality			
			Readability		Faithfulness	
	baseline	GSViz	baseline	GSViz	baseline	GSViz
10K	648.70	525.20	2.59|0.11	3.28|0.26	0.82|0.11	0.82|0.09
50K	2,603.54	635.42	2.15|0.17	2.85|0.13	0.53|0.25	0.63|0.33
100K	10,165.61	733.67	2.09|0.23	2.48|0.21	0.75|0.32	0.79|0.36

score of Baseline was 0.70, it means that the network was faithful. GSViz had a better average faithfulness score of 0.74. The user study showed that, compared to the Baseline, GSViz had much higher performance and produced visualization with comparable quality.

7 CONCLUSION

In this paper, we presented GSViz, a system to enable progressive visualization of geo-social networks. We first demonstrated how to improve incremental spatial clustering to make it edge-aware to reduce visual clutter. We also studied supporting incremental edge bundling by storing previous edges as nodes in a novel tree index called PEB-tree to optimize the traversal and processing of bundling edges. Moreover, we discussed the integration of the two techniques and solved new challenges. Lastly, we conducted an extensive evaluation of GSViz compared to baseline algorithms for spatial clustering and edge bundling. The experiments also included a user study to evaluate the quality of the produced visualization. The results showed that the techniques can not only support efficient, responsive visualization of networks progressively but also produce high-quality simplified network visualization.

Future Work. GSViz stores the clustering hierarchy and the PEB-tree in memory and follows a heuristic to reduce the tree size by merging nodes when a tree-size threshold is met. We plan to devise a cost-based technique to reduce the tree size given a memory budget. Another improvement is to propose an objective to trade-off the visualization accuracy and performance efficiency. Currently, GSViz follows a heuristic and greedy approach to clustering the vertices and bundling the edges efficiently. The algorithm is bounded by the range radius ρ for clustering and by the compatibility score in edge bundling, so the quality and accuracy is not compromised.

REFERENCES

[1] James Abello, Frank van Ham, and Neeraj Krishnan. 2006. ASK-GraphView: A Large Scale Graph Visualization System. *IEEE Trans. Vis. Comput. Graph.* 12, 5 (2006).

[2] Reda Alhajj and Jon G. Rokne (Eds.). 2018. *Encyclopedia of Social Network Analysis and Mining, 2nd Edition.* Springer.

[3] Syed Mohd Ali, Noopur Gupta, Gopal Krishna Nayak, and Rakesh Kumar Lenka. 2016. Big data visualization: Tools and challenges. In *(IC3I).*

[4] David Auber. 2004. Tulip - A Huge Graph Visualization Framework. In *Graph Drawing Software.* 105–126.

[5] Jie Bao, Yu Zheng, David Wilkie, and Mohamed F. Mokbel. 2015. Recommendations in location-based social networks: a survey. *GeoInformatica* 19, 3 (2015).

[6] Mathieu Bastian, Sebastien Heymann, and Mathieu Jacomy. 2009. Gephi: An Open Source Software for Exploring and Manipulating Networks. In *ICWSM.*

[7] Trinidad Beleche, Joel Ruhter, Allison Kolbe, Jessica Marus, Laina Bush, and BD Sommers. 2021. COVID-19 Vaccine Hesitancy: Demographic Factors, Geographic Patterns, and Changes Over Time. *Published online* 27 (2021).

[8] Nikos Bikakis, John Liagouris, Maria Krommyda, George Papastefanatos, and Timos K. Sellis. 2016. graphVizdb: A scalable platform for interactive large graph visualization. In *ICDE.*

[9] Chien-Yu Chen, Shien-Ching Hwang, and Yen-Jen Oyang. 2002. An Incremental Hierarchical Data Clustering Algorithm Based on Gravity Theory. In *PAKDD*, Vol. 2336.

[10] Eunjoon Cho, Seth A. Myers, and Jure Leskovec. 2011. Friendship and mobility: user movement in location-based social networks. In *SIGKDD.*

[11] Weiwei Cui, Hong Zhou, Huamin Qu, Pak Chung Wong, and Xiaoming Li. 2008. Geometry-Based Edge Clustering for Graph Visualization. *IEEE Trans. Vis. Comput. Graph.* 14, 6 (2008).

[12] Ahmed Eldawy, Mohamed F. Mokbel, and Christopher Jonathan. 2016. HadoopViz: A MapReduce framework for extensible visualization of big spatial data. In *ICDE.*

[13] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Michael Wimmer, and Xiaowei Xu. 1998. Incremental Clustering for Mining in a Data Warehousing Environment. In *VLDB.*

[14] M Fiedler et al. 2004. State-of-the-art with regards to user-perceived Quality of Service and quality feedback. In *Euro-NGI Deliverable D. WP. JRA. 6. 1. 1.*

[15] Tao Guo, Kaiyu Feng, Gao Cong, and Zhifeng Bao. 2018. Efficient Selection of Geospatial Data on Maps for Interactive and Visualized Exploration. In *SIGMOD.*

[16] Su Yeon Han, Keith C. Clarke, and Ming-Hsiang Tsou. 2017. Animated Flow Maps for Visualizing Human Movement: Two Demonstrations with Air Traffic and Twitter Data. In *SIGSPATIAL Workshop on Analytics for Local Events and News.*

[17] Hierarchical Clustering. Hierarchical Clustering. https://github.com/mapbox/supercluster

[18] Danny Holten and Jarke J. van Wijk. 2009. Force-Directed Edge Bundling for Graph Visualization. *Comput. Graph. Forum* 28, 3 (2009).

[19] Christophe Hurter, Ozan Ersoy, and Alexandru Telea. 2013. Smooth bundling of large streaming and sequence graphs. In *PacificVis.*

[20] Jianfeng Jia, Chen Li, and Michael J. Carey. 2017. Drum: A rhythmic approach to interactive analytics on large data. In *BigData.*

[21] Guoliang Li, Shuo Chen, Jianhua Feng, Kian-Lee Tan, and Wen-Syan Li. 2014. Efficient location-aware influence maximization. In *SIGMOD.*

[22] Jianxin Li, Timos Sellis, J. Shane Culpepper, Zhenying He, Chengfei Liu, and Junhu Wang. 2018. Geo-Social Influence Spanning Maximization. In *ICDE.*

[23] Jaroslav Nesetril and Patrice Ossona de Mendez. 2012. *Sparsity - Graphs, Structures, and Algorithms.* Algorithms and combinatorics, Vol. 28. Springer.

[24] Quan Hoang Nguyen, Peter Eades, and Seok-Hee Hong. 2017. Towards Faithful Graph Visualizations. *CoRR* abs/1701.00921 (2017). arXiv:1701.00921

[25] Yongjoo Park, Michael J. Cafarella, and Barzan Mozafari. 2016. Visualization-aware sampling for very large databases. In *ICDE.*

[26] Viju Raghupathi, Jie Ren, and Wullianallur Raghupathi. 2020. Studying public perception about vaccination: A sentiment analysis of tweets. *International journal of environmental research and public health* 17, 10 (2020).

[27] Sajjadur Rahman, Maryam Aliakbarpour, Hidy Kong, Eric Blais, Karrie Karahalios, Aditya G. Parameswaran, and Ronitt Rubinfeld. 2017. I've Seen "Enough": Incrementally Improving Visualizations to Support Rapid Decision Making. *VLDB* (2017).

[28] Ruth Rosenholtz, Yin Li, Zhenlan Jin, and Jonathan Mansfield. 2010. Feature congestion: A measure of visual clutter. *Journal of Vision - J VISION* 6 (06 2010).

[29] UMN Sarwat Foursquare Dataset (September 2013). http://www-users.cs.umn.edu/~sarwat/foursquaredata/

[30] David Selassie, Brandon Heller, and Jeffrey Heer. 2011. Divided Edge Bundling for Directional Network Data. *IEEE Trans. Vis. Comput. Graph.* 17, 12 (2011).

[31] Yongxia Skadberg and James R. Kimmel. 2004. Visitors' flow experience while browsing a Web site: its measurement, contributing factors and consequences. *Computers in Human Behavior* 20, 3 (2004).

[32] Tableau Website. Tableau Website. https://www.tableau.com

[33] Wenbo Tao, Xiaoyu Liu, Çagatay Demiralp, Remco Chang, and Michael Stonebraker. 2019. Kyrix: Interactive Visual Data Exploration at Scale. In *CIDR.* http://cidrdb.org/cidr2019/papers/p70-tao-cidr19.pdf

[34] Alexandru Telea and Ozan Ersoy. 2010. Image-Based Edge Bundles: Simplified Visualization of Large Graphs. *Comput. Graph. Forum* 29, 3 (2010).

[35] Christian Tominski, James Abello, and Heidrun Schumann. 2009. CGV - An interactive graph visualization system. *Comput. Graph.* 33, 6 (2009).

[36] Cagatay Turkay, Erdem Kaya, Selim Balcisoy, and Helwig Hauser. 2017. Designing Progressive and Interactive Analytics Processes for High-Dimensional Data Analysis. *IEEE Trans. Vis. Comput. Graph.* 23, 1 (2017).

[37] Hong Wei, Jagan Sankaranarayanan, and Hanan Samet. 2017. Measuring Spatial Influence of Twitter Users by Interactions. In *SIGSPATIAL Workshop on Analytics for Local Events and News.*

[38] Jia Yu and Mohamed Sarwat. 2020. Turbocharging Geospatial Visualization Dashboards via a Materialized Sampling Cube Approach. In *ICDE.*

[39] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. 1996. BIRCH: An Efficient Data Clustering Method for Very Large Databases. In *SIGMOD.*

[40] Wen-Yuan Zhu, Wen-Chih Peng, Ling-Jyh Chen, Kai Zheng, and Xiaofang Zhou. 2016. Exploiting Viral Marketing for Location Promotion in Location-Based Social Networks. *ACM Trans. Knowl. Discov. Data* 11, 2 (2016).

A EDGE-AWARE CLUSTERING ALGORITHM

Algorithm 1: Clustering vertices in a batch of edges E

```
Input: A new batch of edges $E$; a set of existing clusters $C$;
       and a range radius $\rho$
Output: Updated set of clusters $C$
foreach edge $e = (l, r)$ in $E$ do
    if distance(l,r) $\leq \rho$ then
        merge them to a point $m$;
        insert $m$ to its nearest cluster;
        continue;
    end
    $C_l = C$.rangeSearch($l$, $\rho$) ;    // find near clusters
    $C_r = C$.rangeSearch($r$, $\rho$);
    if $C_l$ is $\emptyset$ then
        create a cluster $c_l$ for $l$ and add $c_l$ to $C_l$;
    end
    if $C_r$ is $\emptyset$ then
        create a cluster $c_r$ for $r$ and add $c_r$ to $C_r$;
    end
    if $\exists c_l \in C_l, \exists c_r \in C_r$ with a super edge $(c_l, c_r)$ then
        $c_l$.insert($l$); $c_r$.insert($r$);
    else
        insert $l$ to its nearest cluster;
        insert $r$ to its nearest cluster;
        create a super edge between the two clusters;
    end
end
return $C$;
```

B EXTENDING EDGE-AWARE CLUSTERING

Edge-aware merging of clusters. The edge-aware clustering discussed so far is utilized on newly added edges and affects the decision of inserting both vertices into existing clusters or creating new clusters. Recall that our motivation for clustering the points is to reduce the clutter by reducing the number of super edges. We take advantage of the greedy approach of this clustering algorithm to further reduce the number of super edges by merging two super edges into one. We check if two super edges can be merged whenever a new point is inserted into a cluster.

Figure 15 demonstrates the merge operation due to the insertion of a new point into cluster a. When the new point is inserted into cluster a, the cluster's center may shift due to the new addition, as shown in Figure 15-i. Over time, this shift may cause the cluster's center to be within the range radius ρ of a nearby cluster, such as clusters e and c around a as shown in Figure 15-ii. We say two clusters are "overlapping" if their centers are within ρ. We take this opportunity to merge cluster a with one of its neighboring clusters to further reduce the clutter. However, if we merge the two clusters without considering the super edges connected to them, we may not solve the problem of reducing the number of super edges. For example, if we merge the clusters e and a, the number of super edges remains the same. To solve this problem, we add one more condition to the merge operation: two clusters can be merged only if their corresponding other vertices connected to the clusters are also overlapping. Using this approach, clusters a and c are merged into a larger set cluster g, where its center is the weighted average of the centers of the two clusters. Similarly, the other clusters connected to g are also merged, i.e., merge b and d into a cluster h, as shown in Figure 15-iii. Notice that clusters b and d were not merged prior to the addition of the new point as the condition was not satisfied.

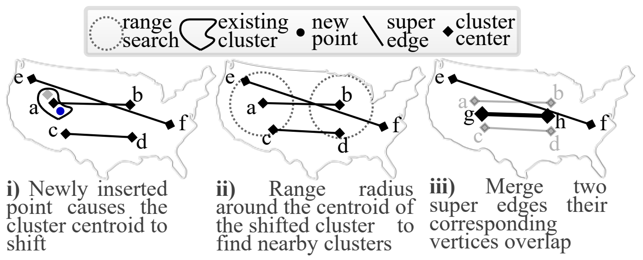

range search
existing cluster
new point
super edge
cluster center
e
b
a
f
c
d
e
b
a
f
c
d
e
a
b
g
h
f
c
d
i) Newly inserted point causes the cluster centroid to shift
ii) Range radius around the centroid of the shifted cluster to find nearby clusters
iii) Merge two super edges their corresponding vertices overlap

Figure 15: Progressive merging of super edges. We omitted the cluster shape for notational simplicity.

C PROGRESSIVE EDGE BUNDLING ALGORITHM

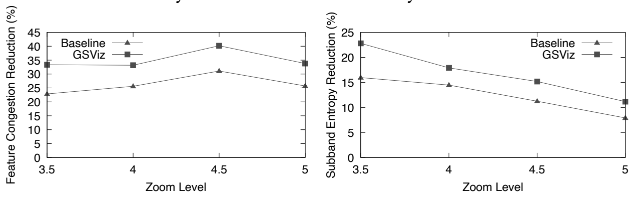
Algorithm 2: Incremental maintenance of PEB-tree and edge bundling

```
Input: A PEB-tree $T$ of edges of previous batches; and a
       new batch of edges $E$;
Output: Updated $T$ and a set of curved edges for $E$.
// Update $T$
foreach edge $e$ in $E$ do
    $n$ = create a new node ($e$,1);
    // get lowest non-leaf node with an edge
       compatible with $e$
    $n'$ = $T$.traverse($n$);
    if $n'$ is not found then
        $n'$ = create a new parent node for $n$;
    end
    $n'$.insert($n$) ;        // insert $n$ as a child of $n'$
end
// Bundle edges in $E$
$S = \emptyset$;
foreach edge $e$ in $E$ do
    let $n'(e', \omega)$ be the corresponding parent node of $e$;
    $\hat{e}$ = Edge-Curving($e$, $e'$, $\omega$) ;   // drag $e$ towards $e'$
    $S$.add($\hat{e}$) ;
end
return $(T, S)$ ;
```

D REDUCTION OF VISUAL CLUTTER EXPERIMENTAL RESULT.

We report the visual quality of the two methods used in the user study across the different zoom levels. We used two common metrics to measure the visual display clutter, namely "feature congestion" and "subband entropy" [28].

Figure 16 shows the percentage reduction on the visual clutter score using both metrics. The higher the percentage, means the reduction was more. Figure 16a shows that the baseline and GSViz on average reduced the clutter score by 26% and 35%, respectively. Figure 16b shows that the baseline reduced the subband entropy clutter score by 15% at zoom level 3.5 (which showed the entire US), and 7% at zoom level 5 (which showed cities). GSViz reduced the score by 22% and 11%, respectively. We observe that both methods reduced the visual clutter score compared to the original graph, and GSViz achieved a better reduction. This finding was consistent with the readability result in the user study.



(a) Feature Congestion. **(b) Subband Entropy.**

Figure 16: Reduction of visual clutter score compared to the original network.