MapSets: Visualizing Embedded and Clustered Graphs

Alon Efrat[1], Yifan Hu[2], Stephen G. Kobourov[1], and Sergey Pupyrev[1,3]

[1] Department of Computer Science, University of Arizona, Tucson, Arizona, USA
[2] Yahoo Labs, New York, USA
[3] Institute of Mathematics and Computer Science, Ural Federal University, Russia

Abstract. We describe MapSets, a method for visualizing embedded and clustered graphs. The proposed method relies on a theoretically sound geometric algorithm, which guarantees the contiguity and disjointness of the regions representing the clusters, and also optimizes the convexity of the regions. A fully functional implementation is available online and is used in a comparison with related earlier methods.

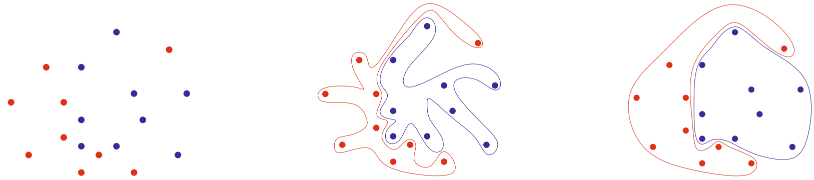
1 Introduction

In many real-world examples of relational datasets, groups of objects (clusters) are an inherent part of the input. For example, scientists belong to specific research communities, politicians are affiliated with specific parties, and living organisms are divided into biological species in the tree of life. Such clusters are often visualized with regions in the plane that enclose related objects. By explicitly defining the boundary and coloring the regions, the cluster information becomes evident. In many instances the data objects are often associated with fixed or relative positions in the plane. In geo-referenced data, for example, the positions of the objects might be based on their geographic coordinates. Thus a natural problem arises: How to best visualize graphs in which vertices are divided into clusters and embedded with fixed positions in the plane?

Several existing visualization approaches seem suitable. For example, methods for visualizing set relations over existing embedded pointsets, such as BubbleSets [6] and LineSets [2] use colored shapes to connect objects that belong to the same set. Alternatively, a geographic map metaphor can be used to represent such data. With self-organizing maps [22] or geometry-based GMaps [9], objects become cities and cluster information is captured by uniquely colored countries. While both approaches can produce compelling visualizations, we argue that neither is perfectly suited to the problem of visualizing embedded and clustered graphs.

As the number of sets increases, set-based methods generate complex and sometimes ambiguous results. More recent methods, such as KelpDiagrams [7] and KelpFusion [15], reduce visual clutter and guarantee unambiguous visualization. But more importantly, all of these methods result in overlapping regions for the sets, even when the input sets are disjoint. This unnecessarily increases visual complexity and might mislead the viewer about the disjointness of the sets. The geographic map approach suffers from a different problem. A country in the map, that represents a given cluster of vertices, might not be a contiguous region in the plane. Even though each cluster is colored with

C. Duncan and A. Symvonis (Eds.): GD 2014, LNCS 8871, pp. 452–463, 2014.



(a) (b) (c)

Fig. 1. (a) An embedded and clustered (red/blue) pointset. (b-c) Two different ways to construct contiguous shapes bounding points of the same color.

a unique color, such fragmented maps are difficult to read as human perception of color changes based on surrounding colors [19] and can be misinterpreted [11].

We want to combine the advantages of existing methods, while attempting to avoid their problems. That is, we are interested in visualizing embedded and clustered graphs with non-fragmented and non-overlapping regions. While constructing such representations is easy in theory, in practice the regions may still have high visual complexity; see Fig. 1. Ideally the regions should be as *convex* as possible, as the convex hull best captures cohesive grouping according to Gestalt theory [12].

With this in mind, we describe MapSets, a method for creating non-fragmented, non-overlapping regions that are as convex as possible, from a given embedded and clustered graph. We consider several criteria for measuring convexity of a shape, and propose a novel geometric problem aiming at optimizing convexity. We present a theoretical analysis of the problem in Section 3. Next, in Section 4, we describe a practical method for visualizing clustered graphs. A comparison of the method with existing techniques is provided in Section 5.

2 Related Work

Set Visualization. Graph clusters can be viewed as sets over graph vertices. In Venn diagrams and their generalization, Euler diagrams, closed curves correspond to (possibly overlapping) sets, and overlaps between the curves indicate intersections. Simonetto et al. [21] automatically generate Euler-like diagrams, by allowing disconnected regions, which can be complex and non-convex. Riche and Dwyer [20] propose a way to avoid the visual complexity problem by drawing simplified rectangular Euler-like diagrams, that do not depict the intersections between the sets explicitly, by duplicating objects that belong to multiple sets. In a user study, they found that it is beneficial to show intersections using simple set regions and strict containment, enabled by the duplication. For the setting where the positions of the objects are fixed, Collins et al. [6] present BubbleSets, a method based on isocontours to overlay such an arrangement with enclosing set regions. The readability of these visualizations suffer when there are many overlapping regions. LineSets [2] aim to improve the readability of complex set intersections and to minimize the overall visual clutter by reducing set regions to simple curved lines drawn through set elements. KelpDiagrams [7] incorporate classic graph-drawing “bubble and stick” style graph or tree spanners over the member points in a set. KelpFusion [15] adds filled-in regions to provide a stronger sense of grouping for close elements. A significant limitation of all these set visualization techniques is that they produce overlapping regions even when the sets are disjoint.

Visualizing Graphs as Maps. The geographic map metaphor is utilized as visual interface for relational data, where objects, relations between objects, and clustering are captured by cities, roads, and countries. Using maps to visualize non-cartographic data has been considered in the context of spatialization [22]. Maps of science showing groups of scientific disciplines are used by a wide range of professionals to grasp developments in science and technology [4].

The geographic map metaphor is used in the Graph-to-Map approach (GMap) [9]. GMap combines graph layout and graph clustering, together with appropriate coloring of the clusters and creating boundaries based on clusters and connectivity in the original graph. However, since layout and clustering are two separate steps, a region representing a cluster may often be fragmented; see Fig. 7(b). Such fragmentation makes it difficult to identify the correct regions and can result in misinterpretation of the map [11]. Note that in the setting when either an input embedding or clustering can be modified, the GMap approach can be improved to achieve contiguous regions [13].

Colored Spanning Trees. From an algorithmic perspective, our geometric approach of optimizing convexity of regions that cover points in the plane is related to several problems in which the input is a multicolored point set [1, 3]. The group Steiner tree problem deals with a graph with colored vertices, and the objective is to find a minimum weight subtree covering all colors [16]. Also related is the problem of computing spanning graphs for multicolored point set [10]. The problem is motivated by optimizing the amount of "ink" needed to connect monochromatic points that arise when visualizing sets using the KelpFusion technique. These trees cannot be directly used as "skeletons" of regions in the plane as they can result in overlapping regions.

3 Constructing Contiguous Non-overlapping Regions

We assume that the input instance consists of a set of objects P with fixed positions $p_i \in \mathbb{R}^2$ for all $i \in P$, for example, cities and their geographic locations. In practical applications labels are often associated with the objects. In this case, we assume that non-overlapping bounding boxes for the labels are given. The input also specifies a clustering $C = \{C_1, \ldots, C_k\}$ of the objects with $\cup_{i=1}^{k} C_i = P$ and $C_i \cap C_j = \emptyset$ for $i \neq j$. We wish to enclose all objects of the same cluster by a single contiguous region so that regions corresponding to different clusters do not overlap.

On one hand, simply overlaying each cluster with a convex region (e.g., bounding box or convex hull) is not always a valid solution, as it might cover elements in other clusters. On the other hand, representing clusters by some minimal regions (e.g., spanning or Steiner trees) is also not always valid, as it might result in intersecting regions.

We require regions that are contiguous and disjoint, and it is not difficult to see that such regions can be easily computed. We can begin by computing a crossing-free spanning tree of points belonging to some cluster. Once the tree is constructed, its vertices and edges become "obstacles" that should be avoided by subsequent trees. Note that

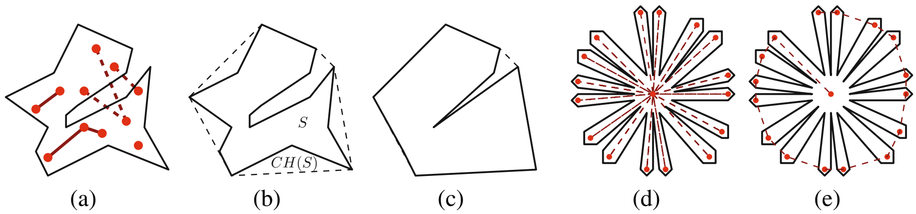


Fig. 2. Convexity measures for a shape S enclosing red points. (a) Solid segments are within S, while dashed ones are not. (b) A shape and its convex hull (dashed). (c) Area-based measure ignores boundary defects. (d-e) Ink needed to connect the points is much bigger than the length of the minimum spanning tree. The shape is enclosed in solid black, while the tree is dashed red.

all the clusters will be processed as the trees do not separate the plane into more than one region. Finally, contiguous non-overlapping regions can be grown, starting from these disjoint trees. However, this procedure often generates "octopus"-like shapes that are neither aesthetically pleasant nor practically useful for visualization; see Fig. 1. Hence, we require a method for creating regions that are as convex as possible. In order to design such a method, a quality criterion for measuring the convexity of regions is needed. Next we review and formalize several convexity measures.

3.1 Convexity Measures

A shape S is said to be convex if it has the following property: If points $p, q \in \mathbb{R}$ belong to S then all points from the line segment $[pq]$ belong to S as well. The definition allows for several different ways to measure the convexity of non-convex shapes.

Point/Vertex Visibility. For a given shape S, this convexity measure is defined as the probability that for points p and q, chosen uniformly at random from S, all points from the line segment $[pq]$ also belong to S [24]. The result is a real number from $[0, 1]$, with 1 corresponding to convex shapes. A problem with this definition is that it is difficult to compute, even if S is a polygon. Hence, we consider its discrete variant, taking into account that the input of our problem specifies points in the plane; see Fig. 2(a).

This vertex-based measure takes into account how many segments $[pq]$ are completely in S for pairs of input points $p, q \in P$ of the cluster corresponding to S. The measure is defined as $\frac{\sum_{p,q \in P} \delta(p,q)}{|P|^2}$, where the sum is over all pairs of input points P and $\delta(p, q) = 1$ if $[pq]$ lies inside S and $\delta(p, q) = 0$, otherwise.

Convex Hull Area/Perimeter. Recall that the smallest convex set which includes a shape S is called the *convex hull*, $CH(S)$, of S; see Fig. 2(b). The area-based convexity measure is defined as $\frac{Area(S)}{Area(CH(S))}$; it is frequently used and appears in textbooks [23]. The result is a real number from $[0, 1]$, with 1 corresponding to convex shapes. Unlike visibility-based measures, the convex hull-based one is very easy to calculate efficiently and is robust with respect to noise. However, the definition does not allow to detect defects on boundary that have a relatively small impact on the shape area; see Fig. 2(c). The perimeter-based definition attempts to remedy this: $\frac{Perimeter(S)}{Perimeter(CH(S))}$.

(a) (b)

Fig. 3. (a) An input for CST with $n = 10$ points and $k = 3$ colors. (b) An optimal solution with minimum ink containing Steiner points.

If a shape S is convex, then there exists a minimum spanning tree on the given point set such that every edge of the tree lies completely in S; non-convex shapes do not necessarily admit such a spanning tree. Hence, the length of a shortest curve that belongs to S and connects all the input points is an indicator of convexity of S. In the following measure, we compare the length of such a curve (or equivalently, the amount of "ink" needed to connect all the points) with the length of a minimum spanning tree on the same point set; see Figs. 2(d)-2(e).

Minimum Ink. Let $|\,\mathrm{INK}(P)|$ be the length of the shortest curve connecting all vertices of V lying in S, and let $|\,\mathrm{MST}(P)|$ be the length of the minimum spanning tree of V. The measure is defined as $\frac{|\,\mathrm{MST}(P)|}{|\,\mathrm{INK}(P)|}$. Again, 1 indicates the best possible value (though, it does not always correspond to a convex shape); smaller values are worse.

There are advantages and disadvantages of all of the proposed convexity measures, and there are also many other ways to define convexity of shapes or polygons. In an attempt to balance theoretical and practical considerations, we focus on visibility-based and the ink-based measures. Similar ink-based criteria are used for constructing LineSets and KelpDiagrams. By minimizing the ink needed for drawing, all of these techniques aim to reduce visual clutter and increase the readability of the representation.

3.2 Algorithm for Ink Minimization

Here we study a problem motivated by computing contiguous regions with minimum ink. The input consists of n points in the plane, and each point is associated with one of k colors. The CST (COLORED SPANNING TREES) problem is to connect points of the same color by mutually non-intersecting curves of shortest total length. In an optimal solution each curve forms a tree spanning points of the corresponding color. The trees may use additional (Steiner) points that do not belong to the original pointset; see Fig. 3.

Computing an optimal solution for CST is NP-hard. This follows from the observation that the known NP-complete MINIMUM STEINER TREE problem is a special case of CST, in which the input consists of monochromatic points. Next we present a heuristic for CST and prove that it is an approximation algorithm in the theoretical sense.

We refer to the minimum spanning and Steiner trees of a set of points P as $\mathrm{MST}(P)$ and $\mathrm{SMT}(P)$, respectively; their lengths are denoted by $|\,\mathrm{MST}(P)|$ and $|\,\mathrm{SMT}(P)|$. We use the Steiner ratio, denoted by ρ, which is the supremum of the ratio of the length of a minimum spanning tree to the length of a minimum Steiner tree. It is conjectured that $\rho = \frac{2}{\sqrt{3}} \approx 1.15$, and ≈ 1.21 is the best-known upper bound on ρ [5].

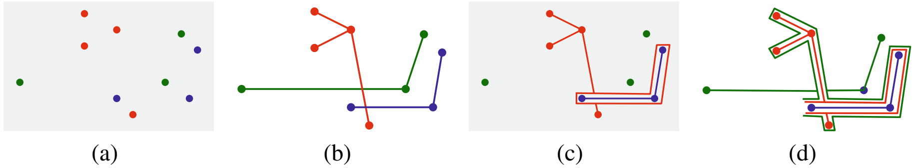


Fig. 4. Steps of the algorithm for the CST problem. (a) An input with $n = 10$ points and $k = 3$. (b) Computing minimum spanning trees. (c) Bounding the tree having the shortest length, and removing red-blue crossings. (d) Merging with the green tree.

We begin with the description of our algorithm in the setting when the input consists of blue and red points. First, we compute a minimum spanning tree of the blue points (ignoring the red ones), and a minimum spanning tree of the red points; see Fig.4(b). If the trees do not intersect, then they form a solution for CST. Otherwise, we create a red "shell" bounding the blue tree; see Fig.4(c). Now red-blue crossings appear inside the constructed shell, and they can be eliminated by removing all portions of the red tree inside the shell. Finally, the red curve, consisting of the original spanning tree and the constructed shell, can be transformed to a tree by disconnecting its cycles; see Fig.4(d).

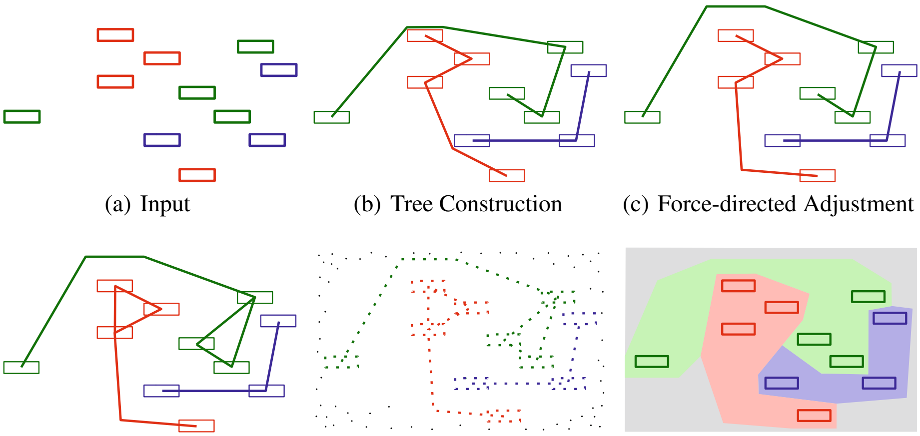
The general algorithm works in the following steps. First, create a minimum tree $\mathrm{MST}(C_i)$ spanning the set of points C_i for $1 \leq i \leq k$, ignoring points of the other colors. Sort the colors with respect to the length of the corresponding spanning trees. Without loss of generality, we may assume that the resulting order is $C_1, \ldots, C_k$ and $|\mathrm{MST}(C_1)| \leq \cdots \leq |\mathrm{MST}(C_k)|$. Then the resulting curve for C_1 is the tree $\mathrm{MST}(C_1)$. A curve for each successive color C_i is constructed by adding a "shell" bounding the curve corresponding to C_{i-1}. The length of the shell is exactly $2\sum_{j<i}|\mathrm{MST}(C_j)|$, since it bounds all the spanning trees corresponding to already processed colors; see Fig. 4. The length of a curve for C_i is then $|\mathrm{MST}(C_i)| + 2\sum_{j<i}|\mathrm{MST}(C_j)|$.

In order to analyze the algorithm, we denote the amount of ink in the optimal solution by OPT, and the total length of the constructed solution by ALG. An optimal solution induces a curve connecting all points of the same cluster, that is, the solution is a Steiner tree for the set of points (but not necessarily the minimum one). Hence, $\mathrm{OPT} \geq \sum_i |\mathrm{SMT}(C_i)| \geq \sum_i |\mathrm{MST}(C_i)|/\rho$. On the other hand,

$$\mathrm{ALG} \leq \sum_{i=1}^{k}\left(|\mathrm{MST}(C_i)| + 2\sum_{j=1}^{i-1}|\mathrm{MST}(C_j)|\right) = \sum_{i=1}^{k}(2k-2i+1)|\mathrm{MST}(C_i)|, \text{ and}$$

$$\frac{\mathrm{ALG}}{\mathrm{OPT}} \leq \frac{\sum_{i=1}^{k}(2k-2i+1)|\mathrm{MST}(C_i)|}{\sum_{i=1}^{k}|\mathrm{MST}(C_i)|/\rho} =$$
$$= \rho\frac{\sum_{i=1}^{\lfloor k/2\rfloor}(2k-2i+1)|\mathrm{MST}(C_i)| + \sum_{i=1}^{\lceil k/2\rceil}(2i-1)|\mathrm{MST}(C_{k-i+1})|}{\sum_{i=1}^{k}|\mathrm{MST}(C_i)|} \leq k\rho.$$

Hence, our algorithm is a $(k\rho)$-approximation for the CST problem for any $k \geq 1$.



(a) Input (b) Tree Construction (c) Force-directed Adjustment

(d) Edge Augmentation (e) Adding Auxiliary Points (f) Computing Map Regions

Fig. 5. Algorithmic pipeline of MapSets

4 MapSets

Here we describe MapSets, starting with a high-level overview; see Fig. 5. We assume that the input is a set of rectangular shapes (bounding boxes of labels) embedded in the plane along with a clustering. In the first step, we compute spanning mutually non-crossing trees interconnecting centers of rectangles corresponding to the same cluster, while minimizing the total ink needed to draw the trees. In the second step, we modify the trees by adding buffers of free space around the segments of the trees, using a force-directed heuristic. In the third step, we try to optimize the convexity of the resulting regions based on the vertex visibility measure, by adding edges between vertices in the same cluster, while ensuring that edges of different clusters do not cross. In the fourth step, we use the modified trees and the added edges to build contiguous non-overlapping boundaries for all clusters.

Tree Construction. In order to construct the trees, we employ the approximation algorithm described in Section 3.2. For each cluster, we first compute a minimum tree spanning the set of rectangle centers, ignoring other clusters. The clusters are then sorted in non-decreasing order by the length of the computed trees and processed in this order. At each step we consider all the precomputed trees as obstacles that should be avoided when constructing the current tree. The rectangles are also treated as obstacles. We compute a sparse visibility graph on the set of obstacles, where the vertices are all the centers and corners of the rectangles, and there is an edge between two vertices if one can draw a straight-line segment without crossing the obstacles. The sparse visibility graph (unlike the full visibility graph) has a linear number of edges and can be constructed efficiently [8]. We then compute shortest paths (of the visibility graph) between every pair of rectangles of the current cluster. From these shortest paths, we compute a minimum spanning tree for the current cluster. We add the tree to the set of obstacles and proceed with the next cluster.

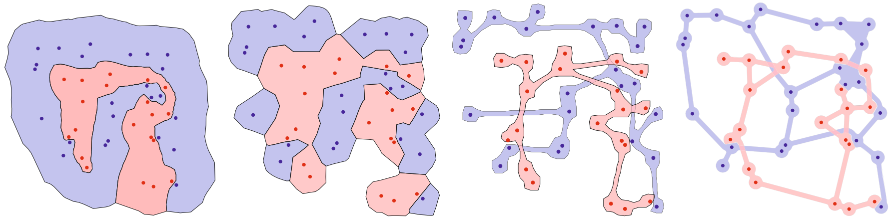
Force-directed Adjustment. This step improves the constructed trees. Our goal is to provide some free space around the edges of the trees so as to avoid (1) narrow channels between parts of the same region and (2) region borders lying too close to the input vertex labels. To accomplish this, we consider an adjustment graph G^{adj} in which vertices are the end points and bends of the constructed trees and edges are maximal straight-line segments of the trees. We then build a force system moving the vertices of G^{adj} that correspond to the bends of the tree. The system relies on the following forces.

- **Vertex-vertex Attraction.** We would like to keep the ink of the drawing low. Therefore, for every vertex of G^{adj}, there is a force pushing the vertex towards its neighbor vertices in G^{adj}.
- **Edge-edge Repulsion.** This repulsive force attempts to push the edges of G^{adj} apart to provide enough space to draw the regions. In order to compute the force, it is convenient to replace edges of G^{adj} with cylinders of a specified thickness. Then, if two cylinders corresponding to different trees intersect, the force repels them away from each other. This force also ensures that the trees do not overlap and do not intersect during the adjustment process.
- **Edge-label Repulsion.** This force prevents edges from being routed too close to the input text labels. Again, it is convenient to consider the edges of G^{adj} as cylinders. If a cylinder occludes a label, then we introduce a repulsive force moving the corresponding vertices of G^{adj} away from the label.

We use iterative refinement similar to that used in drawing graphs with edge bundles [18] to adjust the positions of the vertices of G^{adj} under these three forces: repulsive forces have equal priorities, and the attractive force is weaker. In our experiments, the force system provides the desired buffer of free space around the trees and converges quickly; see Fig. 8.

Edge Augmentation. In this step we try to optimize the convexity of the regions using the vertex visibility metric. Consider all possible straight-line segments connecting centers of rectangles corresponding to the same cluster. Our goal is to select and add as many of these segments as possible, subject to the condition that they do not cross each other. To this end, we construct a graph H in which vertices are the straight-line segments. A segment is added to H only if it does not intersect the trees found in the previous step. Two vertices of H are connected by an edge if the corresponding straight-line segments cross each other. Notice that now the problem reduces to the problem of finding a maximum non-crossing (independent) set of segments in the plane. The problem can be solved optimally in polynomial time for two clusters, that is, if $k = 2$. Indeed, in this setting the graph H is bipartite, and the size of a maximum independent set in a bipartite graph equals to the number of edges in a minimum edge covering by König's theorem. The latter can be found using a maximum matching algorithm. Unfortunately, the general variant is NP-hard even for $k = 3$ [14]. Therefore, unless $k = 2$, we use a greedy strategy to solve the problem. At every step, we choose the minimum degree vertex in H and remove its neighbors. It is well-known that this strategy guarantees an approximation ratio of $(\Delta + 2)/3$ on graphs with maximum degree Δ.

Adding Auxiliary Points and Computing Map Regions. Given the initial placement of the labels and curves connecting the labels from the previous steps, we need explicit



(a) MapSets (b) GMap (c) BubbleSets (d) KelpFusion

Fig. 6. The senator voting graph (the part of the U.S. west of Mississippi). The vertices are senators (red republicans and blue democrats) positioned according to their home-cities.

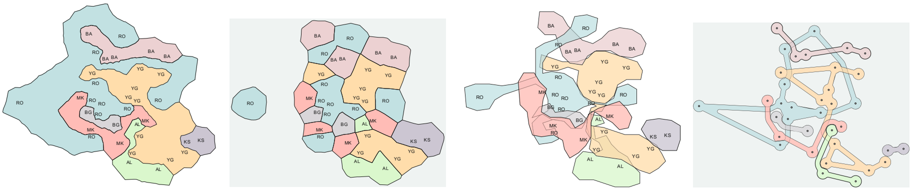
regions grouping together labels and curves in the same cluster. As in GMap, we generate boundaries by adding dummy points to the current embedding. There are three types of the dummy points: (a) random points, sufficiently far away from the set of the input labels, lead to more rounded and thus more realistic region boundaries; (b) random points along bounding boxes of the labels help ensure that the labels are drawn inside the regions; (c) auxiliary points along all the edges constructed on the previous step, that keep the regions connected. The distance between consecutive points on an edge is chosen to be less than the distance to any other point of a different color. After adding the dummy points, we compute the Voronoi diagram of the set of all points and merge the Voronoi cells that belong to the points of the same color.

Time Complexity. Now we discuss the complexity of our algorithm on an input with n points and k clusters, assuming we can compute distances and intersections between geometric primitives (points, line-segments, rectangles) in constant time. The sparse visibility graph can be constructed in $O(n \log n)$ time and it contains $O(n)$ edges [8]. Therefore, computing all pairwise distances takes $O(n^2)$ time and finding a minimum spanning tree for one cluster takes $O(n^2 + n \log n)$ time. Summing over all clusters, we get $O(kn^2)$. In the iterative force-directed heuristic we compute forces between pairs of edges, which can take $O(n^2)$ in the worst case. Hence, the time complexity of the force-directed heuristic is $O(cn^2)$, where c is the maximum number of iterations in the adjustment ($c = 10$ in our implementation). The complexity of the edge augmentation step is $O(n^3)$, as we may add quadratic number of edges in the greedy process. Finally, computing the boundaries takes $O(n \log n)$ time. Therefore, the overall time complexity is $O(kn^2 + n^3)$. More details and actual running times are given in the next section.

5 Experiments

Here we compare our new algorithm, MapSets, with the existing approaches for map-like visualizations: GMap [9], BubbleSets [6], and KelpFusion [15]. A fully functional implementation of MapSets, GMap, and BubbleSets, together with a complete dataset, is available in an online system at `http://gmap.cs.arizona.edu`.

Our first example is the senator voting graph; see Fig. 6. The vertices in the graph are the U.S. senators in 2010 positioned according to their home-cities in the U.S. The



(a) MapSets (b) GMap (c) BubbleSets (d) KelpFusion

Fig. 7. The graph of genetic similarities between 50 individuals in Europe. The layout is computed using the principal component analysis, while the clusters correspond to the countries of origin of the individuals.

clustering is based on the political party they represent, red for republicans and blue for democrats. Clearly, both clustering and geographic information of the vertices are fixed and cannot be changed. One can see that GMap produces fragmented clusters, while BubbleSets and KelpFusion compute overlapping regions. On the other hand, the result of MapSets is contiguous and non-overlapping, which makes it easier to analyze the distribution of senators over the map.

The second example shows the population structure within Europe [17]. The original points correspond to genetic data from $1,387$ Europeans (but we sampled only 50 vertices corresponding to Eastern Europe for illustration purposes). The positions of the vertices come from the original principal component analysis, based on the similarity matrix. As the authors point out, the PCA plot (appropriately rotated) closely matches the geographic outlines of Europe; hence, it is undesirable to change the node positions. The clusters are extracted independently and corresponds to the countries of origin of the individuals. Again, only MapSets constructs non-fragmented disjoint regions; see Fig. 7. Arguably, this is easier to analyze than the overlapping regions produced by BubbleSets and KelpFusion.

We next analyze the performance of our ink minimization heuristic. To this end, we utilize a collection of 9 real-world networks, that are embedded and clustered using the GMap tool with the default setting. Table 1 gives details about the graphs and measurements of our ink saving algorithm. Here, ALG shows the ratio of the total ink of the computed trees to the total length of the minimum spanning trees computed individually for every cluster. In other words, this is an approximation factor achieved by our algorithm on the test cases. Although we can only guarantee factor $k\rho$, in practice the algorithm performs very well, always producing a solution at most 1.6 times worse than the optimal. Our experiments indicate that ink minimization strategy often results in aesthetically more pleasant map visualizations.

Similarly, ALG_{fd} indicates the utilized ink after the force-directed adjustments. As expected, the ink increases after the step, but the increase is not significant. On the other hand, the adjustments improve the quality of the resulting regions.

Table 1. Measurements of MapSets on test cases: ALG and ALG_{fd} stand for the ratio between the total ink of the drawing and the total length of the minimum spanning trees after the steps *Tree Construction* and *Force-directed Adjustment*, respectively.

graph	$\|P\|$	k	ALG	ALG_{fd}
Colors	50	6	1.002	1.012
GD	506	23	1.582	1.612
Recipes	381	15	1.356	1.502
Trade	211	8	1.101	1.259
Universities	161	8	1.366	1.443
SODA	316	11	1.204	1.296
IPL	336	11	1.337	1.414
SOCG	500	11	1.492	1.601
TARJAN	252	16	1.150	1.197
ALGO	500	5	1.547	1.650

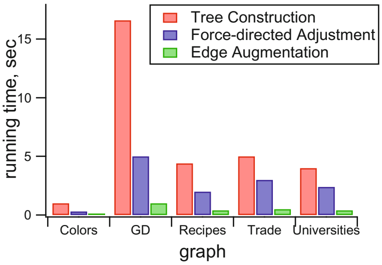


Fig. 8. Running times of the different steps of MapSets on some of the test cases.

The algorithm is implemented in C++. We use a machine with Intel i5 3.2GHz and 8GB RAM for measuring running time; see Fig. 8. The last two steps, *Adding Auxiliary Points* and *Computing Regions*, are very efficient taking few milliseconds for the largest graphs, and hence are not included in the chart. The first step, *Tree Construction*, is usually the most time consuming; it is more efficient for nearly contiguous clusters (e.g, Colors) and less efficient for graphs with many fragments (e.g., GD). Although *Edge Augmentation* theoretically has cubic time complexity, it is among the fastest steps in practice, because there are usually not many edges added. Overall, our algorithm processes all the graphs (most with hundreds of vertices) in less than a minute. This is slower than the GMap and LineSets but comparable to BubbleSets. Since our algorithm extensively utilizes many primitive geometric operations (e.g., testing for segment intersections), using a specialized geometric library will likely improve the performance.

6 Conclusion and Future Work

We designed and implemented a new approach for visualizing embedded and clustered graphs. Unlike existing techniques, our MapSets method always produces contiguous and non-overlapping regions. Results of the initial evaluations seem promising. We also presented a simple approximation algorithm for the geometric problem of ink minimization motivated by the method. A natural future direction is to improve the approximation factor. It would be also worthwhile to carefully evaluate different convexity measures and select one that offers the best balance between ease of computation and visual quality of the resulting regions. Similarly interesting would be in-depth user study comparing map-based visualizations constructed with different approaches considered in the paper.

Acknowledgements. The work supported in part by NSF grants CCF-1115971 and DEB 1053573. We thank the authors of [17] for the DNA dataset. The drawings of KelpFusion are courtesy of the authors of [15].

References

1. Agarwal, P.K., Edelsbrunner, H., Schwarzkopf, O., Welzl, E.: Euclidean minimum spanning trees and bichromatic closest pairs. Discrete & Comput. Geom. 6(1), 407–422 (1991)
2. Alper, B., Riche, N.H., Ramos, G., Czerwinski, M.: Design study of LineSets, a novel set visualization technique. IEEE Trans. Vis. Comput. Graphics 17(12), 2259–2267 (2011)
3. Arora, S., Chang, K.: Approximation schemes for degree-restricted MST and red–blue separation problems. Algorithmica 40(3), 189–210 (2004)
4. Boyack, K.W., Klavans, R., Börner, K.: Mapping the backbone of science. Scientometrics 64, 351–374 (2005)
5. Chung, F., Graham, R.: A new bound for Euclidean Steiner minimal trees. Annals of the New York Academy of Sciences 440(1), 328–346 (1985)
6. Collins, C., Penn, G., Carpendale, S.: Bubble sets: Revealing set relations with isocontours over existing visualizations. IEEE Trans. Vis. Comput. Graphics 15(6), 1009–1016 (2009)
7. Dinkla, K., van Kreveld, M.J., Speckmann, B., Westenberg, M.A.: Kelp diagrams: Point set membership visualization. Comput. Graph. Forum 31(3, pt1), 875–884 (2012)
8. Dwyer, T., Nachmanson, L.: Fast edge-routing for large graphs. In: Eppstein, D., Gansner, E.R. (eds.) GD 2009. LNCS, vol. 5849, pp. 147–158. Springer, Heidelberg (2010)
9. Hu, Y., Gansner, E.R., Kobourov, S.G.: Visualizing graphs and clusters as maps. IEEE Comput. Graphics and Appl. 30(6), 54–66 (2010)
10. Hurtado, F., Korman, M., van Kreveld, M., Löffler, M., Sacristán, V., Silveira, R.I., Speckmann, B.: Colored spanning graphs for set visualization. In: Wismath, S., Wolff, A. (eds.) GD 2013. LNCS, vol. 8242, pp. 280–291. Springer, Heidelberg (2013)
11. Jianu, R., Rusu, A., Hu, Y., Taggart, D.: How to display group information on node-link diagrams: An evaluation. IEEE Trans. Vis. Comput. Graphics 20(11), 1530–1541 (2014)
12. Kanizsa, G., Gerbino, W.: Convexity and symmetry in figure-ground organization. Vision and Artifact, 25–32 (1976)
13. Kobourov, S.G., Pupyrev, S., Simonetto, P.: Visualizing graphs as maps with contiguous regions. Comput. Graph. Forum (2014)
14. Kratochvíl, J., Nešetřil, J.: Independent set and clique problems in intersection-defined classes of graphs. Commentationes Math. Univ. Carolinae 31(1), 85–93 (1990)
15. Meulemans, W., Riche, N., Speckmann, B., Alper, B., Dwyer, T.: KelpFusion: A hybrid set visualization technique. IEEE Trans. Vis. Comput. Graphics 19(11), 1846–1858 (2013)
16. Mitchell, J.S.: Geometric shortest paths and network optimization. Handbook of Computational Geometry 334, 633–702 (2000)
17. Novembre, et al.: Genes mirror geography within Europe. Nature 456(7218), 98–101 (2008)
18. Pupyrev, S., Nachmanson, L., Bereg, S., Holroyd, A.E.: Edge routing with ordered bundles. In: van Kreveld, M., Speckmann, B. (eds.) GD 2011. LNCS, vol. 7034, pp. 136–147. Springer, Heidelberg (2011)
19. Purves, D., Lotto, R.B.: Why we see what we do: An empirical theory of vision. Sinauer Associates (2003)
20. Riche, N.H., Dwyer, T.: Untangling Euler diagrams. IEEE Trans. Vis. Comput. Graphics 16(6), 1090–1099 (2010)
21. Simonetto, P., Auber, D., Archambault, D.: Fully automatic visualisation of overlapping sets. Comput. Graph. Forum 28(3), 967–974 (2009)
22. Skupin, A., Fabrikant, S.I.: Spatialization methods: a cartographic research agenda for non-geographic information visualization. Cartogr. Geogr. Inform. 30, 95–119 (2003)
23. Sonka, M., Hlavac, V., Boyle, R.: Image Processing, Analysis, and Machine Vision. Thomson-Engineering (2007)
24. Zunic, J., Rosin, P.L.: A convexity measurement for polygons. IEEE Trans. Pattern Anal. Mach. Intell. 26, 173–182 (2002)