A Simple Approach for Boundary Improvement of Euler Diagrams

1 INTRODUCTION

Euler diagrams [10] are an intuitive way to visualize sets and their contents, because they visualize set content in the most straightforward way possible: sets in these diagrams are polygons and if the polygons overlap, the sets intersect. Recent techniques aim at automating the process of drawing Euler diagrams with closed curves to directly visualize the intersections of sets. Elements membership can easily be represented using enclosure by these curves. Given a list of sets and their intersections, these general techniques are powerful enough to produce a drawing for any input instance [18, 21, 8, 17, 20, 3]. Because these drawing are intuitive, Euler diagrams have been an important visualization technique applied in various fields including: bioinformatics [4, 11], digital humanities [28], social networks [15], multimedia database queries [27], and many others.

An issue with these general techniques is the shape of the polygons that represent the sets in these drawings are often jagged and elongated due to the large number of constraints that the drawing must satisfy to be valid. Intuitively, rounder polygons are more aesthetically pleasing. Empirical evidence provides support for smooth closed curves as they pop out visually [12, 26, 16], and early empirical evidence into the readability of Euler diagrams suggests that smooth set contours are perceptually more effective [6]. Thus, an approach that is able to transform the output of any of the above approaches into a smoother diagram has the potential to improve its aesthetics and its readability.

In this paper, we introduce a simple method that is able to significantly improve the boundaries of any Euler diagram drawing approach that uses polygons and their overlap to indicate the relationship between sets of elements. The approach proposed is a highly simplified form of curve shortening flow for shape improvement [24, 25, 9, 31, 7] applied to the problem of Euler diagram drawing. Such techniques, which have often been used in geometry processing, graphics, and scientific visualization, to our knowledge, have not been applied to the problem of Euler diagram drawing or the area of information visualization in general. The essence of our approach is to derive a new force system based on boundary curvature. This force system can be applied in conjunction with diagram constraints to greatly improve the appearance of the Euler diagram while ensuring a correct drawing.

The primary contribution of this paper is an algorithm that combines a simplified form of curve shortening flow with Euler diagram drawing methods to improve the smoothness of any Euler diagram drawn with curves. The approach we propose is general and can be applied to the output of all Euler diagram drawing methods that represent sets as closed curves and intersections as overlap between those curves. Our approach is significantly more scalable than existing techniques to improve the smoothness of Euler diagrams. More importantly, the approach can guarantee a valid drawing for every input instance. Thus, the refined drawing has precisely the same topology as the Euler diagram provided as input – at no point are intersections between the polygons created or destroyed during refinement. Finally, the approach is easy to implement, consisting of only four forces, greatly simplifying the algorithm when compared to competitive approaches. We demonstrate the effectiveness of this method through case studies and metric evaluations that compare our proposed technique to state-of-the-art techniques.

2 RELATED WORK

The related work for this work combines two seemingly disparate areas of work: shape improvement and Euler diagram drawing. In this section, we describe the related work in these areas and how they can be combined to improve the curvature of Euler diagrams.

2.1 Shape Improvement and Mean Curvature Flow

Mean curvature flow is a classic method for smoothing shapes. Each point on the curve (in two dimensions) or the surface (in three dimensions) is moved in the direction of the normal in a way that is proportional to its curvature. This process flattens the object [9]. Unfortunately, mean curvature flow also also tends to shrink the object,
which may not be desired.

Taubin [24, 25] proposes $\lambda - \mu$ smoothing which is a classic method to avoid shrinking of the surface during smoothing. The approach works by alternating the direction of improvement at every other step. Taubin shows that his proposed algorithm, $\lambda - \mu$ smoothing, is able to smooth out rough parts of the model while preserving volume through an elegant analogy to Fourier space operations – the algorithm essentially reduces the amount of energy in high-frequency components while preserving the total amount of energy in the system. In this paper, we use the principle behind $\lambda - \mu$ to move vertexes of the boundary along the direction of curvature. However, as set elements are located inside the diagram, we can use the repulsive forces of the elements for area preservation.

2.2 Euler Diagram Drawing

Methods for visualizing sets and their intersections has been an active area of research in recent years with many novel visualization approaches developed for this purpose [3]. As the algorithm we propose in this work improves diagrams that represent sets as polygons, we focus on these methods for set visualization in this section. However, it is important to note that other approaches exist for visualizing sets and their intersections that are based on circles [11, 22, 29], ellipses [13], and approaches that are not based on closed curves at all [1, 2].

A number of approaches are able to produce a drawing for every input instance. Rodgers et al. introduce a method for drawing a subclass of Euler diagrams [19], and then extend this research to work on any input instance [18]. In these approaches, the dual graph of the Euler diagram is transformed into a planar graph for drawing. Then, a triangulation is used to route set contours through the plane, creating a diagram. Simonetto et al. [21] propose a force-directed solution that is guaranteed to produce a correct drawing. In this approach, a planar representation of the Euler diagram is computed and drawn initially with a planar graph drawing algorithm. Subsequently, the drawing is improved using a modification of the PrEd [5] graph drawing algorithm to ensure that the structure of the Euler diagram does not change, nodes and edges do not cross boundaries during force-directed refinement, and that elements of the sets stay in their proper zones. This approach was further improved in later work [20], allowing for boundaries with adaptive complexities and faster convergence. Collins et al. [8] describe a method for drawing Euler diagrams when the elements of the sets have fixed positions. The approach relies on marching squares and implicit curves to derive the boundaries for each set. Riche and Dwyer [17] describe a method where splits to the sets involved in the diagram are connected with edges and an approach that prioritizes containment. Stapleton et al. [23] present a method for inductively adding curves to a diagram and preserving well-formedness properties in the drawing.

The research proposed is a simple approach for improving the appearance of any Euler diagrams drawn using polylines. In fact, we demonstrate our technique operating on the output of many of these algorithms [21, 8, 17, 14].

The closest work to our own is EulerForce [14] where boundaries and curve positions have been modified to improve the readability of the Euler diagram. This approach is based on a force system, consisting of fourteen forces, to optimize the shape of the curves in the diagram, driving them towards circular shapes.

In contrast, the approach presented in this paper is fundamentally different than this approach. Our force system is simpler, consisting of only four forces, which helps improve complexity in terms of execution speed and implementation. The results are more scalable and able to refine Euler diagrams of realistic complexity beyond the five or so sets of EulerForce. Finally, and most importantly, our approach can be configured to never produce an invalid drawing — that is, we can impose constraints such that no new zones are created or destroyed by the refinement procedure.

3 Method

In this section, we present a description of the proposed approach. Firstly, we describe how our simplified version of curve shortening flow can be adapted in order to optimize the smoothness of polygons. Then, we will analyse differences and similarities between the original and the adapted formulation of this method. Finally, we identify additional requirements needed in order to successfully apply this method to the optimisation of Euler diagrams.

3.1 Discrete Curve Shortening Flow

Curve shortening flow is defined on a continuous curve and continuous time, ideally requiring the computation of infinite points for infinitesimal increments of time (see Fig. 2a). Instead, the optimization of Euler diagrams operates in discrete time and space, as vertex positions can only be optimized on a discrete basis and the curves themselves are modelled as polylines. Thus, we must adapt the original method to:

- **Discrete Space**. The concept of curvature, which is well defined for a sufficiently smooth curve, cannot be directly applied to a vertex $u$ of a polygon $\ldots tuv \ldots$. The centre of curvature should be placed perpendicular to the line normal at $u$, but this line is not uniquely defined as the slope of the secant for a left and right increment from $u$ only correspond when $t, u, v$ are collinear.

- **Discrete Time**. In continuous curve shortening flow, the movement speed of a point can approach infinity when the radius of the osculating circle approaches zero. In continuous time this never occurs, as the speed of the vertex is re-calculated after an infinitesimally small period of time in the new lower stress position of the vertex. However, in discrete time, the time period $t$ cannot be infinitesimally small, and therefore points with high speed will cause abrupt movements in $t$. Put simply, we require a more conservative approach to node movement in order to reduce the impact of large movements.

When defining our method for computing direction and intensity of movement, the properties of the original formulation must be preserved as much as possible. This formulation should be easy to implement and fast to compute. A solution that adequately satisfies all of these requirements compute the movement of a vertex $u$ in $\ldots tuv \ldots$ along the vector that connects $u$ to the centroid of the triangle $\Delta uv$ (see Fig. 2b).

**Definition 1.** Given a polygon $p = p_0 \ldots p_n$, we define smoothing on $p_i$ as the operation that transforms $p$ into $p' = p_0 p_{i-1} b p_{i+1} \ldots p_n$ where $b$ is the centroid of $\Delta p_{i-1} p_i p_{i+1}$.
We describe properties of the centroid smoothing technique defined in Theorem 2. Each polygon $p$ asymptotically converges to a single point.

**Theorem 1.** Given a convex polygon $p$, by performing smoothing on $p$ as the application of smoothing on each polygon vertex $p_i$.

**Definition 2.** Given a polygon $p = p_0 \ldots p_n$, we define smoothing on $p$ as the application of smoothing on each polygon vertex $p_i$.

**3.2 Comparison with Curve Shortening Flow**

We describe properties of the centroid smoothing technique defined above and we discuss similarities and differences between the definitions of discrete and continuous curve shortening flow.

**Theorem 1.** Given a convex polygon $p$, by performing smoothing on any of its vertexes, we obtain a polygon $p'$ that is still convex.

**Proof.** A polygon is convex iff all its internal angles are equal or less than 180°. Let $p = \ldots tuv\ldots$ and $u$ be the vertex selected for smoothing. Since $p$ is convex, the internal angles of $t, u, v$ are equal or less than 180°. By performing smoothing on $u$, we substitute $u$ with the centroid $b$ of $\Delta tuv$. This operation does not increase the internal angle of vertexes $t$ and $v$, which will still be equal or less than 180°. Also, since the centroid of $\Delta tuv$ is contained in $\Delta tuv$, the angle $\angle tvb$ is not greater than 180°. Since all other internal angles remain unchanged, all the internal angles of the polygon are not greater than 180°, which means $p'$ is still convex.

**Lemma 1.** Given a convex polygon $p$, by performing smoothing on any of its vertexes, we obtain a polygon $p'$ whose boundary do not self-intersect.

**Proof.** As $p$ is convex, $p'$ is also convex and therefore simple.

**Theorem 2.** Each polygon $p$ asymptotically converges to a single point by repeatedly applying smoothing on $p$.

**Proof.** Let us consider a vertex $u$ of a polygon $\ldots tuv\ldots$. If $u$ is not collinear with $t$ and $v$, by moving $u$ to the centroid $b$ of $\Delta tuv$, the polygon edge $tu$ becomes $tb$ and the polygon edge $uv$ becomes $bv$, both of which are shorter than the original edges (or $Tu + tv > Tb + bv$). If $u$ collinear with $t$ and $v$, the edges computed after smoothing $u$ will have a total length equal to the original edges ($Tu + tv = Tb + bv$). Since not all polygon vertexes $p_i$ can be collinear with the line defined by $p_i-1$ and $p_i+1$, repeated application of smoothing on the polygon results in a sequence of polygons $p, p', p''\ldots$ of strictly decreasing perimeter. Since the polygon edges have positive lengths, the perimeter tends to zero or exactly the same point.

The previous theorems show that the discrete curve shortening preserves non-crossing properties for convex polygons, and that the asymptotic results of the two flows correspond for any polygon. However, the first property does not hold for concave polygons, as shown in Fig. 3a.

By re-sampling the polygon borders the chance of self intersection is reduced, as shown in Fig. 3b. Our experiments verify that this procedure is sufficient for avoiding self-intersections. However, if avoiding self-intersections is strictly required, we can enable used in previous work [5, 20] reduce movement to a safe distance whenever this movement can cause self intersections.

Border re-sampling however provides a second, crucial advantage. When a polygons are over-sampled, refinement of the boundaries is impeded. In fact, a vertex $u$ is very close to neighbours $t$ and $v$, the triangle $\Delta tuv$ is small and therefore its centroid is close to $u$. Therefore, the re-sampling can reduce curve complexity, improving the speed of the computation.

**3.3 Preservation of Diagram Properties**

Discrete curve shortening flow provides a method for curve simplification. However, it cannot be used directly for Euler diagrams improvement without first ensuring that it will preserve fundamental diagram properties such as the presence or absence of set intersections and element containment. First, boundaries should not shrink to a point. As Euler contain set elements in the interior of the curves, we use these elements as a backstop, allowing set boundaries to nicely shrink to fit the elements they contain and preventing the diagram from collapsing to a point. In section 4, we discuss this in further detail.

Secondly, sufficient spacing between set elements as well as between set elements and set boundaries needs to be ensured. Therefore, we propose to model discrete curve shortening flow using a force-directed algorithm, allowing us to use other forces to ensure proper spacing.

Finally, we must prevent set elements from crossing boundaries and ensure that set intersections in the diagram are neither created nor destroyed. This property is accomplished by constrained node movement.

**4 IMPLEMENTATION**

In this section, we describe the implementation of a force-directed algorithm that can be used to improve the appearance of Euler diagrams through discrete curve shortening flow. First, we explain how the input diagram, the output of any Euler diagram drawing method, is encoded into a graph. Then, we present the algorithm itself, EulerSmooth. Finally, we discuss how the algorithm can be configured and its parameters to improve the diagrams presented in this paper.

**4.1 Input Diagram**

The input diagrams are modelled as graphs with polylines edges. These graphs have two sets of nodes: the set elements and the boundary junctions. The set elements have degree zero and have position, colour, size and labels. The junctions identify crossing points between boundaries of the original diagram. These nodes are connected with polylines edges that follow the boundary contours with arbitrary precision. Sets with no boundary crossings are still identified with a singular junction, that is placed in a random point of the set boundary, closing a polyline loop (see Figs. 4a and 4b). For each set, we identify a subgraph formed by all the elements contained in that set, and all junctions and edges that compose its boundary (see Fig. 4c).

This construction ensures that any crossings and concurrent boundaries are modified simultaneously over all sets, preserving the original topology of the graph. We define this condition as dependent boundaries. Whenever this behaviour is considered too restrictive, is it possible relax these constraints and duplicate junctions and shared edges so that the duplicated elements share the same initial position but are free to evolve separately. We define this configuration as independent boundaries (see Fig. 4d). In this paper, all the input diagrams have been constructed by manually tracing over output images, constructing this graph.

**4.2 EulerSmooth**

The proposed algorithm can be considered a modular version a force-directed approach. In the implementation (see Algorithm 1), forces, constraints, and post-iterations steps are modules that can be plugged into or removed from the algorithm, providing a flexible approach.

The input corresponds to the Euler diagram represented as a graph composed of polylines edges. Polylines bends and segments are treated
Fig. 4: The input diagram. (a) The original diagram the algorithm aims to improve. (b) The diagram is modelled as a graph. In green, the junctions. Polyline edges define the set boundaries. (c) For each set, we identify a subgraph containing all graph elements contained by that set. This subfigure shows the subgraph for set C. (d) We can choose to use independent boundaries, allowing for a less constrained refinement. Junctions and edges shared between subgraphs are duplicated (e.g. \( u \) is duplicated in \( u_B \) and \( u_C \)), allowing for separate refinement.

Algorithm 1 EulerSmooth.

\[
\text{for} \ \text{numberOfIterations} \ \text{do} \\
\quad \text{for all} \ v \in V \ \text{do} \\
\hspace{1em} \text{force}(v) \leftarrow 0 \\
\hspace{1em} \text{constraint}(v) \leftarrow +\infty \\
\quad \text{\quad \quad \quad \ // \ \text{FORCE COMPUTATION}} \\
\quad \text{for all} \ \text{ForceDef in ForceSystem} \ \text{do} \\
\hspace{1em} \text{force} \leftarrow \text{force} + \text{ForceDef.compute()} \\
\quad \text{\quad \quad \quad \ // \ \text{CONSTRAINT COMPUTATION}} \\
\quad \text{for all} \ \text{ConstraintDef in ConstraintSystem} \ \text{do} \\
\hspace{1em} \text{constraint} \leftarrow \min (\text{constraint}, \text{ConstraintDef.compute()}) \\
\quad \text{\quad \quad \quad \ // \ \text{NODE MOVEMENT}} \\
\quad \text{for all} \ v \in V \ \text{do} \\
\hspace{1em} \text{if} \ \text{magnitude(force}(u)) > \text{constraint}(u) \ \text{then} \\
\hspace{2em} \text{force}(u) \leftarrow \text{force}(u) + \text{constraint/magnitude(force}(u)) \\
\hspace{2em} \text{position}(u) \leftarrow \text{position}(u) + \text{force}(u) \\
\quad \text{\quad \quad \quad \ // \ \text{POST-ITERATION}} \\
\quad \text{for all} \ \text{PostIteration in PostIterationSteps} \ \text{do} \\
\hspace{1em} \text{PostIteration.compute()} \\
\text{end for} \\
\text{end for} \\
\text{end for} \\
\text{end for} \\
\]

by most modules as if they were standard nodes and edges. However, we support polyline edges to differentiate elements of the drawing that can be added/removed (polyline bends and segments) versus those that must remain in the drawing (nodes and whole edges). All modules use a QuadTree data structure to accelerate the computation time.

In the following, we will discuss the modules these modules, starting with the forces, then the constraints, and finally the post processing steps.

4.2.1 Forces

EulerSmooth allows flexibility in defining the force system to be used. In particular, for all forces it is possible to indicate subsets of the graph elements to which the force is applied. Also, the technique can load multiple instances of the same force, allowing it, for example, to enforce different distances between elements in the drawing according to their role.

In the following, we indicate with \( p_u \in \mathbb{R}^2 \) the position of the node \( u \). If \( e = (u,v) \) is an edge, we indicate the line segment delimited by \( p_u \) and \( p_v \) as \( p_{uv} \).

\begin{itemize}
  \item **EdgeAttraction(\( d \))** This force attracts the extremities of an edge toward each other, decreasing the length of the edge. The parameter \( d \) indicates the ideal distance under which the force starts becoming weak. The attractive force \( \mathcal{F}_u^a \) acting on the node \( u \) of the edge \( e = (u,v) \) is:

\[
\mathcal{F}_u^a(e,d) = \left( \frac{\|p_u - p_v\|}{d} \right) (p_v - p_u)
\]

  \item **NodeNodeRepulsion(\( d \))** This force repels two nodes from each other. The parameter \( d \) indicates the ideal distance under which the force intensifies. As is usual for force-directed algorithms, the attraction and repulsion balance close to \( d \). The repulsive force \( \mathcal{F}_u^r \) is defined as:

\[
\mathcal{F}_u^r(u,v,d) = \left( \frac{d}{\|p_u - p_v\|} \right)^2 (p_u - p_v)
\]

  \item **EdgeNodeRepulsion(\( d \))** This force repels nodes from nearby edges. The parameter \( d \) as a similar meaning than for the force above. This force has been modified from its original formulation in ImPrEd to improve the force stability of node-boundary configurations (Fig. 5).
\end{itemize}
Let $u$ be a node, $e = (v, w)$ be a non-incident edge $(u \notin e)$. Let $p$ be the projection of $u$ on the line defined by $e$. We define $v$ as the closest edge extremity to $u$. The repulsive force $\mathcal{F}_v^r$ is:

$$
\mathcal{F}_v^r(u, v, d) = \begin{cases} 
(p_u - p_p) / \|p_u - p_p\| & \text{if } p \in p_e \\
\mathcal{F}_v^r(u, v, d) & \text{otherwise}
\end{cases}
$$

The respective forces on the segment extremities are:

$$
\mathcal{F}_v^l(u, e, d) = \begin{cases} 
-\mathcal{F}_v^r(u, v, d) / \|p_e - p_w\| & \text{if } p \in p_e \\
-\mathcal{F}_v^r(u, v, d) & \text{if } p \notin p_e
\end{cases}
$$

$$
\mathcal{F}_w^r(u, e, d) = \begin{cases} 
-\mathcal{F}_v^l(u, e, d) / \|p_u - p_v\| & \text{if } p \in p_e \\
0 & \text{if } p \notin p_e
\end{cases}
$$

The factor introduced in the first cases intensify the repulsive force on the closest edge extremity and decreases it on the farther one, better approximating the behaviour of an analogue physical system (see Fig. 5b).

**CurveSmoothing** This force implements the smoothing effect of the discrete curve shortening flow. The force moves a node toward the centroid of the triangle formed with its neighbours. Therefore, the $\mathcal{F}_u^s$ acts on node $u$ with neighbours $t$ and $v$ is:

$$
\mathcal{F}_u^s = \frac{2}{3} \left( \frac{p_t + p_v}{2} - p_u \right)
$$

**4.2.2 Constraints**

These modules can be used to control node movements or avoid movements that could compromise the properties of the drawing that we want to preserve. In this section, we describe these constraints.

**DecreasingMaxMovement($d$)** This constraint sets a maximal movement for all nodes which starts at a value of $d$ and linearily decreases to 0 as the computation progresses, allowing for a precise final positioning.

**MovementAcceleration($d$)** This constraint influences movement consistency in the algorithm. For each node $v$, a constraint $c_i(u)$ is updated according to the angle $a$ between consecutive iterations:

$$
c_i(u) = \begin{cases} 
d & \text{if } i = 0 \\
c_{i-1} \left(1 + 2 \left(1 - \frac{d}{d_i} \right) \right) & \text{if } a < 60^\circ \\
c_{i-1} & \text{if } 60^\circ \leq a < 90^\circ \\
c_{i-1} / \left(1 + 4 \left(\frac{a}{90^\circ} - 1\right) \right) & \text{if } a \geq 90^\circ
\end{cases}
$$

In other words, the constraint is multiplied by a factor in the range $[1,3]$ for movement in the same direction and divided by $[1,5]$ for movement in opposite directions. This greatly reduces node oscillation.

**PinnedNodes** This module that set to 0 the movement constraint of a given set of nodes, forcing them to keep their original position.

**SurroundingEdges** Given a set nodes $V$ and a set of edges $E$, the module ensures no $u \in V$ can cross any edge $e \in E$. For a given $u$ and $e = (v, w)$, the module identifies a line $l$ that divides the plane into two halves. If the projection $p$ of $u$ in lies in the edge $(p \in p_e)$, the line $l$ is identified as the axis of symmetry of the segment $uv$. If the projection lies outside the edge, $l$ is the axis of symmetry of the segment $uw$, with $v$ being the closest endpoint to $u$. This module ensures that $u$ and $e$ by limiting their movement to the collision distance between node and $l$ along the direction of movement (see Fig. 6).

**4.2.3 Post-Iteration Steps**

Post-iteration modules allow for the processing of the output of each iteration.

**FlexibleEdges($d_e, d_c$)** Provides the flexible edge functionality of ImPrEd. Given an edge set $E$, the module increases or decreases the number of bends depending on the stress of its edge segments. Given an edge $e$, the module first tries to contract the edge, by removing bends, and then it tries to expand it by adding them. When contracting an edge, the module marks $b_1 \ldots b_n$ for removal. The bend $b_i$ is removed if the distance between its previous bend $b_{i-1}$ (edge source, if $i = 1$) and the next bend $b_{i+1}$ (edge target, if $i = n$) is less than $d_c$ and no node is contained in the triangle formed by the three bends (see Fig. 7a). When expanding an edge, the segments $e_1 \ldots e_n$ are checked for expansion. A segment is expanded when its length is greater than $d_e$ by inserting a new bend at its midpoint (see Fig. 7b).

**4.3 Algorithm Configuration**

Despite its modular nature, configuring EulerSmooth is simple. In order to ensure comparable results, we run the same algorithm configuration for all the diagrams in this paper. This requires a single distance parameter $d'$ and setting three boolean variables. The parameter $d'$ can be thought of as an ideal distance between set elements and between set elements and boundaries. This parameter is proportional to the scale of the imported diagram.

The core modules, loaded for the improvement of each diagram, are:
**CurveSmoothing**

- **EdgeNodeRepulsion**($d^*$) between elements and boundary edges
- **EdgeAttraction**(0.3 $d^*$) for boundary edges
- **DecreasingMaxMovement**(d$^*$)
- **MovementAcceleration**(d$^*$)
- **FlexibleEdges**(1.45 $d^*$, 1.5 $d^*$)

The curve smoothing module drives the boundary improvement and all other modules are defined as above.

The first boolean indicates if dependent or independent set boundaries are required (see Section 4.1). We denote **Dep** to indicate dependent boundaries and **Ind** to indicate independent boundaries:

- **Dep** then
  - **SurroundingEdges** between all nodes and boundaries.
- **else**
  - **SurroundingEdges** between elements and boundaries.

Through surrounding edges, we prevent set elements and boundaries from crossing and ensure that set elements do not leave the set. As this behaviour can be turned on or off for boundary nodes, when off it will allow for the creation or elimination of new zones that are not in the original diagram. The general result is a smoother diagram, where some new empty zones can be created. However, if this behaviour is not desired and the precise topology of the diagram needs to be preserved, surrounding edges can be applied to all nodes in the drawing, including boundary nodes, ensuring that no new zone is created or destroyed by the procedure.

The second boolean option pins element movement. We use **Fix** to indicate fixed position and **Mov** to indicate free moving elements:

- **Fix** then
  - **PinnedNodes** on set elements
- **else**
  - **NodeNodeRepulsion**(d$^*$) between set elements

Pinning elements is particularly useful when refining drawings such as those produced by Bubble Sets [8]. Otherwise, forces are applied to set elements in order to optimize their distribution.

The final boolean allows concurrency between set boundaries to be turned on and off. We use **Sep** to indicate the activation of this option:

- **Sep** then
  - **EdgeNodeRepulsion**(d$^*$ / 15) between boundary nodes and edges

When set, a small radius repulsive force between nodes and unrelated boundary edges is inserted, creating a separation between collinear boundaries. This option improves the readability of diagram, but could generate unnecessary stress and boundary irregularity on diagrams without concurrent boundaries.

For all images presented in this paper, we specify the options set to centre which might make the original diagram less readable. We eliminate an unnecessary empty zones (blue-only region in the top-left corner) by using independent boundaries.

**5 Analysis**

To evaluate the effectiveness of **EulerSmooth**, we extracted diagrams published in previous work and tried to improve them. The initial diagrams have been created by tracing contours and adding set elements as prescribed by the output images. Then, these diagrams have been optimized using appropriate parameters for a given case. For example, when improving the Manhattan Bubble Sets diagram (see Figure 13), we use fixed node positions as they are prescribed by the locations on the map. Whenever applicable, we improved the diagram using options, to showcase the many outputs available with **EulerSmooth**.

**Contour Based Diagrams** Euler diagrams drawing approaches might or might not show set elements, causing significant differences in how a diagram is created and interpreted. For example, in a diagram that does not represent set elements (contour based), the existence of an overlap in the diagram implies that the given intersection contains elements. On the other hand, requirement is not strict for element based approaches, since elements present in the region determine if it is empty or not.

**5.1 Qualitative Evaluation**

Figure 8 shows the results of **EulerSmooth** on two basic diagrams: one with a single set and a second with two overlapping sets. We notice how the contour regularity is improved both when enabling or disabling element movement, but **EulerSmooth** achieves a better result when the nodes can be moved into more desirable positions.

Figure 9 shows an improved version of a diagram generated with **Set Visualiser** [30]. By using independent boundaries, we can eliminate an unnecessary empty zones (blue-only region in the top-centre) which might make the original diagram less readable. We notice little difference between the results obtained with fixed versus
However, the shapes obtained with EulerForce look more regular as quantitative analysis can confirm. Nevertheless, these new regions will contain elements and the boundaries are much easier to trace. However, none of these new regions will contain elements and the boundaries are much easier to trace. Since that point must assume a position that must satisfy three different curves, we obtain unpleasant sharp angles for two boundaries (blue and green).

If preserving the exact topology of the initial diagram is less of a priority, we enable independent boundaries and obtain the results in Figure 10c. By enabling this option, we introduce new regions not present in the original drawing (in Figure 10c, see for instance blue–only triangular region in the centre of the drawing). However, none of these new regions will contain elements and the boundaries are much easier to trace.

Figure 12 shows the same input diagram improved with EulerSmooth and EulerForce. The diagrams improved with EulerForce look more regular as quantitative analysis can confirm. However, the shapes obtained with EulerSmooth are smooth and readable.

Figure 13 shows application of our method to the Manhattan Bubble Sets [8, Figure 9]. Although the obtained sets are far from circles, EulerSmooth can simplify boundaries even in these constrained circumstances, improving the ability to trace these contours.

Finally, in Figure 11, EulerSmooth is applied to an Untangled Euler diagram [17, Figure 1a], transforming it into a more classical looking one (whenever this is desired) with regular boundaries.

### 5.2 Quantitative Evaluation

We evaluated the effectiveness of EulerSmooth in smoothing the set boundaries by computing the average initial and final isoperimetric quotient for the diagram sets. Given a closed curve \( \ell \), the isoperimetric quotient computes the ratio between the area enclosed by the line and that of a circle with equal circumference: \( Q = \frac{\text{area}(\ell)}{\text{length}(\ell)} \).

Since circles are the shape that maximize closed area for a given perimeter, all simple closed curves will have an isoperimetric quotient between 0 and 1. This value is computed for each set in the diagram and averaged.

**Improvements over Various Techniques** Table 1 reports options used, running time, and initial and final average isoperimetric quotient for the diagrams in this paper. We notice that our approach improves \( Q \) over all cases with larger improvements when less restrictive options are enabled (Mov instead of Fix, Ind instead of Dep). A low smoothing value is obtained for Bubble Sets (Figure 13), which is a heavily constrained diagram that is difficult to improve.
Fig. 13: Improvement of Bubble Sets diagram over the Manhattan map [8, Figure 9]. (a) The original diagram reconstructed. (b) The diagram optimised with EulerSmooth (Fix, Ind).

### Table 1: Statistics for the execution of EulerSmooth over the diagrams in this paper. The statistics for EulerForce diagrams (Figures 12 and 14) are collected in Table 2. Constraints activated are indicated by their first letter (F for Fix, M for Mov, D for Dep, I for Ind). Iterations shows the number of smoothing cycles performed. $Q_i$ is the initial average isoperimetric quotient. $Q_s$ is the average isoperimetric quotient for the improved diagram. The running time is the average running time over five runs of the approach.

<table>
<thead>
<tr>
<th>Diagram</th>
<th>Fig.</th>
<th>Opt.</th>
<th>Iter.</th>
<th>$Q_i$</th>
<th>$Q_s$</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Euler</td>
<td>1b</td>
<td>MD</td>
<td>300</td>
<td>75.3</td>
<td>96.5</td>
<td>3.35</td>
</tr>
<tr>
<td>Untangled Small</td>
<td>1d</td>
<td>MI</td>
<td>100</td>
<td>36.7</td>
<td>76.3</td>
<td>2.42</td>
</tr>
<tr>
<td>Single</td>
<td>8b</td>
<td>FD</td>
<td>120</td>
<td>39.7</td>
<td>90.3</td>
<td>1.83</td>
</tr>
<tr>
<td></td>
<td>8c</td>
<td>MD</td>
<td></td>
<td></td>
<td>99.6</td>
<td>1.96</td>
</tr>
<tr>
<td>Double</td>
<td>8e</td>
<td>FD</td>
<td>120</td>
<td>45.7</td>
<td>85.3</td>
<td>1.88</td>
</tr>
<tr>
<td></td>
<td>8f</td>
<td>MD</td>
<td></td>
<td></td>
<td>98.5</td>
<td>2.07</td>
</tr>
<tr>
<td>Imdb20</td>
<td>—</td>
<td>—</td>
<td>25</td>
<td>56.9</td>
<td>74.6</td>
<td>6.52</td>
</tr>
<tr>
<td></td>
<td>—</td>
<td>FI</td>
<td>40</td>
<td></td>
<td>84.6</td>
<td>9.67</td>
</tr>
<tr>
<td></td>
<td>10b</td>
<td>MD</td>
<td>25</td>
<td></td>
<td>73.9</td>
<td>6.71</td>
</tr>
<tr>
<td></td>
<td>10c</td>
<td>MI</td>
<td>40</td>
<td></td>
<td>86.0</td>
<td>10.15</td>
</tr>
<tr>
<td>BubbleSet</td>
<td>13b</td>
<td>FI</td>
<td>50</td>
<td>4.5</td>
<td>10.2</td>
<td>6.43</td>
</tr>
<tr>
<td>Untangled</td>
<td>11b</td>
<td>FI</td>
<td>100</td>
<td>47.1</td>
<td>79.2</td>
<td>5.25</td>
</tr>
<tr>
<td></td>
<td>11c</td>
<td>MI</td>
<td></td>
<td></td>
<td>82.8</td>
<td>5.21</td>
</tr>
<tr>
<td>SetVis</td>
<td>9b</td>
<td>FI</td>
<td>50</td>
<td>77.3</td>
<td>92.3</td>
<td>1.22</td>
</tr>
<tr>
<td></td>
<td>9c</td>
<td>MI</td>
<td></td>
<td></td>
<td>93.3</td>
<td>1.19</td>
</tr>
</tbody>
</table>

### Table 2: Statistics for the improvement of diagrams with EulerSmooth and EulerForce. $Q_i$ is the average isoperimetric quotient. Subscript $i$ indicates initial, $s$ indicates EulerSmooth, $f$ indicates EulerForce. $Q_f$ has not been computed for the three 5-sets instances (ID 1, 4 and 5) as EulerForce produced invalid diagrams. On all diagrams, we run EulerSmooth with the option Mov, Dep and Sep. For 3 and 4 sets, 350 iterations were sufficient. For 5 sets, we ran 500 iterations.

<table>
<thead>
<tr>
<th>Sets</th>
<th>Id</th>
<th>$Q_i$</th>
<th>$Q_s$</th>
<th>$Q_f$</th>
<th>Time $s$</th>
<th>Time $f$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>58.3</td>
<td>95.5</td>
<td>99.0</td>
<td>3.6</td>
<td>2.35</td>
</tr>
<tr>
<td>2</td>
<td>57.0</td>
<td>95.6</td>
<td>98.5</td>
<td>3.5</td>
<td>3.48</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>51.3</td>
<td>96.1</td>
<td>98.6</td>
<td>3.6</td>
<td>2.25</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>51.0</td>
<td>96.2</td>
<td>97.4</td>
<td>4.6</td>
<td>2.57</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>58.5</td>
<td>95.5</td>
<td>98.8</td>
<td>5.9</td>
<td>2.57</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>53.6</td>
<td>96.2</td>
<td>98.1</td>
<td>5.4</td>
<td>4.85</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>38.4</td>
<td>93.9</td>
<td>94.0</td>
<td>5.7</td>
<td>5.85</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>45.5</td>
<td>94.9</td>
<td>97.2</td>
<td>6.2</td>
<td>4.63</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>41.9</td>
<td>93.3</td>
<td>93.0</td>
<td>4.5</td>
<td>6.14</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>53.6</td>
<td>95.6</td>
<td>98.5</td>
<td>5.1</td>
<td>5.85</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>36.5</td>
<td>84.9</td>
<td>Inv</td>
<td>8.6</td>
<td>32.89</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>41.7</td>
<td>95.2</td>
<td>98.2</td>
<td>8.2</td>
<td>16.64</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>43.0</td>
<td>95.6</td>
<td>95.0</td>
<td>8.2</td>
<td>14.53</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>44.3</td>
<td>86.2</td>
<td>Inv</td>
<td>13.9</td>
<td>25.67</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>42.6</td>
<td>85.6</td>
<td>Inv</td>
<td>13.5</td>
<td>22.47</td>
<td></td>
</tr>
</tbody>
</table>
Running times are averaged over five executions and are in the range of one to a dozen of seconds. Times of this magnitude are acceptable for an interactive environment, as long as the data is only updated every few seconds. As expected, larger diagrams and independent with boundaries are generally responsible for increased computation times. However, relatively simple diagrams might still require a relatively high number of iterations. When observing the smoothing over time, we notice that boundary shapes are rapidly smoothed, but that a large amount of time is used to shrink the contours to snugly fit the set elements. In fact, the shrinking process is slower on regular shapes, which are quickly achieved in small and simple instances.

**Comparison with EulerForce** We run a more in-depth comparison between EulerSmooth and EulerForce. The authors of EulerForce provide a software that can generate an initial random diagram with 3, 4 or 5 sets that can be improved. We generate five diagrams for each number of sets (a total of 15 diagrams) and optimize them using EulerSmooth and EulerForce. We compare the resulting average isoperimetric quotient and recorded the average running time over the executions for both algorithms. Table 2 reports this data.

The computed metrics indicate a higher set boundary regularity for EulerForce, as suggested by visual inspection. The values are high and about the same for both approaches, indicating that all output diagrams are of high quality. However, EulerForce does not always produce a valid diagram. It produces an invalid configuration, with introduced and removed zones, for three out of five instances of the five set data set. The authors explain that the increase in diagram complexity increases probability of obtaining invalid configurations, resulting in a success rate of 61% for five sets. As seen in this paper, we can scale to diagrams of twenty sets.

EulerSmooth, due to its simple force system, can generate correct and pleasant diagrams even for the instances not supported every time (see Figure 14). Also, the running time of EulerSmooth is below that of EulerForce around four to five sets, indicating that it can scale to larger diagrams.

### 6 Conclusion

In this paper, we present a simple approach for Euler diagram improvement based on a simplification of curve shortening flow. We translate this method into a force system that can be used to improve the output of any Euler diagram drawing method using polygonal curves and set elements. The approach has been demonstrated on a variety of methods [21, 8, 17, 14] for drawing Euler diagrams and evaluated against competitive approaches.

As future work, it would be interesting to see if we can extend other aspects of curve shortening flow and vector field design to the improvement of Euler diagrams in general. As methods exist that do not use a force system to optimize the input shape, it may be beneficial, in terms of computation time, to use these methods instead of the force system to optimize Euler diagrams.

**References**


