CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Kent Academic Repository

Drawing Graphs in Euler Diagrams

Paul Mutton[1], Peter Rodgers[1], and Jean Flower[2]

[1] Computing Laboratory, University of Kent, Canterbury, Kent, CT2 7NF, UK
{pjm2, P.J.Rodgers}@kent.ac.uk
http://www.cs.kent.ac.uk/people/rpg/pjm2/
[2] School of Computing, Mathematical and Information Sciences,
Watts Building, University of Brighton,
Lewes Road, Brighton, BN2 4GJ, UK
J.A.Flower@bton.ac.uk
http://www.cmis.brighton.ac.uk/research/vmg/

Abstract. We describe a method for drawing graph-enhanced Euler diagrams using a three stage method. The first stage is to lay out the underlying Euler diagram using a multicriteria optimizing system. The second stage is to find suitable locations for nodes in the zones of the Euler diagram using a force based method. The third stage is to minimize edge crossings and total edge length by swapping the location of nodes that are in the same zone with a multicriteria hill climbing method. We show a working version of the software that draws spider diagrams. Spider diagrams represent logical expressions by superimposing graphs upon an Euler diagram. This application requires an extra step in the drawing process because the embedded graphs only convey information about the connectedness of nodes and so a spanning tree must be chosen for each maximally connected component. Similar notations to Euler diagrams enhanced with graphs are common in many applications and our method is generalizable to drawing Hypergraphs represented in the subset standard, or to drawing Higraphs where edges are restricted to connecting with only atomic nodes.

1 Introduction

The system described here links graph drawing and Euler diagram drawing into a system for drawing graph-enhanced Euler diagrams. In a graph-enhanced Euler diagram, we have a graph, an underlying Euler diagram, and a mapping from the graph nodes to the zones of the Euler diagram. In any drawing, the nodes are required to be included in the corresponding zone. We are, in effect, embedding graphs in Euler diagrams. Our approach is to draw the Euler diagram first, and later add the graph in a way that minimizes the number of edge crossings and total edge length in the graph.

There are various application areas which can be visualized by such structures and so benefit from the work described here such as databases [3] and file system organization [2]. However, we show our system being used with a form of constraint diagram, the spider diagram [9]. This application area is in particular need of automatic layout for the diagrams because automatic reasoning algorithms produce abstract diagrams that have no physical layout.

An Euler diagram is a collection of *contours* (drawn as simple closed curves), arranged with specific overlaps. The parts of the plane distinguished by being contained within some contours and excluded from other contours are called *zones*. The essential structure of an Euler diagram is encapsulated by an *abstract Euler diagram*. An abstract Euler diagram is made up of information about contours and zones. Contours at the abstract level are not drawn, but have distinguishing *contour labels*. Zones are not parts of the plane, but a partition of the contour set into containing contours and excluding contours. To clarify these concepts, Figure 1 shows, first, an abstract Euler diagram, and, second, a drawn representation of the same Euler diagram. The shaded zone in the drawn diagram corresponds to the abstract zone $\{a\}$.

Contours : $\{ a, b \}$
Zones: $\{\{\}, \{a\}, \{b\}\}$

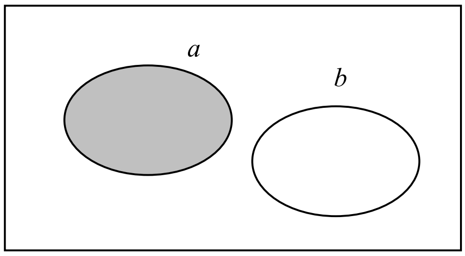


Fig. 1. The distinction between an abstract Euler diagram and a corresponding drawn Euler diagram.

Most graph drawing systems do not take account of regional constraints, where nodes must be contained within complex shapes. Simulated annealing can be an effective method of drawing graphs using a set of simple criteria [21]. These criteria are used to judge the aesthetic quality of the resulting layout. Each criterion can be weighted to change its importance. Such systems are applicable to embedding graphs in Euler diagrams when used with suitable aesthetic criteria.

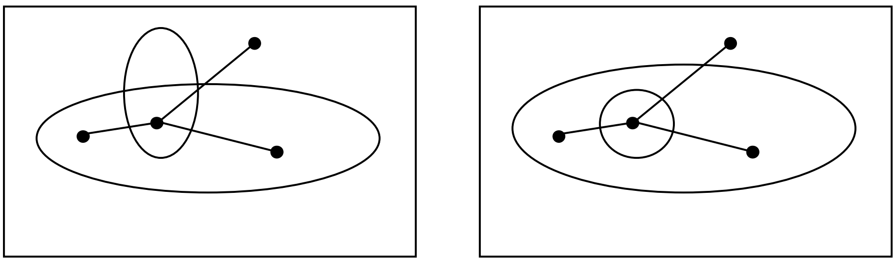


Fig. 2. Two equivalent hypergraph drawings which are different when interpreted as Euler diagrams.

In graph-enhanced Euler diagrams, the absence of a zone from the second diagram in Figure 2 would convey extra information, whereas, considered as hypergraphs, these two Figures convey the same information

Inspired by the widespread use of diagrammatic notations for modeling and specifying software systems, there has been much work recently about giving diagrammatic notations formal semantics. The analysis of a diagrammatic specification can be done using diagrammatic reasoning rules - rules to transform one diagrammatic assertion into a new diagram that represents equivalent or a weaker semantic statement.

One such notation, and reasoning system, is that of *constraint diagrams* [16,5,19]. A simple subset of constraint diagrams, with a restricted notation and restricted rule system, is that of spider diagrams. *Unitary spider diagrams* are Euler diagrams with extra notation comprising shading in zones and a graph superimposed on the diagram. The components of the superimposed graph are trees (called spiders). Contours represent sets and zones represent subsets of those sets, built from intersection and exclusion. The absence of a zone from the diagram indicates that the set corresponding to that zone is empty. Thus the absence of a zone from the diagram conveys information, and the two diagrams in Figure 2 have different semantics.

Each spider drawn on the diagram has a *habitat*: the collection of zones that contain nodes of the graph. The spiders assert semantically the existence of an element in the set corresponding to its habitat. Spiders place lower bounds on the cardinality of sets. Shading in a zone (or collection of zones) indicates that the set corresponding to that zone (or zones) contains only elements for the spiders that are in it, and no more. Shading places an upper limit on the cardinality of sets. See Figure 3 for an example of a spider diagram.

Contours : $\{ a, b \}$
Zones: $\{\{\},\{a\},\{b\}\}$
Shading: $\{\{a\}\}$
Spiders : $\{\{\{\},\{b\}\},\{\{a\}\}\}$

Semantics:
$|A| = 1$ and $A \cap B = \{\}$ and $|U\text{-}A| \geq 1$

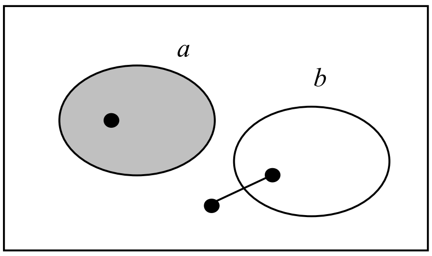


Fig. 3. An abstract spider diagram and a corresponding drawn spider diagram.

The semantics of spider diagrams provide a foundation upon which we build reasoning rules. In the case of spider diagrams, there are seven rules which transform a spider diagram into another. For example, one rule transforms a diagram with an absent zone into the equivalent diagram which contains the zone, shaded. This reasoning rule changes the structure of the underlying Euler diagram and necessitates reconstruction of a drawn diagram. A sequence of reasoning rules, applied to a premise diagram, gives a proof which ends with a conclusion diagram. An example of such a proof is shown, drawn by hand, in Figure 4. The same proof is shown again later in Figures 14-16.

The seven reasoning rules each make a small change to a diagram, and they have been proven to be sound [20,23]. If a rule transforms diagram d_1 into diagram d_2 then d_2 represents a semantic consequence of d_1. Other rules could be devised which are sound, and in any logic system, the choice of rules is to some extent arbitrary. But these rules form a logically complete set.

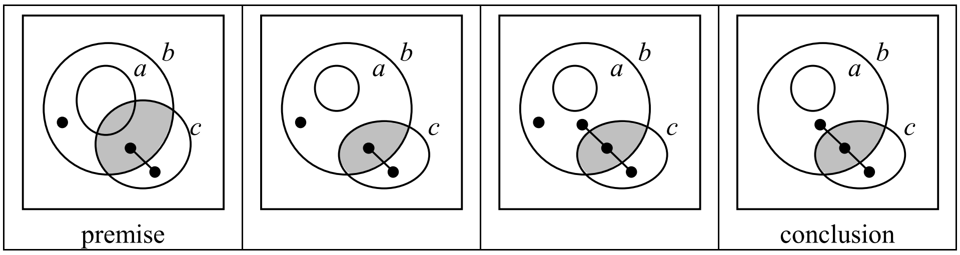


Fig. 4. An example of a proof in the spider diagram reasoning system

The full spider diagram reasoning system allows for the manipulation and interpretation of compound spider diagrams: that is, expressions built up from spider diagrams using the propositional logic connectives "and" and "or". This extension leads to many more reasoning rules, giving a sound and complete reasoning system, equivalent in its expressiveness to monadic first order predicate logic with equality. More details on the system, its rules and its expressiveness can be found in [10,18].

We have developed a tool [9,10] to assist users with the application of reasoning rules to transform diagrams. At the heart of this must be an algorithm to generate diagrams for presentation to the user as the outcome of a rule application.

2 Related Work

The task of drawing an Euler diagram - taking an abstract diagram and producing a corresponding drawn Euler diagram is analogous to the task of graph drawing. Previous research has addressed some initial issues concerning the drawing of Euler diagrams. The paper [7] outlined well-formedness conditions on drawn diagrams and presented an algorithm to identify whether an abstract diagram was drawable subject to those conditions. If a diagram was diagnosed as drawable, then a drawing was produced. Later work, [8], sought to enhance the layout of a drawn Euler diagram using a hill-climbing approach in combination with a range of layout metrics to assess the quality of a drawing. There exists an Euler diagram drawing system [22] that embeds some small diagrams, which can be drawn with a limited subset of shapes.

There has been some previous work in drawing extended graph systems. Clustered graph visualization systems are common (e.g. [4,14]), but in such structures the regions only nest and cannot intersect, hence they are not as expressive as Euler diagrams. There are a limited number of drawing methods for more complex graph-like structures such as hypergraphs and higraphs. Hypergraphs are similar to standard graphs, but with hyperedges rather than edges. Hyperedges connect to several nodes, in contrast with standard edges which connect at most two nodes. Hypergraphs are commonly represented in two ways: by the edge standard and the subset standard [17]. The edge standard draws hyperedges as lines, effectively adding a dummy node for each hyperedge, where the lines connecting to each node meet. The subset standard is a representation closer to enhanced Euler diagrams, where the hyperedges are indicated by closed curves surrounding the grouped nodes. However, there are still significant differences as hypergraph closed curves that intersect have no extra meaning, and current hypergraph drawing methods [1] emphasize node groupings, putting little emphasis on the layout of the curves. Hypergraphs with binary edges are represented with the edge standard and with non binary edges represented with the subset standard are similar to commonly applied subsets of higraphs [13,15].

3 Drawing Euler Diagrams Enhanced With Graphs

In this section we describe our three stage generic method for laying out graphs in Euler diagrams. The software system has been implemented in Java.

3.1 Stage 1: Euler Diagram Smoothing

The basic process of drawing Euler diagrams in stage 1 has been detailed previously [8]. In outline, firstly we produce an initial diagram based on the zone specification as described in [7]. This results in a structurally correct, but not very well laid out diagram. We then apply a multicriteria optimizer, which attempts to improve a weighted sum of various diagram layout criteria using a hill climbing method. This adjusts the contours by both moving them and moving the individual points of the polygons that are used to represent them. It assesses the layout formed on each single move for the presence of the correct zones and to see if the change has improved the weighted sum. We use several criteria for measuring diagram features, such as contour smoothness, contour size, zone area and contour closeness. The criteria and the hill climber are described in [8].

This system has since been extended to deal with nested diagrams [11]. Nested Euler diagrams have subdiagrams entirely enclosed in a zone of a containing diagram. To draw a nested diagram, assuming we have a mechanism for drawing each atomic (non-nested) part independently, the first step is to identify, in the abstract diagram, which are the atomic components and which zones of containing diagrams each nested part belongs to. Each atomic component can be drawn and this tree-structure of drawn atomic components is combined into a single diagram as follows. For each zone which contains sub-diagrams, we find its bounding box, split it into a $j \times j$ grid and consider sequences of sub-boxes, width i, within the bounding box. The sub-boxes occupy a fraction i / j of the bounding box, and are placed sequentially at $(j - i)^2$ positions scanning the whole bounding box (starting centrally). As j gets larger, the subboxes shrink and eventually one will be found which fits inside the zone. This sub-box is partitioned into disjoint boxes, within which the nested diagrams are inserted. This process is illustrated in Figure 5.

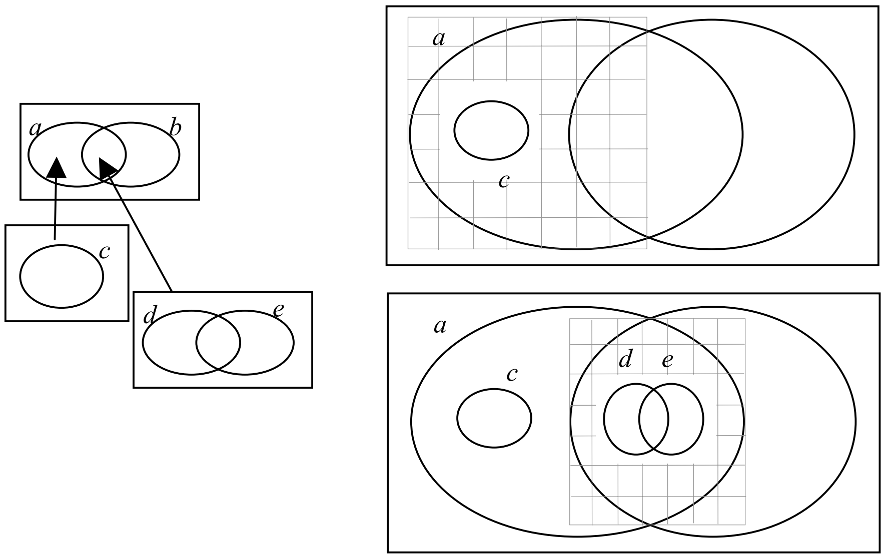


Fig. 5. Nesting Euler diagrams.

Once the nested diagram has been built in this way, the next step is to improve its appearance by smoothing. As the nesting can be arbitrarily deep, the amount of movement of polygons and polygon corners could be too large for very small nested contours. Hence, the amount of movement has been scaled to be proportional to the size of the contour (in fact, the bounding box of the contour) against the size of the whole diagram.

The result of Stage 1 is normally a well laid out Euler diagram. The graph can then be superimposed as described in the following sections.

3.2 Stage 2: Finding Locations for Nodes

A node belonging to a particular zone must be placed such that the node is contained within the region defined by the drawn zone. Each concrete zone is defined by a sequence of line segments. We do not concern ourselves with disconnected zone areas, as these are not present in a well-formed [7] Euler diagram, however, for nested diagrams, at least one zone fails to be simply-connected (i.e. it’s ring-shaped, or worse; see Figure 6). Zones which are simply connected (i.e. disc-like) have one polygon as their boundary, but non-simply connected zones have multiple polygons bounding them. In topology, a set is defined to be simply-connected if any path which starts and finishes at the same point can be continuously deformed until the path is constant at a point. A zone is simply connected if it is isotopic to a disc.

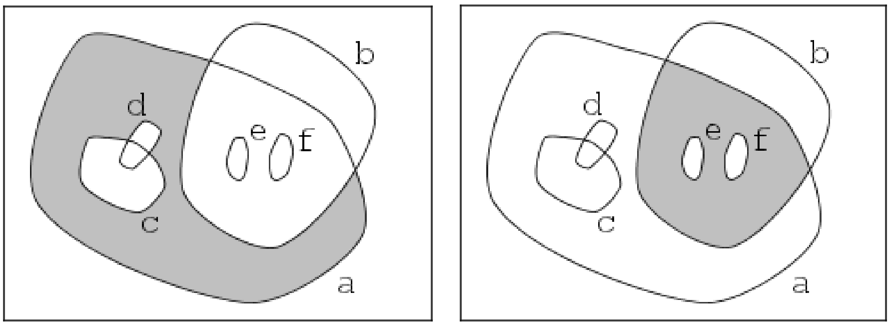


Fig. 6. Two examples of non-simply-connected zones (shaded), drawn by our implementation.

A variety of possible strategies exist for the initial placement of a node inside its containing zone. We use a fast and simple method that is primarily concerned with ensuring that the node is contained inside the zone, regardless of how bad that placement is. Subsequent application of a force model refines the placement so that the node is not too close to any of the boundaries of the zone. The force model also ensures that all nodes sharing the same zone are reasonably spaced.

The initial placement of a node requires a line to be drawn through the containing zone. For simplicity of implementation, this line is horizontal and passes through the bounding box of the concrete zone. The y-coordinate of the horizontal line is chosen randomly between the range of the bounding box in order to give a scattering effect when there is more than one node present in a zone. By intersecting the bounding box horizontally, we can be certain that there is at least one subinterval of the line that is contained by the area of the concrete zone.

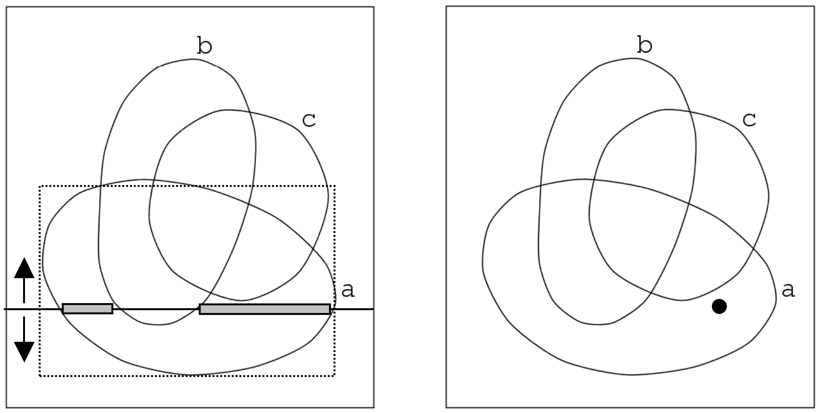


Fig. 7. Candidate locations for a new node in zone *a* excluding *b*,*c*. The horizontal line is placed such that it intersects the bounding box of zone *a* at a random height. This diagram shows two subintervals where it is valid to place the new node.

An ordered set is built up from the intersection points of the horizontal line and the line segments which make up the boundary of the zone. This set must contain at least two points, and any location between the $2n$-1^{th} and $2n^{th}$ intersection point must belong to the zone (see Figure 7).

The Stage 1 method for placing nested diagrams described in Section 2.1 could have been used for the initial placement of nodes. However this node placement method is faster as we are placing a point rather than a shape with a bounding area and we are unconcerned about a central placing of the point, anticipating the refinement which is described next.

After initial placement, refinement of node locations is achieved by applying a force model to the set M of nodes in the zone. We introduce a repulsive force acting between each pair of nodes in the zone, causing them to become evenly distributed. This repulsive force is inversely proportional to the separation d, and proportional to the number of nodes, $|M|$, in the zone. A constant c is used to affect the desired separation between pairs of nodes. This repulsive force is based on the force model by Fruchterman and Reingold [12] and is commonly used in force directed graph layout.

$$\text{Repulsive force between two nodes} = |M| \times \frac{c}{d}.$$

To prevent nodes escaping from a zone or getting undesirably close to the boundary of a zone, we make each line segment in the boundary of the zone exert a repulsive force on each contained node. It is desirable to let the set of nodes spread about a reasonably large area of the zone, however it is still essential to keep each node away from the line segments that define the zone. For this reason, we depart from the previously used force model and make the repulsive force acting on a node proportional to the inverse square of the distance from the line segment. This encourages nodes to spread over a reasonable area with very little chance of getting too close to a boundary due to the prohibitively high resultant forces.

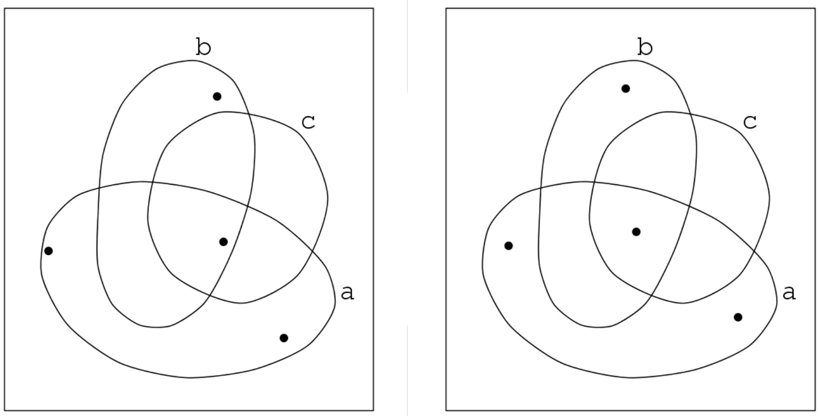


Fig. 8. Initial placement of spider feet (left) and refinement under the force model (right).

The repulsive force is proportional to $|M|^2$, as this helps to contain larger sets of nodes where there will be more node-node repulsions. As the zone may consist of an arbitrary number of line segments of arbitrary lengths, the repulsive force is also proportional to each length.

$$\text{Repulsive force between a line segment and a node} = |M|^2 \times \frac{lc}{d^2}\,.$$

We have observed that better results can be obtained when there are more line segments bounding a zone. We use a method that breaks a zone boundary into more line segments without affecting the region contained; typically so there become more than a hundred new line segments. We use the simple method of dividing each existing line segment into two new line segments of equal length. The process is repeated until it yields enough new line segments. This reduces the chance of a node escaping from a corner of the zone when the force model is applied.

The simulation of the force model is an iterative process. For each iteration, the resultant force acting on each node is the sum of all repulsive forces from the line segments of the containing zone and the repulsive forces from all other nodes in the same zone. After calculating all of the resultant forces, the location of each node is updated by moving it a small distance in the direction of the force. The distance of the movement is proportional to the magnitude of the force. After a number of iterations, the system nears an equilibrium and the nodes occupy their new locations.

3.3 Stage 3: Laying Out the Edges

The previous stage calculates locations for nodes. We can think of these locations as being candidate locations for the set of nodes in the zone, and we are free to swap the location of pairs of nodes, within a zone, without changing the meaning of the diagram (see Figure 9). Swapping pairs of nodes changes the location of edges emanating from those nodes. We use a simple hill climbing approach on this with two metrics to improve the quality of the diagram.

One desirable feature of a diagram is to have a minimal number of edge crossings. Our first metric returns the number of edge crossings in the current diagram, so values closer to zero will represent a better quality of layout in terms of edge crossings. To further enhance the understandability of the diagram, we introduce a second metric, which is based on the length of edges in the diagram. Shorter edges make graphs easier to navigate and identify, so the value returned by this metric will represent an improvement in the layout if the value is closer to zero. The value returned is the sum of each edge length squared.

In our current system, we are only concerned with simple straight-line edges, although it is worth noting that our software can deal with non-simple edges. For example, some notations use curves or shapes to represent special edges and our system is able to detect intersections with these nonlinear edges.

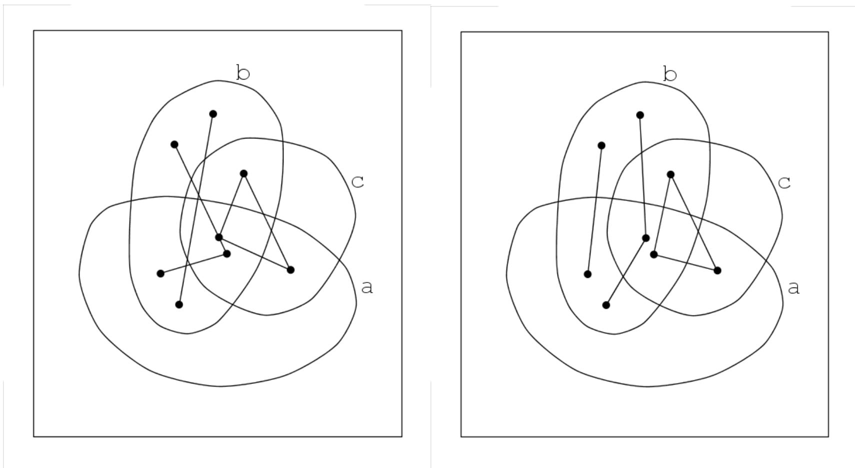


Fig. 9. A diagram with 4 edge crossings (left) and the same diagram produced using the hill climber, with no edge crossings (right). Notice the common locations for all nodes. The right hand diagram has had 3 pairs of nodes swapped, in zones *b*, *ab* and *abc*.

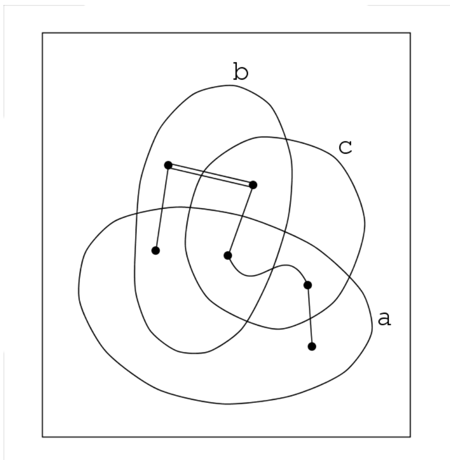
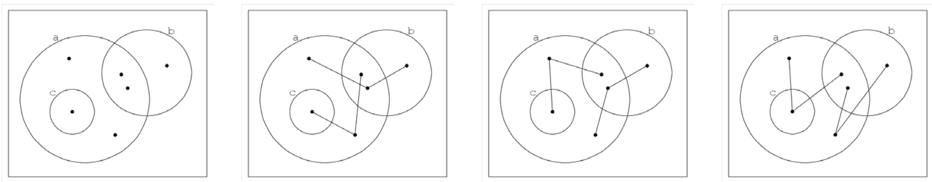


Fig. 10. A diagram demonstrating the different types of edges that are supported by our system. Intersections with the more complicated types of edges can still be computed.

As the value returned by the edge length metric is based on the sum of edge lengths squared in the diagram, we make this value dimensionless by dividing it by the area of the diagram. This makes the metric return the same value for a particular diagram, regardless of the scaling.

The two metrics are combined as a weighted sum to work out the current quality of a diagram. As we have determined minimization of edge crossings to be the most important factor, we apply a much higher weighting to this metric. That is, we are unlikely to reduce the total edge length in a diagram at the expense of introducing a new edge crossing.

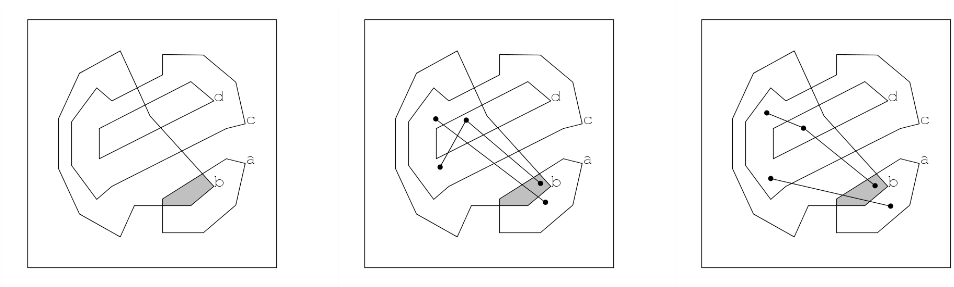
In our implementation of the system, we use a weighting of 1 for the edge crossing metric. The weighting of the edge length metric is relative to this and is chosen such that when the returned value is multiplied by the weighting, the value is typically less than 1. Larger values may allow total edge length to be reduced at the expense of introducing new edge crossings. Our implementation uses a weighting of 1×10^{-3} for the edge length metric weighting. Some examples of "quality" values are illustrated in Figure 11.



Crossings: 0.0000	Crossings: 1.0000	Crossings: 0.0000	Crossings: 0.0000
Length: 0.0000	Length: 0.1143	Length: 0.0940	Length: 0.1515
Total: 0.0000	Total: 1.1143	Total: 0.0940	Total: 0.1515

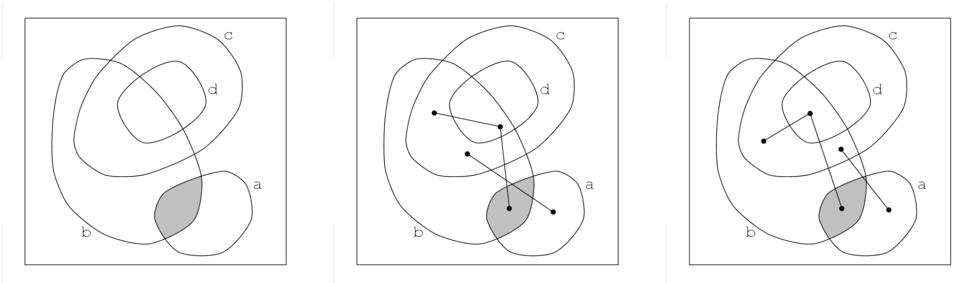
Fig. 11. Total quality metrics for some graph-enhanced diagrams.

The hill climber is also an iterative process and runs for either a fixed number of iterations, or a user may interact with the process and apply more iterations if it is deemed necessary. Each iteration begins with selecting a random zone that contains more than one node. A random pair of nodes is selected from this zone and their locations are swapped. This does not alter the meaning of the diagram, as they both lie within the same zone. If the new quality of the diagram is worse than before, the nodes are swapped back to their original locations; otherwise, the change is kept. After a number of iterations, the quality of the diagram according to the metrics improves. The effect of the hill climber can be seen in the last image in Figure 12. Smoothed versions of these diagrams are shown in Figure 13.



Initial Euler diagram	Initial node placement	After force directed placement and hill climbing

Fig. 12. The effect of using the force directed node placement and hill climber on graphs being embedded into an Euler diagram that has been drawn automatically.



Initial Euler diagram | Initial node placement | After force directed placement and hill climbing

Fig. 13. Embedding the previous graphs into the same Euler diagram laid out with the smoothing system. It is easier to distinguish between the curved contours and the straight edges.

4 Drawing Spider Diagrams

In this section we describe how we apply our method to spider diagrams. The method is essentially that described in Section 3, where we describe a method to draw graphs on Euler diagrams, except that spider diagrams do not have arbitrary graphs connecting nodes. Instead, nodes are connected in spanning trees, and the manner in which the nodes are connected in the tree is not significant. The abstract syntax of spider diagrams expresses spiders purely in terms of their habitat: a collection of zones. A spider whose habitat comprises three zones, z_1, z_2, z_3 can be drawn with a graph edge (the spider's *leg*) drawn between graph nodes (the spider's *feet*) in z_1 and z_2 and a second leg between graph nodes (the spider's *feet*) in z_2 and z_3. An alternative drawing might draw legs between z_1 and z_2 and between z_1 and z_3. Only once a spider is drawn do we know which of its feet have a leg between them. As we only have the information about which sets of nodes are connected, our drawing method needs an additional process that develops a tree between the nodes.

Once the feet for each spider have been placed, it is possible to use Prim's or Kruskal's algorithm to form a minimal spanning tree. This completes the concrete representation of the spider with the smallest total edge length, but does not take into account edge crossings. As our hill climbing method gives preference to changes that reduce edge crossings, we do not create a minimal spanning tree, but trivially form a chain of spider legs that connect each spider foot.

In [18,10], spider diagrams are given semantics, and diagrammatic reasoning rules. A reasoning rule transforms one diagram into another, whose semantics are a logical consequence of the premise diagram semantics. A proof in the spider diagram reasoning system is simply a sequence of these diagrammatic transformations, which could be elicited from a user, with a software tool assisting in the valid application of reasoning rules. Alternatively, proofs can be automatically generated between given premise and conclusion diagrams [9,10]. An example of an automatically generated proof is shown in Figures 14, 15 and 16, where the rules “Add Shaded Zone” and

"Add Spider Foot" have been applied. The first rule changes the underlying Euler diagram, and the second rule changes the superimposed graph. Without any results on drawing spider diagrams, the proof can only be presented in its abstract form (Figure 14). The preliminary work on drawing can present the proof with correct but unappealing diagrams (Figure 15). After combining the algorithm described in this paper with the previous work on smoothing [8], the proof is presented in a most readable fashion (Figure 16).

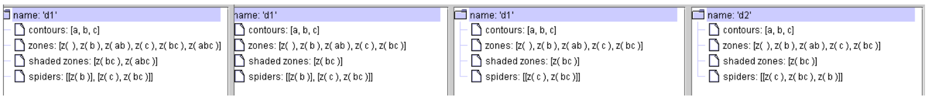


Fig. 14. An abstract proof.

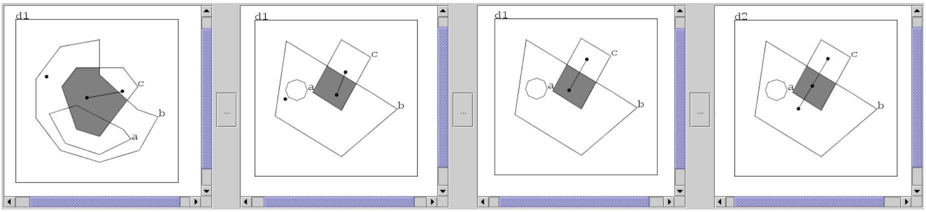


Fig. 15. A drawn proof with unappealing diagrams.

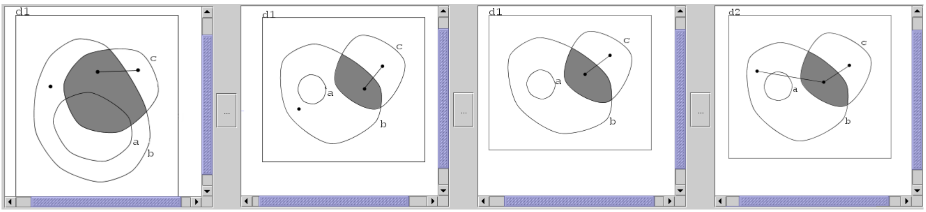


Fig. 16. A drawn proof with smoothed diagrams.

The final position of spider feet in an automatic layout often depends on their initial placement before the force model is invoked. We sometimes observe nodes getting stuck in locally minimal energy states, where they are unable to move elsewhere in a zone. To escape from the local minima, simulated annealing could be employed to make nodes periodically jump a larger distance to see if it is beneficial to the energy level in the force model. Figure 17 illustrates one example of a bad layout caused by local minima.

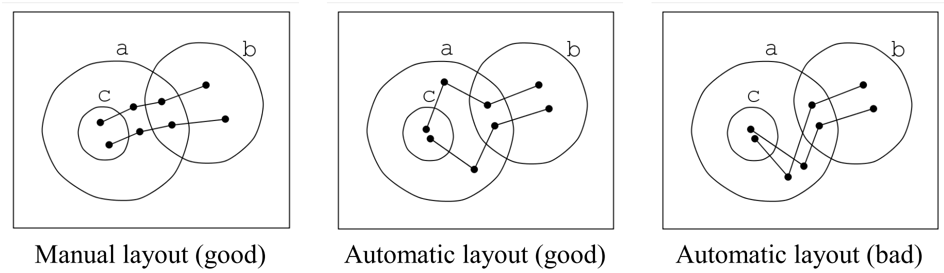


Fig. 17. Different layouts for the same graphs in an Euler diagram. The bad automatic layout occurs when both nodes in zone *a* are initially placed close to each other and reach a local minima while the force model is being simulated. In this case, it is not possible to reduce edge crossings without moving nodes.

5 Conclusions and Further Work

We have presented a method for automatically embedding graphs in Euler diagrams. The Euler diagrams are laid out using a multicriteria optimizing system. Nodes are placed at initial locations before being refined with a force model that involves interactions between zone boundaries and other nodes in the same zones. Finally, edge crossings and total edge length in the graphs are reduced without changing the meaning of the diagram, using a hill climbing approach. We have also specialized our method to apply to the syntax of spider diagrams and we demonstrate a software tool that draws automatically generated proofs in a spider diagram reasoning system.

The current implementation of the force model for placing nodes does not guarantee in all cases that nodes will remain inside the correct Euler zones. Allowing nodes to move out of their zones is very undesirable, as this would change the structure and meaning of the diagram. We are confident that in all but special cases (those where the nodes are initially placed very close to zone borders and with other nodes nearby) the force model maintains the node locations, but it would be relatively simple to add a structure check to the diagram after each iteration of the force model. This would check that the nodes remain in the correct part of the diagram. If a node movement had changed the structure by moving outside of the correct zone then the node could either be placed back where it originated or placed randomly in the correct zone.

At the moment, the optimization of the graph layout relies on swapping nodes that are in the same zone. A further addition to this for spider diagrams is to change the spanning tree of a spider as a move in the hill climber, in an attempt to improve edge crossings and edge length of the final graph. We feel that this can improve the layout of spiders.

The example of a proof shown in Figures 14, 15 and 16 was chosen well, to ensure that all the intermediate diagrams are drawable (a property of Euler diagrams, as described in [7]). More work needs to be done to resolve, and draw, Euler diagrams that are currently diagnosed as “undrawable”. Resolution will require the drawing of diagrams which use multiple crossing points between contours, and may even allow different contours to share a concurrent path. There are usability drawbacks to these kinds of diagram syntax, but perhaps even more serious usability drawbacks if we can create no drawing at all for a proof step. The smoothing approach would need to be adapted in the case that multiple contours were allowed to pass through the same point, for example. Without adapting the algorithm, the concurrent contours would inevitably “pull apart”, creating extra zones which change the underlying diagram structure. This is an issue with the applicability of drawing Euler diagrams and not directly relevant to this paper.

Another important question raised by the proof-presentation application is that of continuity of proofs. When a diagram transformation is made (a new zone is added or a new spider foot is joined), the transition is best understood if the concluding diagram closely resembles the preceding diagram, highlighting only a local change. We seek to maintain the mental map between the dynamic visualizations at each proof step. There are several possible ways to achieve this. One method is to include a mental map criteria across all diagrams when performing hill climbing at both the Euler Diagram and node location stages. Another method is to draw the first diagram nicely, and then attempt to draw subsequent diagrams incrementally to remain as close to previous ones as possible. It is also worth noting that phases 2 and 3 of the current system could be combined to form a meta-heuristic, which may simplify mental map preservation.

This task of drawing one diagram given another diagram which is structurally similar, generalizes to drawing given a context which is a library of drawn examples. Creating drawings in such a context could allow a tool to learn about user preferences in diagram layout.

Acknowledgements

This work has been partially supported by EPSRC grants GR/R63509/01 and GR/R63516/01. We would also like to thank John Taylor and Richard Bosworth at the University of Brighton for helpful comments on early versions of this paper.

References

1. François Bertault, Peter Eades. Drawing Hypergraphs in the Subset Standard. GD 2000. LNCS 1984. pp. 164-169.
2. R. De Chiara, U Erra and V. Scarano. VENNFS: A Venn-Diagram File Manager. IEEE Proceedings of Information Visualization (IV03). pp. 120-126. 2003.
3. M.P. Consens and A.O. Mendelzon. Hy+: A Hygraph-based Query and Visualization System. Proc. of the ACM SIGMOD Int. Conf. on Management of Data, pp. 511-516, 1993.
4. Peter Eades, Qingwen Feng. Multilevel Visualization of Clustered Graphs. GD 1996. LNCS 1190. pp. 101-112.
5. A. Fish, J. Flower, and J. Howse. A Reading Algorithm for Constraint Diagrams. Proc. IEEE 2003 symposium on Human-Centric Computing languages and environments (HCC `03), pp. 161-168.

6. A. Fish, and J. Howse Computing Reading Trees for Constraint Diagrams. To appear in Proceedings of AGTIVE `03.
7. J. Flower and J. Howse. Generating Euler Diagrams, Proc. Diagrams 2002, Springer Verlag, pp. 61-75. 2002.
8. J. Flower, P. Rodgers and P. Mutton. Layout Metrics for Euler Diagrams. Proc. IEEE Information Visualization (IV03). pp. 272-280. 2003.
9. J. Flower and G. Stapleton. Automated Theorem Proving with Spider Diagrams. To appear in proc. Computing Australasian Theory Symposium (CATS04).
10. J. Flower, J. Masthoff and G. Stapleton. Generating Readable Proofs : A Heuristic Approach to Theorem Proving With Spider Diagrams, Diagrams '04.
11. J. Flower, J. Howse and J. Taylor. Nesting in Euler Diagrams, Syntax, Semantics and Construction. In Journal of Software and Systems Modelling (SoSyM), Springer Verlag, Issue 1, article 5.
12. T.M.J. Fruchterman, E.M. Reingold. Graph Drawing by Force-directed Placement. Software – Practice and Experience Vol 21(11). pp. 1129-1164. 1991.
13. D. Harel. On Visual Formalisms. Communications of the ACM. 31(5). pp. 514-530. 1988.
14. David Harel, Gregory Yashchin. An Algorithm for Blob Hierarchy Layout. Working Conference on Advanced Visual Interfaces. pp. 29-40, May 2000.
15. Higraph web page: http://db.uwaterloo.ca /~gweddell/higraph/higraph.html.
16. J. Howse, F. Molina, J. Taylor, S. Kent. Reasoning with Spider Diagrams, Proc. IEEE Symposium on Visual Languages 1999 (VL99), IEEE Press, 138-147.
17. Erkki Mäkinen, How to draw a hypergraph. International Journal of Computer Mathematics 34 (1990), 177-185
18. G. Stapleton, J. Howse, J. Taylor and S. Thompson. What Can Spider Diagrams Say? Diagrams 2004.
19. G. Stapleton, J. Howse and J. Taylor. A Constraint Diagram Reasoning System. Proc Visual Languages and Computing '03, pp. 263-270.
20. Visual Modelling Group: technical report on spider diagram reasoning systems at www.cmis.brighton.ac.uk/research/vmg/SDRules.html
21. R. Davidson, D. Harel. Drawing Graphs Nicely Using Simulated Annealing. ACM Transactions of Graphics, Vol. 15, No. 4, pp. 301-331. 1996.
22. S. Chow and F. Ruskey. Drawing Area-Proportional Venn and Euler Diagrams. To appear in Proceedings of GD2003. LNCS. Springer Verlag.
23. Howse J, Molina F, Taylor J, On the completeness and expressiveness of spider diagram systems. Proceeding of Diagrams 2000, Edinburgh 2000, LNAI 1889, Springer-Verlag, 26-41.