Eurographics/ IEEE-VGTC Symposium on Visualization (2007)
Ken Museth, Torsten Möller, and Anders Ynnerman (Editors)

Manual Clustering Refinement using Interaction with Blobs

Christian Heine and Gerik Scheuermann

Department of Computer Science, University of Leipzig, Germany

Abstract

The huge amount of different automatic clustering methods emphasizes one thing: there is no optimal clustering method for all possible cases. In certain application domains, like genomics and natural language processing, it is not even clear if any of the already known clustering methods suffice. In such cases, an automatic clustering method is often followed by manual refinement. The refined version may then be used as either an illustration, a reference, or even an input for a rule based or other machine learning algorithm as a new clustering method.
In this paper, we describe a novel interaction technique to manual cluster refinement using the metaphor of soap bubbles, represented by special implicit surfaces (blobs). For instance, entities can simply be moved inside and outside of these blobs. A modified force-directed layout process automatically arranges entities equidistant on the screen. The modifications include a reduction to the expected amount of computation per iteration down to $O(|V|\log|V|+|E|)$ in order to achieve a high response time for use in an interactive system. We also spend a considerable amount of effort making the display of blobs fast enough for an interactive system.

Categories and Subject Descriptors (according to ACM CCS): G.2.2 [Discrete Mathematics]: Graph algorithms H.3.3 [Information Storage and Retrieval]: Clustering

1. Introduction

Clustering deals with the identification and grouping of similar entities according to a given metric. Let S be a set of entities, then a *clustering* C is a set of clusters $C \subseteq 2^S$ (2^S denotes the *power set* of S). A *partition* or *classification* is a clustering C which satisfies

$$\bigcup_{c\in C} c = S$$

and

$$\forall a,b \in C : a \neq b \rightarrow a \cap b = \emptyset.$$

A *strictly hierarchical clustering* is a clustering C where

$$\bigcup_{c\in C} c = S$$

and

$$\forall a,b \in C : a \cap b \neq \emptyset \rightarrow (a \subseteq b \vee a \supseteq b).$$

Other types of clusterings may be simply called “common”.

There are two goals that have to be achieved for the manual improvement of clusterings. The first is to effectively communicate the current structure of the clustering to the user and the other is to provide him with means to change that structure.

One method of achieving the first goal is to lay out the entities, represented by graphical elements, in a scatterplot, positioning similar entities very close to each other. A cluster is indicated by a high density of points at one location. This can be used to quickly communicate attributes like size and extend of a cluster but often suffers from a poor usage of screen space. Large parts of the screen usually stay blank. As an improvement, it is possible to use additional information like color or shape of graphical elements to indicate which entity belongs to which cluster. Using this method, the position of the graphical element has less importance, or – what is of more importance – there is a larger degree of freedom for the placement of elements. This larger degree of freedom may be used to distribute the entities more uniformly on the drawing area, but in general, only a small number of colors and shapes can be properly distinguished by an observer [War04]. So most existing systems combine scatterplots with colors and shape and neglect to use the freedom of positioning elements evenly on the screen.

There is another possibility for increasing the degree of freedom for placement. Again graphical elements are drawn

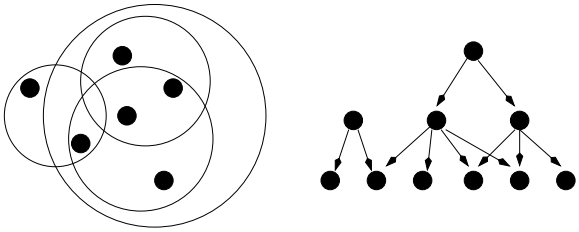


Figure 1: *DAG visualization of a clustering.*

so that similar items are in similar places. These elements are then surrounded by a graphical shape per cluster that directly indicates that all elements inside the region of this shape belong to the cluster. As long as these graphical shapes do not overlap each other except for elements that belong to each cluster simultaneously, this method communicates the clustering properly. The elements may be distributed more uniformly on the screen as long as neighborhood relationships between the graphical elements are maintained. We use blobs as graphical shapes that indicate clusters.

For the distribution of elements on the screen we use a force-directed layout algorithm. Generally, such a layout algorithm determines the embedding of a graph $G = (V,E)$ in R^n, where n is usually 2 or 3, using a physical simulation. Vertices of V are considered to be particles, sometimes having a mass or charge, and edges of E are replaced by springs or springlike connections between the vertex particles. The simulation then computes the physical forces between the particles and moves the particles according to these forces towards a configuration of lowest energy, or at least one configuration of equilibrium, i.e., where the cumulated force on each particle vanishes. To avoid such local minima simulated annealing is used.

Finally there is the interaction. Blobs appear to most lay observers like soap bubbles. In reality soap bubbles are too fragile, but we allow entities to be pushed inside or pulled outside of bubbles. For generality, we also allow entities to be part of multiple clusters, although, again in reality, it is impossible for two soap bubbles to intersect each other.

The structure of a clustering can be described by an acyclic directed graph. Each entity and each cluster is a vertex in that graph, and there is an edge from vertex A to B, if and only if there are corresponding clusters C_A and C_B with $C_A \supset C_B$ and there is no vertex C with corresponding cluster C_C and $C_A \supset C_C \supset C_B$. The layout of a DAG can also be used to visualize the clustering (Figure 1), but we will use this constructed DAG only as a data structure later on. Intersecting blobs remind the observer of Venn diagrams. Many people are more familiar with them than with directed acyclic graphs.

Some optimizations used in this paper require common techniques of computational geometry, namely Voronoï diagrams, Delaunay triangulations and trapezoidal maps. The introduction to these topics and algorithms is beyond the scope of this paper, they are well known and can be found in any textbook about computational geometry (e.g. de Berg *et al.* [dBvKOS00]). The knowledge of these methods is not required for the understanding of the interaction techniques presented in this paper.

2. Related Work

There are many possibilities shapes that can be drawn around the entities of clusters. Some systems already implement some of them, e.g., Frishman and Tal [FT04] use rectangles, Kumar and Garland [KG06] use nicely shaded circles, van Ham and van Wijk [vHvW04] use one sphere per entity and closely positioned elements automatically clump together. Sprenger *et al.* constructed ellipses around entities of the same cluster using a PCA technique to determine the orientation and length of the ellipses' axes. None of these techniques could guarantee that the shapes would not include elements from different clusters. Gross *et al.* [GSF97] and Sprenger *et al.* [SBG00] use blobs in 3D but only consider partition and hierarchical clusterings. We also chose blobs, as they create compact graphical shapes even for awkwardly positioned entities. However, we only use the 2D representation of them, because it makes the interaction methods simpler for the user.

The force-directed method was first proposed by Eades [Ead84], who used electrically charged particles that repel each other and springs that force vertices connected through an edge to attract each other. Kamada and Kawai [KK89] modeled the system using springs exclusively. These early approaches were refined by Fruchterman and Reingold [FR91]. Most formulations of these force-directed layout algorithms require the graph to be connected and suffer from long running times. Our system implements a simple spring embedder that has been modified to avoid these two problems. We chose a single-level force-directed layout mainly, because the user of the system can watch it work and it has a high "dynamic stability", i.e., if the graph is altered after its layout has been determined the layout of the new graph will look almost the same.

The simple force-directed layout algorithms converge very slowly, if they converge at all, to readable layouts for very large graphs. Newer formulations solve this problem by calculation of the layout on multiple resolutions or levels (e.g. [HJ04, HK01, ACL04, CCLP03]), which perform better both in visual appearance as well as running time according to Hachul and Jünger [HJ06]. While these algorithms are useful for an initial layout of the entities on the screen, during the interaction phase, we actually prefer the simpler force-directed algorithms for their higher "dynamic stability" and because it is easier to trap them in a local minimum. This leaves more freedom for the user to arrange entities after their fashion and taste.

3. Cluster Visualization using Blobs

We visualize clusters by blobs. Let S be a set of vertices. A (simple) blob of a cluster $C \subseteq S, C \neq \emptyset$ is an implicit surface where each $\vec{x}$ on this surface satisfies:

$$\sum_{e \in C} \frac{\omega(e)}{\|\vec{x} - \vec{x_e}\|^k} = \gamma$$

$\vec{x_e}$ is the position of the element e, γ is an arbitrary positive threshold and k an arbitrary positive number. For our implementation we chose $k = 2$ and for simplicity $\gamma = 1$. $\omega(e)$ is the weight of an element. It is used to give entities of high importance a larger area around them. If no weights are desired one can safely set $\omega(e) = 1$ for all elements $e \in E$. The points that satisfy the equation not necessarily form a connected structure. So we either have to guarantee that the blob is always connected, or indicate blobs belonging together by an additional attribute, e.g., color. We chose to indicate such enclaves by color, and draw not only the surface but also the interior of every blob. A point $\vec{x}$ belongs to the interior if it satisfies:

$$\sum_{e \in C} \frac{\omega(e)}{\|\vec{x} - \vec{x_e}\|^k} \geq \gamma$$

However, we modified this formula to the following one:

$$\sum_{e \in E} \frac{g_C(e)\omega(e)}{\|\vec{x} - \vec{x_e}\|^k} \geq \gamma \tag{1}$$

where

$$g_C(e) = \begin{cases} 1 & e \in C \\ -0.25 & e \notin C \end{cases}$$

This avoids blobs accidentally overlapping entities that are not part of the cluster. Even badly positioned entities would rather create a hole in the blob than to be overlapped by it.

If the number of entities is large, it may be preferable to show only a subset of them. For that we use the standard zoom and pan technique. We also allow the user to collapse and expand clusters. We represent collapsed clusters by a blob of one base point. Subclusters are no longer visible, i.e., there will be no blob for them. There is a problem when elements are part of more than one collapsed cluster. In that case we use one representative for the intersection of both clusters and one for the rest of the cluster.

Usually the entities are represented on screen by their name. We do not automatically determine the optimal name of a collapsed cluster based on its contents as this is highly application dependent. If an application does not provide a name for a collapsed cluster, only a number is shown giving the number of entities in thus cluster.

Instead of using an isosurface algorithm to extract an approximation of the blob surface, we simply test each pixel of the screen if it is inside a certain cluster. As we allow elements to be part of any cluster, i.e., the property that the clustering is a partition or a hierarchy is not required, a pixel may belong to multiple clusters. When presented on the screen the pixel will get the average of all colors that have been assigned to each cluster.

A naïve implementation can be very slow, because each pixel requires the computation of the distance of it to each entity position and the summation of the contributions to each cluster. This results in a complexity of $O(A \cdot N \cdot M)$, with N being the number of entities, A being the area or number of pixels, and M being the number of clusters. A meaningful approximation of the area A in our case is $A \in O(N)$ as our layout algorithm will give each element the same amount of screen space, and rarely clusterings will not satisfy $M \in O(N \log N)$. So the overall complexity of this method can be approximated by $O(N^3 \log N)$.

If the clustering is a strict partitioning, then $M \leq N$ and the method is bounded by $O(N^3)$. If the clustering is strictly hierarchical, then the contribution of each entity to each pixel has only to be computed for the leaves of the hierarchy tree and can be propagated in $O(M+N)$ time upwards along the tree edges. So the overall complexity can be reduced to $O(A(N+N+M)) = O(N^2 \log N)$. This idea can be extended to general clusterings. The clustering can be described by a directed acyclic graph with $M+N$ vertices, and a propagation tree inside this DAG may easily be constructed and maintained if the clustering changes.

The method can be sped up using the following heuristic. Most of the time, neighboring pixels belong to the same cluster. First we determine the set of clusters for each pixel of a rectilinear grid. Then we test the four adjacent points of each mesh cell for the equality of the set of clusters. If they are all equal, we assume that each pixel of the cell belongs to the same clusters. If one of them differs, we split the cell in four equally sized cells, compute the five new points, and proceed recursively, terminating if our cells' area equals exactly one pixel. This is illustrated in Figure 2. This makes the method much faster, although still staying in the aforementioned order of time complexity. Additionally it can introduce artifacts, e.g., if the cells of the mesh are too large, and a blob is fully contained inside it. As a rule of thumb, our implementation uses a grid size that is the biggest power of two that is still smaller than the diameter of a blob with a single entity.

But we can still do better. When testing a certain pixel, we might neglect entities that are far away from it, because their contribution to Equation (1) is minimal. So it is sufficient to just consider entities that are very near. One possibility is to compute the Voronoï partitioning of the screen space using the entities positions as input sites. This partitioning is then used to find the nearest site to each pixel considered in the quad-tree test and the neighbors of that site. As a site may have up to $N-1$ neighbors, this does not reduce the upper bound of the time complexity but can be expected to perform much better.

An alternative is to compute the Delaunay triangulation

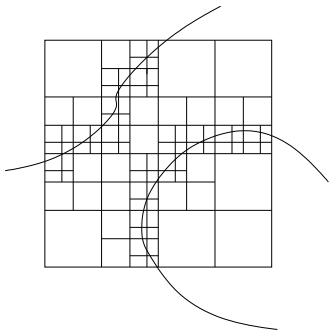


Figure 2: *quad-tree test to approximate boundary*

of the entities' positions and find for each considered pixel the surrounding triangle and consider the contribution of the three entities adjacent to the triangle and optionally also the adjacent entities to the three adjacent triangles. Using this method a maximum of 6 entities contribute to Equation (1). The search for the correct Voronoï cell or Delaunay triangle can be done in $O(\log N)$ time using trapezoidal maps so the overall complexity reduces to $O(N\log N + A\cdot(\log N + h))$. The first part is given by the Voronoï partitioning or Delaunay triangulation, the second part consists of determining the Voronoï cell or Delaunay triangle and the propagation of the contribution of the neighboring sites through the directed acyclic graph for each considered pixel. h is the height of the propagation tree in the DAG describing the clustering structure. This height is seldomly outside $O(\log N)$ so the overall complexity can be approximated by $O(N\log N)$. It has to be noted, however, that the later optimizations may result in discontinuities along cell or triangle boundaries. As long as the entities are evenly distributed the artifacts are neglectible. Fortunately the layout process automatically achieves that property.

4. Layout

The layout process generates a simple force-directed layout. Most force-directed layout algorithms require the graph to be connected, however, some of the datasets we are provided with contain only entities without any edges between them. Furthermore, we want to show the clustering structure and not the full relationships between the entities. We use a simplified van der Waals force for the general forces between particles:

$$\vec{F}_r(e) = \sum_{n\in X\setminus\{e\}} \Omega\left(\frac{\|\vec{x}_e - \vec{x}_n\|}{2\cdot\gamma}\right)\frac{\vec{x}_e - \vec{x}_n}{2\cdot\gamma}$$

where

$$\Omega(d) = \begin{cases} -2 & d < \frac{1}{2} \\ \frac{1}{d} - \frac{1}{d^2} & \frac{1}{2} \le d \end{cases}$$

It repels particles if their distance is less than $2\cdot\gamma$ and attracts them otherwise. So this force is sufficient to nicely distribute the not necessarily connected entities uniformly on the screen.

If we would implement this method each entity would create a force on another one, so our algorithm would be $O(N^2)$ per iteration. However, because the force vanishes quickly with increasing distance it is sufficient to look only at the closest entities. Again, the set of neighbors of each entity can be computed using either Voronoï partitioning or Delaunay triangulation. The formula becomes:

$$\vec{F}_r(e) = \sum_{n\in N(e)} \Omega\left(\frac{\|\vec{x}_e - \vec{x}_n\|}{2\cdot\gamma}\right)\frac{\vec{x}_e - \vec{x}_n}{2\cdot\gamma}.$$

Because the number of neighbors to consider lies in $O(N)$ the Voronoï partitioning or Delaunay triangulation respectively become the driving factor for the complexity. Because of that, the general forces may be computed in $O(N\log N)$ time.

However, if we were initially provided with edges, we use them in our layout. We use simple logarithmic springs to determine the current force, setting the optimal spring length to $2\cdot\gamma$. The complexity of one layout iteration is now $O(N\log N + E)$, E being the number of edges. Much more interesting, however, is how to enforce entities of the same cluster to be grouped together. The straightforward way would be to add edges or springs that connect entities of the same cluster. But because each entity might be connected with each other entity of the same cluster and each entity may belong to multiple clusters, this introduces far too many edges. Instead, we implemented a probabilistic algorithm. First we choose a cluster randomly, but respect the size of the cluster, i.e., large clusters have a linearly greater probability for being chosen. Inside this cluster we choose two distinct entities and compute the force of a logarithmic spring between them. We repeat the process $O(N\log N)$ times. The ideal spring length is set to γ. This results in entities to be nearer, if they are in the same cluster.

We allow the user to arrange the entities on the screen in any way he sees appropriate. He can do this by moving either a single entity, in which case he has to click on that entity and drag it around with the mouse, or even a whole cluster. To do that, he clicks somewhere inside the blob, the program will determine the correct cluster or clusters just like it would determine the set of blob that contain the pixel. If the mouse is then dragged, all entities of all selected clusters follow accordingly.

Because the layout process is active during this interaction for unselected elements, these will automatically make way. If the entity or the cluster is released the layout process will again arrange the items do that entities are uniformly distributed, but, in general, it will preserve the horizontal and vertical ordering of the entities.

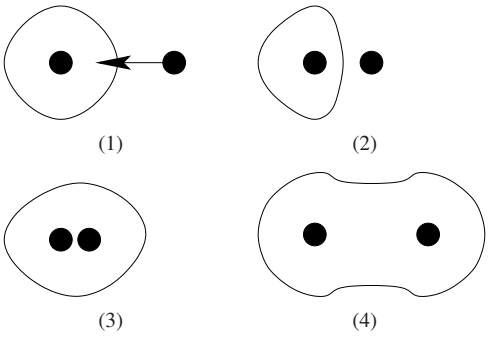


Figure 3: *Addition of an entity to a blob.*

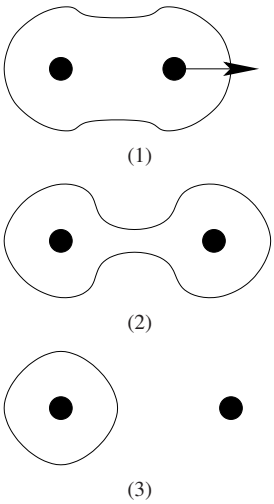


Figure 4: *Removal of an entity from a blob.*

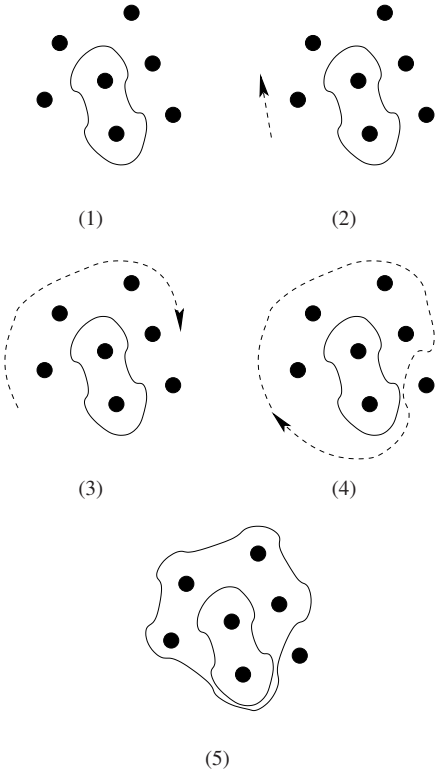


Figure 5: *Grouping of multiple entities.*

5. Cluster Modification using Blobs

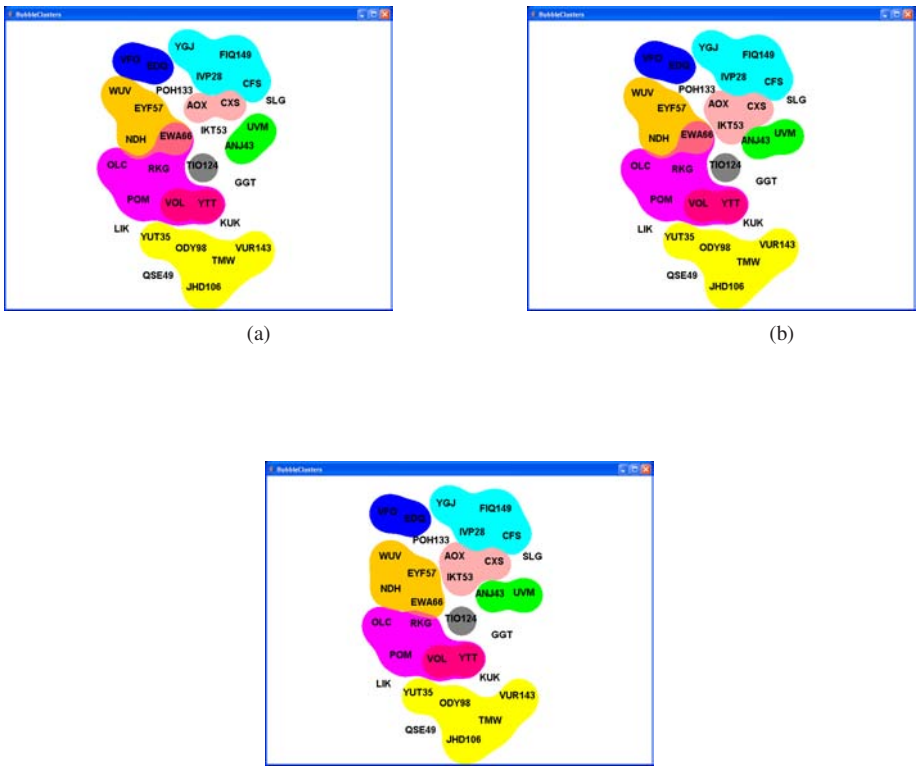
A user can change a given clustering using one of the following methods:

- The user moves one entity very close to another entity that is in one or more clusters. If the distance of both entities drops below a certain threshold δ the moved entity is assigned to all the clusters the other entity belongs to. It appears as if the entity was pushed into the soap bubble. This process is illustrated in Figure 3. We chose $\delta = \frac{\gamma}{2}$.
- A single entity is moved away from entities of the same cluster, thereby stretching the cluster. Once the distance to all the other entities grows beyond a certain threshold Δ, it is removed from the cluster. It appears as if the entity was pulled out of the soap bubble. This process is illustrated in Figure 4. In our implementation we use $\Delta = 2 \cdot \gamma$.
- The user draws a freehand line around some entities. These entities will be assigned to a new cluster. This appears as if the user has made a new soap bubble around some entities (Figure 5).
- The user double-clicks on a cluster, thereby deleting it. The bubble appears to be pierced and burst.

Because the layout process is active during this interaction, the user has to move the entities quickly enough, so that the other entities do not flee to fast (in case of pushing an entity into a cluster) or follow to fast (in case of pulling an entity from a cluster). In our implementation we update the layout exactly 50 times a second and enforce an upper bound on the velocity of an item empirically. An alternative would be to use a modifier key (e.g. "Shift") to slow down the layout update. We also use the "Ctrl" modifier key, if we drag entities around and do not want them to leave the blob regardless of how far it is dragged from its original cluster. This is really helpful in situations where we want one entity of an cluster be part of another one as well, but leave the others as they are.

6. Results

We implemented our methods in a system called "BubbleClusters" in Java. The optimization techniques used make layout recomputations and redisplay of blobs fast enough to be interactive on a Pentium IV 2.54GHz. The first users of our system found it very intuitive and "fun to work with".



(a) (b)

(c)

Figure 6: *(1) BubbleClusers' grouping technique was used to create this initial clustering to some randomly created gene names. (2) An entity has been pushed inside a cluster. (3) An entity has been pulled outside a cluster.*

Figure 6 shows the system working on an example that was inspired by a common task of the domain experts that we developed the application for. The application depicted is from the field of genomics, where a given set of genes have been clustered for same functionality, but genes often are part of multiple clusters, as they behave differently in the presence of other genes. Biologists would like to create a partition of the genes to gene groups using the knowledge they already have from specific experiments. Figure 7 shows a real world dataset to illustrates how complex the clusterings before refinement can get. Another application is the clustering of words of natural language into topics or grammatical groups like nouns or verbs. Some word (e.g. space) may belong to multiple topics and may even have different meanings in each context.

7. Conclusions and Future Work

Since automatic clustering approaches are helpful but not perfect for most applications, there is an urgent need for manual refinement of clusterings. We represented an intuitive, interactive system based on blobs that can display arbitrary clusterings and allows for fast and intuitive layout and clustering manipulation. First user tests demonstrated a "fun factor" that gives some evidence for the intuitive understanding of the system.

While the system is of value to any clustering problem where manual refinement of clustering or layout is necessary, we have two applications areas as major targets that led us to the development of the system. First, we are working closely with biologists looking at clusterings generated from gene expression data. The initial automatic clustering based on the data contradicts knowledge from other experiments to some extend, so they asked us to provide them with a tool like the developed system. Currently, intensive user testing with them is the next step in this direction. Second, we cooperate also with the natural language department group in our department. They work intensively on topic maps, where automatic clustering and manual refinement based on human

Figure 7: *The complexity "BubbleClusters" can handle. The shown dataset is a randomly selected part of a real world dataset of a correlation analysis of genes. It consists of 78 entities and 24 clusters. On average each entity is part of 4 clusters.*

understanding are an important subtopic. This different application allows us to do intensive user tests with a second user group in the near future.

We also consider improvements to the actual drawing method of the blobs. Mixing colors often leads to many overlapping clusters to be drawn gray. Instead, we could respect information from the clustering structure respectively the DAG, and only mix colors of unrelated clusters and use the color of a subcluster in case we detect that the overlap is only between a cluster and its subcluster. We could also use simple textures instead of plain colors and mix them. A method to draw the border of the blobs would also help improve the perception of the blobs.

8. Acknowledgements

We would like to thank Maciej Rosolowski for providing some datasets to work on. We would also like to thank the anonymous reviewers for all their concerns and suggestions to improve the paper.

References

[ACL04] ANDERSEN R., CHUNG F. R. K., LU L.: Drawing power law graphs. In *Graph Drawing* (2004), Pach J., (Ed.), vol. 3383 of *Lecture Notes in Computer Science*, Springer, pp. 12–17.

[CCLP03] CHAN D., CHUA K., LECKIE C., PARHAR A.: Visualisation of power-law network topologies. In *Proceedings of the Eleventh IEEE International Conference on Networks* (2003), pp. 69–74.

[dBvKOS00] DE BERG M., VAN KREVELD M., OVERMARS M., SCHWARZKOPF O.: *Computational Geometry: Algorithms and Applications (2.,rev. Ed.)*. Springer, 2000.

[Ead84] EADES P.: A heuristic for graph drawing. *Congressus Numerantium 42* (1984), 149–160.

[FR91] FRUCHTERMAN T. M. J., REINGOLD E. M.: Graph drawing by force-directed placement. *Software - Practice and Experience 21*, 11 (1991), 1129–1164.

[FT04] FRISHMAN Y., TAL A.: Dynamic drawing of clustered graphs. *infovis 04* (2004), 191–198.

[GSF97] GROSS M., SPRENGER T., FINGER J.: Visualizing informationon a sphere. *infovis 00* (1997), 11.

[HJ04] HACHUL S., JÜNGER M.: Drawing large graphs with a potential-field-based multilevel algorithm (extended abstract). In *Graph Drawing, New York, 2004* (2004), Pach J., (Ed.), Springer, pp. 285–295.

[HJ06] HACHUL S., JÜNGER M.: An experimental comparison of fast algorithms for drawing general large graphs. In *Graph Drawing, Limerick, Ireland, September 12-14, 2005* (2006), Healy P., Nikolov N. S., (Eds.), Springer, pp. pp. 235–250.

[HK01] HAREL D., KOREN Y.: A fast multi-scale method

for drawing large graphs. In *Graph Drawing, Colonial Williamsburg, 2000* (2001), Marks J., (Ed.), Springer, pp. pp. 183–196.

[KG06] KUMAR G., GARLAND M.: Visual exploration of complex time-varying graphs. *IEEE Transactions on Visualization and Computer Graphics 12*, 5 (2006), 805–812.

[KK89] KAMADA T., KAWAI S.: An algorithm for drawing general undirected graphs. *Information Processing Letters 31*, 1 (1989), 7–15.

[SBG00] SPRENGER T., BRUNELLA R., GROSS M.: H-blob: A hierarchical visual clustering method using implicit surfaces. *vis 00* (2000), 6.

[vHvW04] VAN HAM F., VAN WIJK J. J.: Interactive visualization of small world graphs. *infovis 04* (2004), 199–206.

[War04] WARE C.: *Information Visualization: Perception for Design (2. Ed.)*. Morgan Kaufman, 2004.