ASK-GraphView : A Large Scale Graph Visualization System

James Abello, *Member, IEEE*, Frank van Ham, and Neeraj Krishnan

Abstract—We describe ASK-GraphView, a node-link-based graph visualization system that allows clustering and interactive navigation of large graphs, ranging in size up to 16 million edges. The system uses a scalable architecture and a series of increasingly sophisticated clustering algorithms to construct a hierarchy on an arbitrary, weighted undirected input graph. By lowering the interactivity requirements we can scale to substantially bigger graphs. The user is allowed to navigate this hierarchy in a top down manner by interactively expanding individual clusters. ASK-GraphView also provides facilities for filtering and coloring, annotation and cluster labeling.

Index Terms— Information Visualization, Graph Visualization, Graph Clustering.

◆

1 INTRODUCTION

Over the past years two of the authors have spent significant time and effort designing, deploying and reviewing visualizations of large graphs. Most current systems are focused on nicely visualizing relatively small graphs [5,14,24] while most of the recent academic techniques in large scale visualization and graph drawing are focused on providing static drawings of relatively large graphs. Often these techniques are only applicable to graphs with specific structural properties. For example, [19] is only truly effective for very sparse or tree-like graphs. Layout techniques such as [18] offer increased scalability but are only really effective on grid-like graphs, such as those resulting from finite element methods.

Some of the interaction techniques and systems that have been proposed to navigate graphs (such as [11,12]) use the clustered graph navigation paradigm, which allows the user to navigate a potentially large graph by iteratively expanding or collapsing aggregate (cluster) nodes. However, these proposals assume a hierarchy is given for the input graph, which is rarely the case in practice. Their implementations are also limited in scale, with samples given for graphs of a few hundred nodes.

Other techniques, such as [2,3] also use the clustered graph paradigm. They create a hierarchy by using a fast algorithm and then display this hierarchy in an adjacency matrix representation. Because these systems construct a single huge hierarchy for a very large graph, the clustering algorithm used cannot be too sophisticated and often results in suboptimal hierarchies.

A second practical problem is that some users find it hard to understand the abstract matrix representations [16] of graphs that are used in these systems. We think that if a desktop node-link graph visualization is to be successfully applied as a practical tool for analysis of large amounts of data it needs to provide at least the following functionality:

1. A scalable architecture that is able to deal with hierarchical graphs even if they do not fit the RAM of the desktop machine and does not depend on the precise characteristics (such as density or link structure) of the graph (Section 4);

- *James Abello is with Ask.com and DIMACS, Rutgers University, E-Mail: jabello@ask.com.*
- *Frank van Ham is currently with IBM, but this work was performed while consulting for Ask.com in 2005, E-Mail: fvanham@us.ibm.com.*
- *Neeraj Krishnan is with Ask.com, E-Mail: nkrishnan@ask.com.*

2. The ability to automatically cluster large graphs, even if there is no additional hierarchical information available. Note that an often overlooked issue is the fan-out and depth of the hierarchy, which need to be controlled if one wants to navigate a graph interactively (Section 5);

3. Interactive techniques that allow the user to filter and explore this hierarchy without losing context (Section 6).

We present ASK-GraphView: a system that addresses all of these issues and is currently being used for the analysis of graphs with up to 200,000 nodes. Among the practical use cases of the system are manual inspection and updating of search content, assessment of new clustering approaches and infrastructure maintenance.

2 RELATED WORK

ASK-GraphView combines two major ideas from existing graph visualization systems. For user interaction and navigation it uses a node link layout of a clustered graph, which allows users to arbitrarily expand and collapse clusters to show the subgraph below. This basic idea is described in [12] and a number of experimental implementations based on it have been presented [11,20]. However, all of these implementations assume that the actual cluster hierarchy is either provided in the data or manually supplied by the user. Manually creating a hierarchy for a huge graph with unknown structure is difficult and tedious, and automatically creating a suitable hierarchy for a very large graph in limited time is by no means an easy task either. A second problem with these approaches is that users often tend to lose context when navigating clustered graphs with deeper hierarchies (in our opinion, more than 4 or 5 levels deep).

In terms of architecture ASK-GraphView borrows the concepts of memory and disk resident antichains from previous work on large graph navigation [1,2,3] and expands on these by incorporating time constraints. In previous approaches we used matrix oriented views of the antichains, but users often found these views too abstract [16]. Also, these views did not allow inspection of fine graph details and their relationships to the rest of the graph. Our system is probably closest to Tulip [7] which also provides automatic rendering and clustering. The main difference is that Tulip has been set up as a flexible test bed for graph visualizations, allowing (expert) users to compose different layouts and clustering approaches. We instead aimed for a system that can automatically provide a useful visualization of a graph upon loading and is easily accessible to domain experts with no visualization experience. Other related interaction techniques for large graphs include fisheye views
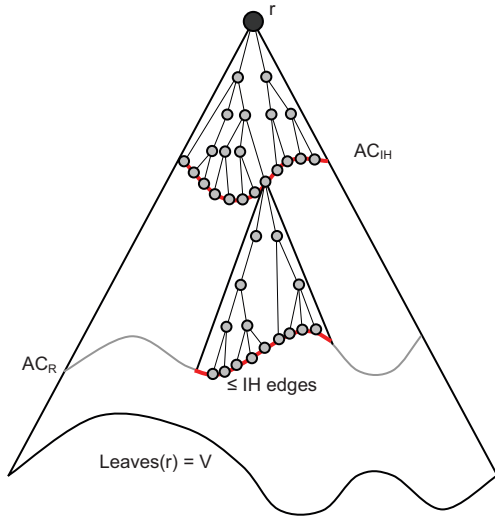
Fig. 1. A hierarchy on an arbitrary input graph $G = (V, E)$. AC_R indicates the part of the graph that is processed in RAM. AC_{IH} indicates the part of the hierarchy that is sent to a visualization client for further processing and visualization. For each node in AC_{IH} we ensure that there are no more than IH edges in the subgraph induced by its descendants on AC_R below.

of large clustered networks [15,17]. These create an abstraction of the graph based on the positions of one or more foci such that nodes nearer to the focus are displayed in greater detail. The major practical disadvantage of these types of views is that it is very hard to relate two visualizations of the same graph with different foci, even when providing visual cues such as animation or coloring.

3   PRELIMINARY DEFINITIONS

We consider weighted undirected graphs $G = (V, E, w)$ where V and E denote the set of vertices and edges respectively. w is a function that assigns to each edge in E a non-negative real weight. $|V|$ and $|E|$ denote the cardinalities of V and E. A subgraph of G is said to be induced by a subset U of V, if it consists of the nodes in U and those edges in E that have both endpoints in U. We denote an induced subgraph by $G(U)$.

A hierarchy tree TG for a graph G is a tree rooted at r whose set of leaves is in one to one correspondence with V. That is, $leaves(r) = V$, if we denote the set of descendant leaves of a node u in the tree by $leaves(u)$. The set of children of a node u is denoted by $children(u)$. Similarly, the unique parent of u is denoted by $parent(u)$.

A (maximal) antichain A in TG is a (maximal) set of nodes in TG such that no two distinct nodes in A are ancestors of one another. An antichain A_1 is said to be above another antichain A_2 if each node in A_1 is an ancestor (not necessary proper) of a node in A_2.

A maximal antichain A formalizes the notion of abstract views [1,3], slices [2] or abridgements [12] and it defines a partitioning of the nodes of G. On this partitioning we can define a macro edge set $E'=\{e_{ab} : a \in A \land b \in A : (e_{xy} \in E: x \in leaves(a) \land y \in leaves(b))\}$ with weights $w(e_{ab}) = \sum w(e_{xy}): e_{xy} \in E: x \in leaves(a) \land y \in leaves(b)$. The weighted graph (A, E') is a macro view of G. To make the defining antichain A explicit we refer to it as an A-view of G (see also [1,3]). In other words, the A-view of a graph is a macro graph consisting of the nodes in A and the macro edges connecting them. In the rest of this paper we assume that the visualization client has random access to every macro edge of a view, that is, we have a data structure available that stores both a hierarchy TG and the entire macro edge set on that hierarchy in RAM.

4   ARCHITECTURE

ASK-GraphView is a client server system that builds a data structure (i.e. a hierarchy tree) that treats the available screen (S), random access memory (R) and disk as three buffers of increasing size and with different access characteristics. Although we have previously considered only size constraints [1,2,3], we cannot offer interactive navigation if we do not also consider the temporal dimension. That is, we have to ensure that we are able to process any data that is transferred from disk to RAM (clustering) and from RAM to screen (layout) in reasonable time.

To address the first issue, we construct a stack of progressively finer disk resident macro views of our graph, such that the increase in number of edges in successive macroviews is limited (see also Fig. 1). We deal with the second issue by customizing a structural clustering algorithm such that we can guarantee that the fan out of the hierarchy generated by that algorithm is bounded.

The rest of this section discusses the system's time/space parameterized data structure and the direct relation between macro-views and antichains in hierarchy trees. In Section 6.2 we present a notion of graph context that allows the user to explore the graph without exceeding S.

4.1   Constrained Resources

In interactive graph navigation our two main bottlenecks are space and time. Space refers both to the maximum size of the graph R which we can still store in RAM and the maximum size S of the graph that we can interactively show on screen (see also [1,2,3]). However, if we want to offer interactive navigation of large graphs we also need to consider the time dimension.

The two main bottlenecks here are the time needed for computation of a clustering and the time needed for computation of a layout. IL (Interactive Layout) and IH (Interactive Hierarchy) are interactivity parameters that encode sizes of (sub)graphs for which a drawing and a hierarchical clustering can be computed in desired response times t_L and t_H (in our case a few seconds). We express IL in terms of the number of nodes and IH in the number of edges in the (sub)graph. Suitable values for IL and IH are obtained experimentally and clearly depend on the sophistication of the desired layout or hierarchy tree computation and the level of required interactivity for both. Given the current state of algorithmic knowledge and typical hardware configurations, it is reasonable to assume that $IL < S \le IH < R$.

The parameter IL effectively controls the maximum *fan out* of *nodes* in the hierarchy we are using to interactively navigate the graph. A hierarchy that has a maximum fan out of 10,000 is clearly not suitable to use for the interactive navigation of a graph, since it would take us too long to compute a satisfactory layout for a subgraph this big. Apart from this time constraint, IL is also limited by the user's ability to visually 'digest' the new subgraph presented. We determined empirically that $IL = 64$ provides a reasonable trade-off between scale and readability.

Similarly, IH represents the maximum amount of *edges* for which we can compute a *hierarchy* in a few seconds, regardless of the density of the graph. Using the procedures described in Section 5, we are able to hierarchically cluster around 100,000 edges at an interactive rate. Both of these navigation parameters are used to tune system performance in order to offer truly interactive graph navigation. In the following sections we assume that $2 < IL^2 < IH$ and $d*IL < S$ where d is a small non zero integer.

Trying to navigate dense graphs with node link diagrams is not really useful and other representations (such as adjacency matrices) might be more suited here. However, theoretically our infrastructure is also able to deal with denser graphs by replacing the clustering and hierarchy tree computation procedures by a simpler version that groups nodes together such that the fan out in the hierarchy tree is at most $\sqrt{IH}$. A more efficient alternative on weighted input graphs is pruning the graph by weight such that at most $|V|\sqrt{|V|}$ edges remain.

The main idea behind the architecture is that by reusing the parameter *IH* when navigating a macro view of a graph with *IH* edges, we can expand any node in that macroview to a new subgraph, again with at most *IH* edges (Fig. 1). Given that the maximum density is $|V|\sqrt{|V|}$, a subgraph of *IH* edges consists of at least $IH^{2/3}$ nodes. For each of these nodes we can then nest a new subgraph of $IH^{2/3}$ nodes, resulting in a total of $IH^{4/3}$ nodes below, having at most IH^2 edges (given density $|V|\sqrt{|V|}$). In practice however we cannot possibly create a hierarchy tree for a graph this big because our RAM size *R* can fit only a limited number of edges, so usually $R < IH^2$. If we use 32 bytes per edge, a server with 2GB of RAM available for edge storage will allow us to store in the order of 16 million edges.

This means that undirected graphs with up to 16 million edges are navigable with our software infrastructure (depending on available RAM) and this in turn offers a solution to the interactive navigation of graphs with up to 4,000 vertices with no qualifications. This number might seem disappointing since these graphs are not really that large. However, the real gains of our approach become noticeable by considering graph density, i.e. if $|E| < |V|\sqrt{|V|}$, $|V|$ lies between 64,000 and 300,000 vertexes. Similarly, if $|E| < |V|\log(|V|)$, $|V|$ lies between 300,000 and 16,000,000 vertices.

To our knowledge these results represent a substantial improvement over previous work dealing with the navigation of graphs represented by node-link diagrams, without posing qualifications on the actual structure of the graph. In summary, the lower the density of the input graph the larger graph scalability we obtain and improvements in clustering algorithms (i.e. higher *IH*) will automatically make navigation of larger graphs possible subject to RAM availability.

4.2   Data Structure

The fundamental data structure at the server level is a hierarchy tree *TG* that is parameterized by *IH* and *R*. The central idea is to find an antichain AC_R whose associated AC_R-view can be processed on the available RAM of size *R* and an antichain AC_{IH} above AC_R that can be processed in an interactive fashion. The antichain AC_{IH} is then processed on the client side to create a hierarchy. Should the user require parts of the hierarchy that are below AC_{IH}, the client sends a request to the server for more data which is subsequently processed in a similar manner. We assume throughout that $IL < S <= IH < R$, where $|V| < R < |V|+|E|$ (i.e. *G* is a semi-external graph as in [2]).

Definition: A good (*IH*, *R*)-parameterized hierarchy tree *TG* for a graph $G = (V, E)$ has the following properties:
1. It has a maximal antichain AC_{IH} above a maximal antichain AC_R such that the AC_{IH} and AC_R-views of *G* are of sizes not more than $|IH|$ and *R*, respectively. This requirement allows us to process the hierarchy in the first place.
2. Each of the subtrees rooted at nodes in AC_{IH} have at most $|IH|$ edges among their descendants in AC_R. This ensures that the server will return a reasonable amount of edges for each data request.
3. The subgraph induced by *leaves(u)* of each node *u* in AC_{IH} is connected. This is a minimal requirement to ensure that the subgraphs returned by the server reflect some of the graph's structure. Ideally, we would like each node *u* to reflect a dense cluster in the graph.
4. The depth of *TG* is $\log(|E|)$.

To ensure within-group connectivity and incorporate subgraph density and edge weights during the process, we use an ordered external memory version of Boruvka's contraction algorithm [1] that produces a binary hierarchy tree in $O(\log(|V|/R))$ passes over the input data. It also produces an antichain on it with no more than the desired number of elements *R*. The obtained *R*-view then, is a memory resident macro view of the original semi-external input graph *G*. Since AC_R fits in RAM we can use now an internal memory

version of the same described algorithm with AC_R as input to obtain a second antichain AC_{IH}.

4.3   Reducing the Depth of the Hierarchy

Running the procedure mentioned above may provide suboptimal results because the binary hierarchy might be very unbalanced. As an extreme case, consider a binary hierarchy where each node has one leaf and one non-leaf (except for the bottommost node). We refer to such a (part of a) hierarchy as a *comb*. Any antichain of size *n* in a comb, will have exactly *n*-1 leafnodes and one cluster. Clearly this is not a balanced abstraction of a graph. To address this problem we create a more balanced hierarchy by reducing the depth of the given binary hierarchy as much as possible without compromising the structural information that is contained in the hierarchy. We do this by aggregating a select set of nodes in the binary hierarchy under a common parent.

To guarantee that we can always find an antichain AC_{IH} above AC_R such that for any node in AC_{IH} the number of edges among their descendants on AC_R is no more than *IH* (see requirement 2 in 4.2), we have to limit the number of nodes we can aggregate. As an example consider a set of 500 nodes in AC_R that are aggregated under a single parent. Worst case, we might be dealing with a locally complete subgraph with $500^2 = 25,000$ edges, which we cannot process due to the limited *IH* and which we cannot split anymore because all nodes are aggregated under a single parent.

To avoid these problems we define an internal degree parameter *ID* that indicates the maximum allowed fan out in our hierarchy tree. From our above discussion it should be clear that $ID^2 \leq IH$. Prime candidates for aggregation are the combs we described previously and we therefore aggregate all nodes in the comb under the rootnode of this comb. If this aggregation step leads to a fan out higher than *ID*, we recursively split this cluster into subclusters of a most *ID* nodes. Fig. 2 shows a sample of comb reduction using an *ID* of 4. After depth reduction we find a suitable antichain AC_{IH} by aggregating upward from our memory resident antichain AC_R. The resulting AC_{IH}-view is then passed to the visualization client for clustering.
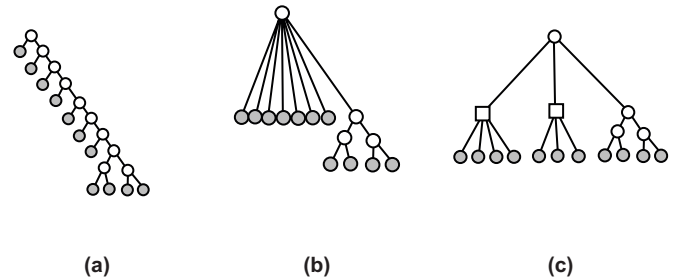

(a)   (b)   (c)

Fig. 2. Compacting binary hierarchies **(a)** Binary hierarchy with a comb; **(b)** All nodes in the comb were contracted with their rootnode and intermediate nodes were deleted; **(c)** Since this might increase the fan-out of the rootnode we recursively bin these nodes under two new nodes.

5   CLUSTERING

In the previous section we used a specialized hierarchy tree to create graph abstractions AC_{IH} and AC_R that allow us to break down the entire graph into more manageable pieces of size at most *IH* edges. In this section we present a sequence of structural clustering operations that we apply to each of these pieces on demand. The hierarchy resulting from this clustering is then used to drive the visual navigation.
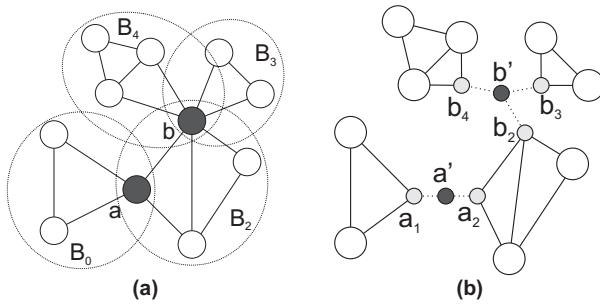
Fig. 3. Effect of separating articulation points: **(a)** input graph with 4 biconnected components and 2 articulation points; **(b)** cloned version, articulation points are split over their respective BCC's. The resulting partitioning is then used as a clustering.

5.1 Preprocessing

From a graph drawing perspective, it does not make much sense to employ expensive algorithms such as force directed algorithms to compute a layout for tree-like parts of the graph. We use a special case of a linear time process known as peeling [3,9,21] to identify peripheral tree-like sections and group them together. If the user decides to expand such a tree group, we can employ much more efficient tree specific algorithms to provide a layout.

We find subtrees in the graph by iteratively identifying nodes with degree one and subsequently decreasing the degree of its neighbor by one. We call the set of trees induced by these nodes the *peripheral forest* of the graph. Since the peeling process can be implemented in time linear on the number of edges [9] we can identify the peripheral forest of G efficiently. If G is connected and not a tree, then each subtree t in the peripheral forest of G is incident to a vertex rt in the complement of the peripheral forest. We then group together all subtrees in the peripheral forest that are incident to rt along with rt itself. Applying peel grouping reduces the amount of vertices that need to be analyzed significantly, and is also meaningful at a structural level.

Biconnected components (BCC's) provide another high level view of the structure of a graph. A BCC is a connected subgraph that cannot be broken into multiple components by removing a single node or edge. The BCC's of a graph form a partition of E and can be computed efficiently in $O(|V|+|E|)$ [23]. Those vertices or edges whose removal disconnects the graph are called articulation points and bridges respectively.

A large number of induced subtrees in a graph corresponds to a large number of BCC's (more precisely one for each vertex in the subtree). However, by first applying tree grouping as outlined above and then applying a biconnected component decompostion to the resulting graph we obtain a substantially smaller number of BCC's without losing essential biconnectivity information, as was proposed in [4] and independently in [6]. However, biconnected components form a partition on the edge set of the graph, where we need a partition of the vertex set. We address this problem in the next section.

5.2 Biconnected component vertex decomposition

The next step is to obtain a vertex partition from the edge partition provided by the biconnectivity decomposition. Simply creating a partition based on biconnected decomposition by assigning each articulation point to a separate cluster might provide undesirable results since this can create a single biconnected component cluster consisting of multiple disconnected components.

The trick is to replace each articulation vertex a that is shared among k biconnected components $B_1, B_2,..., B_k$ by a star centered at a new vertex a' with k spokes, where the ith spoke connects a' to a new 'cloned' vertex named a_i and where the neighborhood of a_i is $a' \cup \{v : v \in B_i \wedge (a, v) \in E\}$. We refer to the a_i's as clones of a and to a' as the central clone of a (see Fig. 3).

Notice that by introducing clones for each articulation point we obtain an enlarged graph G' with more vertices than the original one, but with disjoint BCC's. Deleting the central clones and identifying their associated clone vertices gives us back the original graph. The biconnected decomposition tree of this enlarged graph contains essentially the same biconnected information as the original graph with some redundancy added by the incorporation of the clone vertices.

We can then use G' to represent a clustering of the original graph G by the obtained BCC vertex decomposition. We create one cluster for each BCC and any cloned articulation points in that BCC and one cluster for the central clone of an articulation point. This BCC clustered graph then serves as a top level view from which we can explore the rest of the graph. However, it might very well be possible that the number of nodes in a particular biconnected component B is too large for us to layout or comprehend, that is $|B| > IL$. In that case we have to resort to an off-the-shelf algorithm to compute a clustering, and we apply it recursively to further break down the graph.

5.3 Recursive Clustering

To enforce that our hierarchy has a maximal fan out of IL we recursively break clusters c for which $|children(c)| > IL$ by applying a clustering algorithm of choice (in our case a tuned version of MCL [10]) using the $children(c)$-view as input. When this clustering algorithm returns we have to deal with either one of three cases, depending on the clusters C that were returned:

- $IL < |C| < |children(c)|$: we reapply the clustering algorithm to the C-view of G.
- $1 < |C| \le IL$: In this case we can directly use the clustering returned to us, and create a new node in the hierarchy for each cluster. For each new cluster c that was created we test again if $|children(c)| < IL$ and reapply recursive clustering if necessary.
- $|C| = 1$: In this case the clustering algorithm could not split the subgraph, most likely because it was too dense and we have to resolve to the contraction based hierarchy construction procedure outlined in section 4.34.2.

The whole clustering pipeline (Peeling – BCC – MCL – contraction) outlined above completes in a matter of seconds since we can guarantee that the size of the input subgraph will never be more than IH. The resulting hierarchy on AC_{IH} is then used to drive the visualization.

6 VISUALIZATION

Although some of our previous visualizations for large graphs have mainly focused on matrix oriented representations, we found that in practice these often seemed too abstract and confusing to the novice user. Matrices have the disadvantage that two links that share the same node need not be close together in the adjacency matrix. This makes it particularly hard to identify interesting graph structures, such as subtrees or barrel-like structures, from a matrix representation.

In the GraphView system we opted for a node-link visualization of a clustered graph, similar to the approach described in [12]. The user is allowed to 'open up' clusters of interest, after which the system will embed a more detailed version of that cluster in the current layout.

However, simply allowing the user to successively click open multiple levels quickly leads to disorientation and context loss. The ASK-GraphView system incorporates a number of options to address this problem. Firstly, instead of rendering only the subgraph induced by the clicked cluster [2,3], we keep the entire contextual view of the graph visible at all times. The user is allowed to smoothly zoom in and out of context using the scrollwheel.
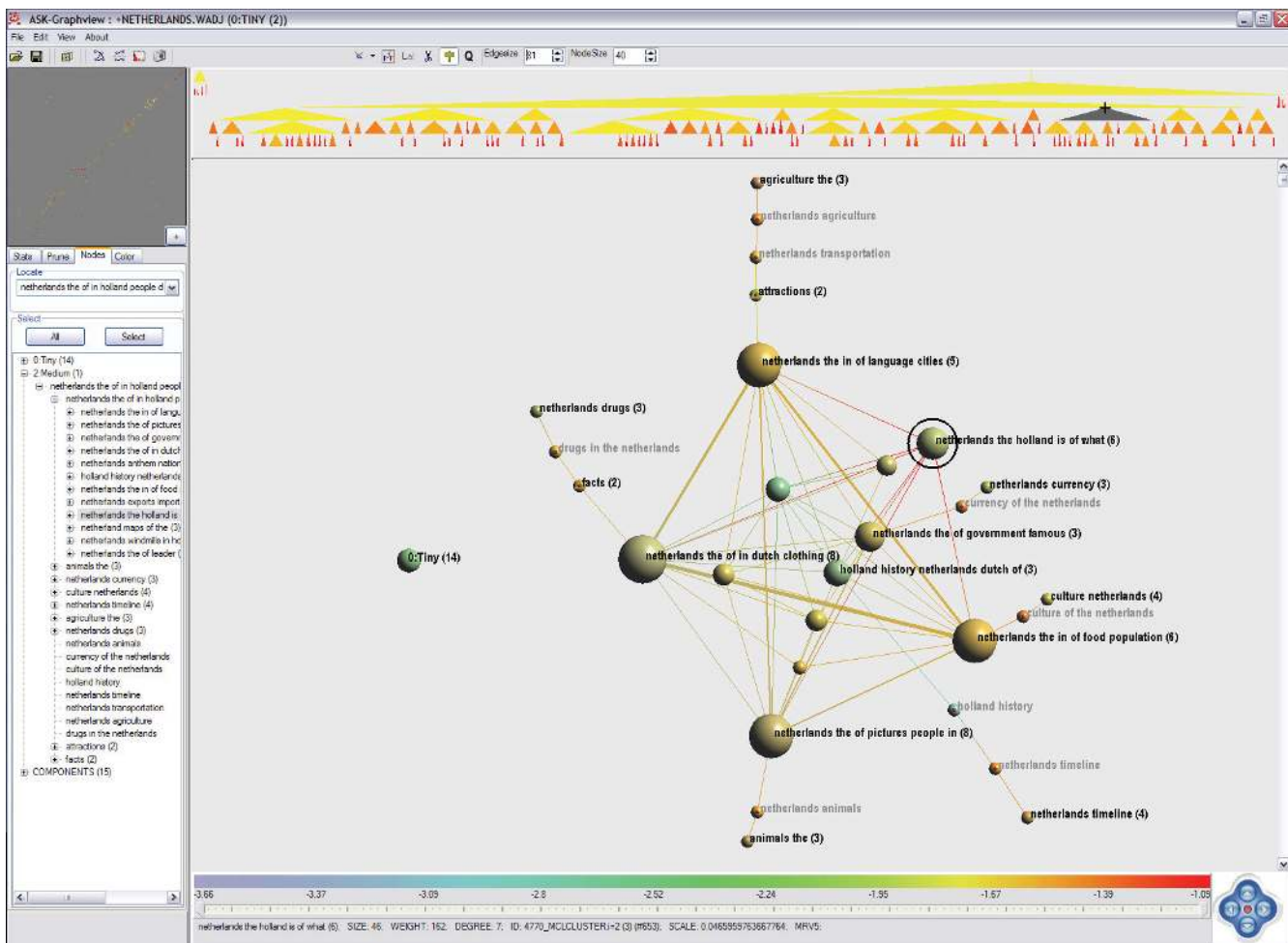
Fig. 4. Screenshot of the ASK-Graphview interface showing small graph (489 nodes) representing information items on *Netherlands*. The treelist is visible on the left, the treeview is at the top of the screen. The grey area in the treeview represents the currently selected node. A matrix representation of the entire graph is visible in top left. The single cluster labeled 'Tiny' represents all smaller connected components. A slider near the bottom of the screen allows quick filtering on a user specified attribute, while the slider at the right edge of the screen allows quick expansion of individual nodes.

Secondly, we provide a traditional textual indented treeview to the side of the main window that lists the labels of the nodes in the hierarchy. To avoid the user having to scroll through the entire list looking for a specific node, we also provide a search box above it. The treeview and searchbox are useful if the user is looking for a specific item in the data, but doesn't know where that item is located in the hierarchy.

Thirdly, we provide a sketch of the entire hierarchy at the top of the main window in the form of an outlined and fisheyed treeview. This representation indicates both the current position of the item the user is currently viewing in the hierarchy, as well as single attribute information on a cluster. We usually display density but other possibilities include the number of times this cluster was browsed or the number of nodes in this cluster matching a particular filter. The treeview also allows random access to any cluster in the hierarchy and the system automatically computes a smooth transition from the current cluster to the selected cluster. These three representations are linked, meaning that operations such as cluster selection, cluster expansion and cluster collapsing can be performed from any of these representations and the others will automatically adapt.

6.1 Interaction

As mentioned previously, the user navigates the clustered graph by iteratively clicking open a cluster C that he or she wants to see in more detail. Or, more formally, at any time during navigation the system is displaying the graph induced by a display antichain, which

we call AC_S. In this case expanding a cluster C amounts to replacing AC_S with $AC_S \cup children(C) - \{C\}$ and collapsing a cluster replaces AC_S with $AC_S - children(parent(C)) \cup \{parent(C)\}$ (see also [3]). To avoid a large number of new links suddenly popping up in the display we animate the expansion by interpolating node positions.

The coordinates of the subgraph induced by the children of C are computed by a force directed layout algorithm. These are then linearly transformed to fit within an area that is dependent on $|leaves(C)|$. In our case we used an optimized [8] force directed layout algorithm for its ability to compute a decent layout of any subgraph with at most IL vertices in reasonable time regardless of the subgraph's structure. It is however very easy to add more algorithms and have the system select the most appropriate one, depending on the graph structure.

To save users from the tedious task of having to successively click open a large number of potentially uninteresting subgraphs we added the option to expand any cluster for which $|leaves(C)| < x$ to its corresponding leaves. The parameter x depends on the desired level of interactivity and the speed of the layout algorithms used, and in our case is set to 150 nodes. Another option lets the user recursively expand subgraphs of a selected node by dragging a slider down to a particular level in the graph. Missing coordinates for subgraphs are then computed on the fly. Since the number of subgraphs that need to be laid out increases exponentially with depth, this feature is only really useful if the user wishes to expand ahead a limited number of levels in the graph.

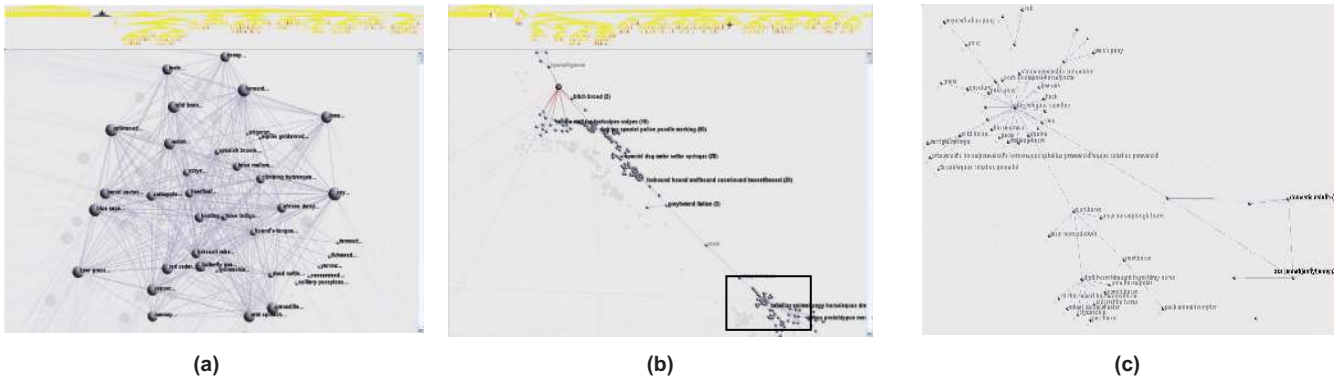**(a)**   **(b)**   **(c)**

Fig. 5. Screen shots from a visualization of the wordnet database [13] totaling approximately 100K vertices: **(a)** Dense high-level cluster showing plants. Note that the user has not (yet) requested detail level data for these nodes, as there are no subtrees below the grey selected node in the treeview at the top; **(b)** Detailed cluster showing *dogs* and *horses*, both are connected through the term *racer*; **(c)** closeup of the highlighted area in **(b)** showing only *horses*. Notice how more labels have become visible.

Another feature is the ability to filter nodes and edges by a parameter. If attribute data is specified with the input data we can filter nodes and edges by setting upper and lower bounds for that attribute either by using a slider directly from the interface or by selecting a range in a window showing the distributions of this attribute. The system then automatically adjusts all of the displays to reflect only the data that falls within this range. Passing all these parameters with the input data instead of computing them internally, allows us great flexibility in assessing the usefulness of different types of computed metrics.

Because node-link visualizations often suffer from on screen edge congestion, we also implemented two solutions to alleviate this problem. The first one makes edges which have both nodes off screen more transparent. Since both of the end nodes from these edges cannot be seen these edges do not add to the display. The other option allows the user to elide all edges from view, except for the ones adjacent to a user specified set of nodes. Since the node layout reflects the clustering of the graph, one can usually deduce the approximate structure from the node density alone (see Fig. 6).

6.2 Context definition

Any usable interactive system dealing with a potentially large number of items on screen will have to provide a mechanism to guarantee interactive performance. Even when using fast hardware accelerated graphics cards, applications often become slow and unwieldy when animating and/or navigating tens of thousands of nodes on screen. Other systems that have been designed for the visualization of large graphs and trees [22,25] include cut-off mechanisms and intelligent pixel level algorithms to keep rendering overhead to acceptable levels. In section 4 we have parameterized the number of nodes that can be rendered at an interactive rate as S. This means that at any time during navigation our display antichain AC_S can have at most S nodes.

If we assume a reasonably balanced hierarchy on a graph of size $|V|$ the average depth of the hierarchy is limited to $d \approx \log(|V|)$ and in practice we can safely say that $d < 20$, especially given the fact that we are aiming for a maximum fan out of IL.

Define the set *context*(u) of a node u in the hierarchy recursively as $children(u) \cup context(parent(u)) - \{u\}$. Note that *context*($u$) defines the minimum maximal antichain that includes the children of u. Each time the expansion of a node x should lead to a number of on-screen nodes that is bigger than S, we replace S with *context*(x). Since the size of *context*(x) is at most $d * IL$ and provided $S > d * IL$ (see section 4.1) we can never have more than S nodes on screen at once. A more intelligent alternative, which also keeps the direct neighbours of a cluster visible, amounts to greedily collapsing nodes in AC_S based on edge weight: Define the union of two antichains A and B as the largest subset of $A \cup B$ which is still an antichain. The set *neighbours*(x) in AC_S is defined as $\{y \in AC_S : e_{xy} \in S\text{-view of } G\}$.

We start out with a set of proposed neighbours $P = neighbours(x)$. While the union of context(x) and the contexts of all nodes in P is bigger than S we replace the node y in P having minimum $w(e_{xy})$ with *parent*(y) and compute a new context. Termination of this loop is guaranteed, since at one point all nodes in the proposed neighbour set will be above *context*(x) and the size of *context*(x) < S. This computation can be performed in RAM in time O(S log(S)).

6.3 Labeling

Another important aspect regarding the usability of the visualization is its ability to automatically assign labels to higher level clusters. Without these labels users are forced to manually inspect the labels of the set of leaf nodes of a cluster every single time to get a rough idea of the data that is contained in that cluster. A straightforward solution could include a weighted *tf-idf* based algorithm, coupled with stemming and a frequency cut-off to avoid misspellings and obscure terms. Here, we incorporated a proprietary algorithm that for a cluster x automatically extracts the log($|leaves(x)|$) most important labels from this potentially large keyword set, based on both node importance and graph structure. These are then displayed in order of computed importance (see Fig. 4).

Note that the general problem of defining a sensible labeling for a cluster is not always easy to solve. For the wordnet [13] graph for example, we would like to label a cluster that contains only plantnames (see Fig. 5a) with the label "plants", but integrating this semantic knowledge into a system is an open problem.

Since displaying a label for every single visible node quickly fills the screen with overlapping (and hence unreadable) labels, we also implemented a level-of-detail algorithm based on the amount of zoom. We recurse through the entire hierarchy tree starting at the root and stop the recursion as soon as the screen size of a cluster falls below a certain pixel threshold. Since the screen size of a cluster depends on the amount of zoom the user has applied this has the effect of incrementally showing more labels as the user zooms in.

7 CONCLUSIONS AND FURTHER WORK

We described ASK-GraphView, a fully implemented system for interactive navigation of large graphs. Its main contribution is an extension of previous work on resource parameterization [3] by also including processing time in the model where previously only memory and screen space were considered. This allows us to visualize external memory graphs with millions of edges at interactive rates. Other contributions include a structural clustering algorithm that improves on [4] and [6] by guaranteeing a bound on the number of subclusters and the ability to break dense clusters if needed, as well as a notion of navigation context that keeps the number of nodes visible on screen below a predetermined constant.
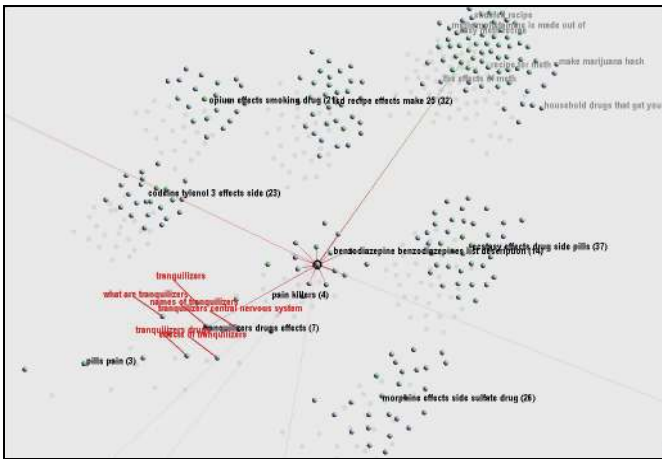
Fig. 6. Low level view of a graph defined on *health* related terms with 86,000 nodes and 2,000,000 edges. The system has marked the term *tranquilizers*. Other clusters found near this cluster are related to specific drugs and have their labels automatically abstracted for readability. Edges are hidden.

7.1   Scalability

Using our system we have been able to interactively navigate graphs with a few million edges with relative ease. More importantly, we do not pose any restrictions on the input graph in terms of structure and density. We even managed to get visualizations of a few graphs with over 2 billion edges by reusing the architecture. When we hit the point where the client ran out of RAM we created a number of separate files for deeper subgraphs below, and have the system automatically load the subsection the user wished to drill into. In terms of scale ASK-GraphView is a substantial improvement over existing systems, which either do not scale to this amount or rely on special structural properties of the graph in question to do so.

7.2   Flexibility

ASK-GraphView was not designed to answer a particular visualization question. Although its main current use is the analysis of relations between search terms, we have also applied it to internet topologies, lexical databases and other types of graphs. Rather than being a highly customized application, it serves as a framework visualization that can answer multiple types of graph-related questions, depending on the attribute parameters that are supplied with the nodes. For example, coloring by an attribute 'density' might be useful if the user is looking for dense subgraphs within the structure. Coloring by an attribute 'number of visits' might entice users to explore areas that are often visited. Although we are currently creating a hierarchy based on the density of subgraphs below (i.e. the system tries not to break dense clusters), we can easily plug in different clustering procedures that allow us to cluster by node attributes or any other graph clustering algorithm for that matter. The system still performs at interactive levels as long as we update our *IL* and *IH* parameters appropriately.

7.3   Usability

Although we have not evaluated our system formally yet, it has been in continuous use by a small number of data analysts over the past six months. During this period interesting requests came up, which we hadn't considered in the initial design and which we think do not surface often in graph visualization applications:

Firstly, the users complained about not being able to annotate certain areas of the graph that they found interesting (for example because of a dense cluster they found or because there were unexpected connections). They would use the system to explore the dataset on one day, only to come back the next day to find that they could not remember where they had spotted interesting features. In

response to this we included the possibility of assigning persistent markers with custom labels to a node (see Fig. 6).

A related request concerned the exploration of a large unknown dataset. Users would be able to navigate around just fine, but had no idea where they should start to look for interesting features. As a result they sometimes stumbled upon something interesting, but spent most of their time randomly browsing the data. We decided to re-use the marker feature and mark the *m* most interesting nodes, where the amount of interest was based on a pre-computed node attribute. Another feature that helped alleviate this problem was the ability to type a random keyword into an edit box and have the system mark all leaf nodes that had that keyword in their label.

7.4   Future work

Future work in the area of large graph navigation should focus on is the automatic computation of a meaningful label for a given cluster. We have used a keyword based method, but meaningful keywords might not always be available.

In terms of architecture, we cannot handle large graphs that come to us in the form of streams, since the architecture that we have presented here requires us to pre-process the entire dataset before being able to visualize it.

REFERENCES

[1]   J. Abello, Hierarchical Graph Maps, Computer and Graphics, Vol . 28, pp. 2004.

[2]   J. Abello and J. Korn, Mgv: a System for Visualizing Massive Multidigraphs, IEEE Transactions on Visualization and Computer Graphics,vol. 8 (1), pp 21–38, 2002.

[3]   J. Abello and F. van Ham, Matrix Zoom: A Visual Interface to Semi-External Graphs, Proc. IEEE Symp. Information Visualization 2004, IEEE CS Press, pp 183-190, 2004.

[4]   J. Abello and F. van Ham, Interactive Navigation of Power Law Graphs, DIMACS TR# 2005-43, May 2005.

[5]   AiSee Graph Visualization Software; http://www.aisee.com/

[6]   D. Archambault, T. Munzner and D. Auber, TopoLayout: Graph Layout by Topological features, Poster Track of the IEEE Symposium on Information Visualization (InfoVis'05), pp 3-4, 2005.

[7]   D. Auber, Tulip : A Huge Graph Visualisation Framework, Graph Drawing Software, Mathematics and Visualization, P. Mutzel and M. Jünger (eds), Springer-Verlag , pp 105-126, 2003.

[8]   J. Barnes and P. Hut, A hierarchical O(N log N) force calculation algorithm. Nature, vol 324(4), pp 44-449, 1986.

[9]   V. Batagelj and A. Mrvar, Pajek - Analysis and Visualization of Large Networks. In M. Jünger, P. Mutzel (Eds.): Graph Drawing Software. Springer, pp 77-103, 2003.

[10] S. van Dongen, Graph Clustering by Flow Simulation, PhD thesis, Universiteit Utrecht, 2000.

[11] P. Eades, and Q.W. Feng, Multilevel Visualization of Clustered Graphs, Proceedings of the 4th Intl. Symp. On Graph Drawing, LCNS 1190, Springer-Verlag, pp 101-112, 1996.

[12] P. Eades, and M.L. Huang, Navigating Clustered Graphs using Force-Directed Methods, Journal of Graph Algorithms and Applications, vol. 4 (3), pp 157-181, 2000.

[13] C. Fellbaum (ed), Wordnet: An Electronical Lexical Database, MIT Press, Cambridge, 1998.

[14] M. Frohlich, and M. Werner, "Demonstration of the interactive Graph Visualization System daVinci", Proc of the DIMACS Workshop on Graph Drawing 1994, LCNS 894, Springer Verlag, pp 15-22, 1995.

[15] E. Gansner, Y. Koren and S. North, Topological Fisheye Views for Visualizing Large Graphs, IEEE Transactions on Visualization and Computer Graphics, vol. 11 (4), pp 457-468, 2005.

[16] M. Ghoniem, J-D. Fekete and P. Castagliola, On the readability of graphs using node-link and matrix-based representations: a controlled experiment and statistical analysis, Information Visualization, vol. 2 (4), pp 114-135, 2005.

[17] F. van Ham and J.J. van Wijk, Interactive Visualization of Small World Graphs, Proc. IEEE Symp. Information Visualization 2004, IEEE CS Press, pp 199-206, 2004.

[18] Y. Koren, L. Carmel and D. Harel, Drawing Huge Graphs by Algebraic Multigrid Optimization, Multiscale Modeling and Simulation, vol 1 (4), pp 645-673, 2003.

[19] T. Munzner, H3: Laying Out Large Directed Graphs in 3D Hyperbolic Space, Proc. IEEE Symp. Information Visualization 1997, IEEE CS Press, pp 2-10, 1997.

[20] D. Schaffer et al., Navigating hierarchically clustered networks through fisheye and full-zoom methods, ACM Transactions on Computer-Human Interaction, vol 3(2), pp 162-188, 1996.

[21] S.B. Seidman, Network Structure and Minimum Degree, Social Networks vol 5 pp 269—287, 1983.

[22] J. Slack, K. Hildebrand and T. Munzner, PRISAD : A Partitioned Rendering Infrastructure for Scalable Accordion Drawing, Proc. IEEE Symp. Information Visualization 2005, IEEE CS Press, pp 41-48, 2005.

[23] R.E. Tarjan, Depth first search and linear graph algorithms. SIAM Journal on Computing, vol 1(2) pp 146-160, 1972.

[24] Tom Saywer Software; http://www.tomsawyer.com

[25] G.J. Wills, Nicheworks – Interactive Visualization of Large Graphs, Journal of Computational and Graphical Statistics, vol 8(2), pp 190-213, 1999.