Interrogative Visualization of Graphs

By Yoav Srebrnik.
Under supervision of Dr. Ariel Shamir.

May 2009.

A partial fulfillment of the requirements for the MSc Degree in Computer-Science from IDC Herzliya.
# Table of Content

1. Introduction  
   - pg 3
2. Related Work  
   2.1. Graph Investigation  
   - pg 4
   2.2. Graph Layout  
   - pg 6
   2.3. Scientometric  
   - pg 8
   2.4. Dynamic Space Data Structures  
   - pg 9
3. Interrogative Operations  
   - pg 10
4. Layout Function  
   4.1. Force Scheme  
   - pg 16
   4.2. Spring's Length  
   - pg 17
   4.3. Spring's Stiffness  
   - pg 17
   4.4. Repulsion Force  
   - pg 18
   4.5. Repulsion Range  
   - pg 19
   4.6. Translating Force to Movement  
   - pg 20
5. Iterative Solver Algorithm  
   - pg 21
6. Speed-Up and Approximation  
   6.1. Scene-Bins Data Structure  
   - pg 24
   6.2. Finding All Feasible Pairs  
   - pg 25
   6.3. Approximating the Repulsion Force  
   - pg 26
   6.4. Computing Local Decay Factor  
   - pg 27
   6.5. Stochastic Update  
   - pg 27
   6.6. Experiments  
   - pg 28
7. System Architecture  
   7.1. Searching Agent  
   - pg 30
   7.2. Graph Creation Agent  
   - pg 31
   7.3. Interrogative Model  
   - pg 31
8. Test Cases  
   8.1. Graph Exploration in Citation Analysis  
   - pg 32
   8.2. Graph Exploration in Social-Networks  
   - pg 37
   8.3. Exploration of Web-Sites Graph  
   - pg 41
9. Conclusion and Future Work  
   - pg 46
10. Reference  
    - pg 47
Appendix  
    - pg 50
1. Introduction

The Graph data structure in computer-sciences is used to represent and model numerous entities. Graphs have been used to model physical networks, virtual relationships between entities, geographical information systems and many more. A graph $G$, is defined as an ordered pair $G=(V,E)$ where $V$ is a set (usually finite) of nodes (usually they are referred to as vertices, but we use the more general term node to distinguish them from vertices of meshes), and $E$ is a set consisting of two element subsets of $V$, which are referred to as edges. A common way to visualize a graph is to draw nodes as circles and edges as segments connecting the two nodes defining the edge.

Gaining insight from a graph-visualization is a complex problem. The user is presented with a drawing of a graph, from which he is supposed to learn and analyze the inner-relations and the graph's attributes, and gain understanding of the metaphor modeled by the graph. Very small graphs are easy to understand: all of the data is visualized, and since the graph is relatively simple, some insights can be drawn only by looking at the image of the graph. That is not the case with large or even medium-size graphs. A network of hundreds of nodes and edges is difficult to understand only by looking at it. The layout algorithm is not enough for making the data visually comprehensible and some exploration must be made in order to understand the data visualized.

Take for example a graph representation of a communication network. The network may have thousands of components attached to it. If the visualization does not differentiate between hubs, switches, firewalls, proxies, servers and pc's, it'll be very difficult for a network architect to understand the network's architecture. Coloring different nodes with different colors may help to understand which node represents which network component. Filtering out the pc's can help in the visualization because the number of components will drop significantly, and the architect will be left with the more important network components.

In our work, we present the interrogative visualization paradigm to graphs. We define and present a set of tools and operations to help interrogate graphs.
interactively. Both the graph and its visualization are constantly-changing, either by incremental insertion of nodes, or by user-initiated actions like filtering and applying some geometrical constraints to the nodes in the graph. These constraints can clarify the relations between different nodes in the graph and allow the user to arrive at some insight. We also suggest other tools for interrogating a graph, like filtering and clustering. Our work follows the different components of the interrogative visualization paradigm [1]:

2. Display – Interactive rendering of the graph.
3. Querying/Interaction – Manipulation and extraction of information from the drawing.

This report is organized as followed: First we present an overview of the previous relevant work. We then discuss the interrogative operations which allow better understanding of the graph. Next, is an extended overview of implementation consideration in force-based graph layout algorithms, and our implementation of a layout algorithm that supports all of our needs presented so far. We describe some of the data structures we use, and provide the system architecture. We finish by describing two test-cases of our system, in the domain of Scientometric.

2. Related Work

2.1 Graph Visualization and Investigation -

Though not focused on graph visualization, a recent work[2] suggests that interaction is a very important yet under-researched topic in the domain of Information Visualization. This work categorizes the techniques of interaction of all InfoVis systems into 7 categories:

- **Select**: mark something as interesting
- **Explore**: show me something else.
- **Reconfigure**: show me a different arrangement
- **Encode**: show me a different representation
- **Abstract/Elaborate**: show me less or more details
- **Filter**: show me something conditionally
- **Connect**: show me related items.
We will address the resemblance and differences between these categories and our Investigation Operation in the next section.

There is a number of previous works, addressing which operations are required in order to efficiently study and explore graphs interactively.

GUESS[3] is a graph exploration system which contains an interpreter for exploration tasks. It encapsulates both visualization and analysis capabilities. The main difference between our system and GUESS, is that our system is more oriented to support nodes insertion/removal. In addition, our system supports adding complex geometrical constraints to the graph, which result in informative layouts.

Herman et al[4] describe in their survey of graph visualization and navigation the types of interactivity desired. They suggest 4 main classes: Zoom & Pan, Focus +Context techniques, Incremental exploration and clustering. Zooming can be geometric (simple blow up of the graph content) or semantic (additional information is provided when approaching a particular area of the graph). Focus + Context techniques are techniques for focusing in a particular area without losing the surrounding context. This can be done, for example, by using Fish-Eye distortion.

In their work on the GGobi system, Swayne, Buja & Lang[5] propose several means to facilitate the exploration and analysis of graphs. They suggest that other than zooming, panning and changing nodes layout, a graph exploration system should allow dragging of nodes, eliminating nodes and collapsing edges. The authors also suggest that the system should allow "brushing" operations, like setting line characteristics and the coloring of nodes by variables, which would be applied to one graph or a pair of linked graphs. They suggest that changes in one graph will affect the other graph.

MatrixExplorer [6] is a social-network visualizer. It uses a dual representation of graph and matrix. In their work on MatrixExplorer, the authors suggested several requirements that every social-network visualizer should fulfill. Those
requirements are mainly based on the requirements of sociologist from such a tool, but they also include a need for interactivity instead of parameter-adjusting, filtering, informative layouts, clustering, and the capability of switching between an aggregated visualization, to a full one.

Pajek [7] is a popular visualization tool aimed at displaying large networks consisting of hundreds to thousands of nodes. It offers the user some strong visualization tools and graph-theory algorithms, like clusters, flows and paths. Pajek was used in the visualization of large genealogies, transportation networks, bibliographies and citation data. Several tools have been presented that create a Pajek input file from an online Data-Base. For example, WoS2Pajek creates an input file from Thomson's 'Web-of-Science' Data Base. Pajek has an extensive layout library and provides implementation to the algorithms of Kamada & Kawai, Fruchterman-Reingold and others. It also provides a mean for adding simple constraints to the layout (optimization of a selected part of the network, fixing some nodes to predefined positions). The main difference of our system from Pajek is that our system provides interactive graph interrogation functionality other than drawing it.

2.2 Graph Layout
A key issue in graph visualization is the layout of the graph. A layout function can be defined as a function that determines the position of each node in $\mathbb{R}^n$:

$$Layout : V \to \mathbb{R}^n.$$  
A survey of graph layouts can be found in [9,10]. Generally, one can categorize the algorithms into two main groups: static and dynamic. Dynamic graph-layout algorithms are algorithms that allow an initial drawing of the graph, and incrementally improve it, or change the position of the nodes while displaying them. The approach mostly used is the Spring-Layout approach, proposed initially by Eades[8]. He suggested reducing the layout problem to a physical system of springs and rings. Later the rings were replaced with electrons so there will be repulsion between any two electrons (or nodes). Initially each node is given a random position in space, and then, in every iteration, it is moved according to the sum of forces (attractions and repulsions) that affect it. The vector-function below describes the force applied on a node $v$:
\[ F(v) = \sum_{(u,v) \in E} \left( k_{uv}^{(1)}(d(p_v, p_u) - l_{uv}) \frac{p_v - p_u}{d(p_v, p_u)} \right) + \sum_{(u,v) \in V \times V} \left( k_{uv}^{(2)} \frac{p_v - p_u}{d(p_v, p_u)(d(p_v, p_u))^2} \right) \]

\( p_u \) is the position (vector coordinates) of node \( u \). \( d(p_v, p_u) \) is the Euclidean distance between the position of nodes \( u \) and \( v \). There are three parameters to this equations: The spring's natural length between nodes \( u \) and \( v \), \( l_{uv} \), the spring's stiffness \( k_{uv}^{(1)} \) which determines the tendency for the spring to be as long as its natural length, and \( k_{uv}^{(2)} \), the strength of the electrical repulsion between \( u \) and \( v \). In most cases those parameters do not vary between different nodes or edges, i.e. \( \forall (u,v) \in E, l_{uv} = l \). It is possible to use other force-functions, like logarithmic springs or other variations of Hooke's law.

As an approximation of the physical world, a key achievement of this model is that the layout of the graph is continuous over time, and there are no "jumps" in the layout between consecutive iterations. The spring system usually converges after 100 iterations or so \[12\]. The drawbacks of the approach are its running time complexity, and the possibility of convergence to a poor local minima. On the other hand, it gives relatively good results in a simple and intuitive manner.

**Static graph-layout algorithms**, which compute the final position for each node, have been developed, and can now allow the drawing of both specific (Trees, DAG's, Orthogonal graphs) and general graphs. They usually approach the problem as an optimization problem in which the optimum is reached when an energy function is minimized (usually locally minimized). The criteria usually use expressions that measure the number of edge crossings, distance between nodes, and uniform edge-length.

Kamada and Kawai[11] proposed a variation to Eades' Spring-algorithm. Their algorithm uses in its computations both graph theoretic distances and Euclidean distances between nodes in the graph. In practice, this means that the attraction force between two nodes is inversely proportional to the length of the shortest
path between those nodes. In this algorithm, attraction exists between non-adjacent nodes. The need to compute graph-distances between every pair of nodes, along with set of the non-linear equations to solve, prevent this algorithm from running incrementally in an efficient manner.

Harel and Davidson[12] solve this optimization problem using Simulated Annealing and achieve good results, but in a relatively long processing time. Similar algorithms were suggested by Fruchterman and Reingold[13], Frick and al[14].

Lately, several works have addressed the layout of incremental-online graphs, where the graph is constantly changing. Tal & Frishman[15] devised an algorithm to be implemented on the GPU supporting thousands of nodes. Their algorithm attempts to keep the graph stability – mental map - by using the last iteration of the layout as input for the new iteration, and animating between different layouts. Similar work has inspired the Ubigraph[16] software, which also uses multi-level approach.

In our work, we chose to use the dynamic graph layout algorithm scheme, since our goal is to provide the user with an interactive graph interrogation tool. Dynamic layout avoids unnecessary long computations. In addition, its simplicity allows us to easily add geometrical constraints to the nodes. This will be elaborated in the following sections.

2.3 Scientometric – Citation Analysis
Our work does not handle the issue of Scientometric directly. Instead, it only uses graphs describing citation analysis and relations between authors as test cases. Therefore, we present here a brief overview of this domain.

Citation data can be used as a science indicator. This statement was proposed in the 1960's by Garfield[17] and Price[18]. Garfield proposed that by evaluation of citations, co-citations and bibliographic coupling, we can learn about the scientific progress of a discipline, key works that were done in that discipline and even map social-relationships in clusters or cliques of authors. Garfield and Small have both
done this independently on large databases, and created clustering and maps of scientific work based on co-citations analysis.

Citation analyses of conferences were performed by Rahm and Thor[19] for Data-Bases conferences, and Jacovi and al.[20] for the CSCW conference. Both groups analyzed the citations and references of works in order to find the most-referenced works, authors and even journals.

Borner et al.[21] wrote a survey of visualization techniques that can be used to map the ever-growing domain structure of scientific disciplines. Other than providing an intensive survey, the authors actually used some of the described methods, like Self-Organizing Maps, ET-maps and VxInsight to visualize the ARISTA database. The main difference between Borner's work and ours is that we address the subject of visualization of online graphs, where Borner works with a static offline database.

Moya-Anegon & al. have proposed a new technique for building maps of large scientific domains[22]. In their work, they compare the different possible methods and propose a schematic approach for the visualization of co-citation graphs. Ma & al.[23] propose, among other things, an agent-based personal article citation assistant. This agent can help find the most related citations of an article in a repository, with collaboration from other agents. Saka & Igami[24] have devised a way to visually map scientific domains using co-citations, which they hope can help in understanding the changes in a scientific domain.

2.4 Dynamic Space Data Structures
In our problem, the nodes are constantly moving around, with the layout algorithm constantly computing their new locations in a 2D domain. As discussed in chapter 6, approximating the repulsion force can be very helpful for a live, interactive program.

There are several well known data structures to help partition a 2D scene. Quad-Trees[25] are used to recursively partition a space into four parts, where each cell has a maximum number of elements. The update process of such a tree is
relatively complicated, especially in the case where many objects are moving at the same time.

Binary-Space-Partition trees[26] (BSP-Trees) are similar to Quad-trees, only partition each cell into 2 smaller cells along arbitrary directions, and are mostly used in Computer-Graphics rendering. Their updates are even more costly than those of Quad-Trees.

There are many other spatial data-structures, like R-Trees and more. A survey of possible data-structures and techniques can be found in [27]. These general data-structures, can be used for many different domains, and offer querying in O(\(\log n\)).

Still, we chose not to use any of the mentioned data-structures, and instead use a simple grid-based structure for several reasons:

1. Update time should be minimal and even O(1) due to the dynamic nature of the interrogation.
2. Due to ever changing nature, accuracy is not a must – the data-structure is used for approximations. Hence, we only need a general gross partitioning of the graph.
3. We were trying to maintain simplicity of the code.

3. Interrogative Operations

In this part we define the operations that are essential for interactive investigation of graphs. Basic interrogative operations ease the process of understanding the data visualized in the graph, especially when visualizing large graphs.

Some of our operations fall into one of the interaction technique categories presented in [2], but others do not agree with any one category and are essentially a combination of techniques, or exist only in graph-visualization. One may suggest that our work lack the growing of the graph (Connect in [2]), but this is not the case, since we define our world as one that is constantly growing in which more and more nodes are endlessly added into the graph (online graphs).

<table>
<thead>
<tr>
<th>Our interrogative graph operations</th>
<th>Techniques of interaction as in [2]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identify</td>
<td>Select</td>
</tr>
<tr>
<td>Pan &amp; Zoom</td>
<td>Explore</td>
</tr>
</tbody>
</table>
Interrogative Visualization of Graphs
Yoav Srebnik

Drag & Drop  Reconfigure
Grouping  Encode
Filtering  Abstract/Elaborate
Geometrical Constraints  Filter
Multi-resolution and Clustering  Connect

Table 1: A comparison of [2] techniques and our online graph investigation operations.

3.1. Identify – Similar to Select in [2]. The basic requirement from a visual-interrogative system is to allow the user to know what the visualization describes. When applying this concept for graphs, it is clear that the user should be able to know (either by clicking, hovering over or a different method) what each node symbolizes.

Figure 1: Hovering over a node pops up a tool tip for identification.

3.2. Pan & Zoom – When visualizing a large graph, that contains numerous nodes, it is essential to allow the user to zoom into interesting regions, and reveal more of the visualized data. Panning is crucial to move around the graph, when zoomed in. This operation falls into 2 categorizes in [2]: Explore and Abstract/Elaborate but we see it as a single basic navigation operation.

3.3. Drag & Drop – There are times when the user knows best how to arrange the nodes in the graph. Hence, an interactive graph visualization system must allow the user to drag & drop nodes around the visualization pane. A graph interrogation system should allow the user to change the layout of the nodes. The system should provide basic generic layout algorithms, but should also
provide the user with the capabilities to change the layout according to his preferences. Agrees with the Reconfigure category in [2].

3.4. Grouping – One of the basic interrogative operations is the grouping of similar or related nodes into groups. This can give great insight of the relations inside a group and between different groups. The most basic way to visually group nodes is to use colors. By applying different colors to the nodes, we are in fact identifying and grouping nodes together and emphasizing the differences between nodes. This operation does not exist as-is in [2].

3.5. Filtering – Taking nodes out of the graph is a sure way to decrease the amount of details in the graph and to reduce cluttering. By doing so, the user can concentrate on the interesting parts, rather than get distracted by irrelevant data.

A graph investigation system should allow the user to filter out nodes based on visual and semantic attributes. Our system allows filtering-out nodes by their groups (colors), or by their in/out degree (authority/survey in the Scientometric domain). In addition, our system allows easy customization of filtering options, which may be dependant on other attributes. For example, in social-network visualization, it may be helpful to allow filtering by a person's sex or age.

There should be a distinction between two types of filtering. A node (or edge) can be *visually filtered*, meaning it is not rendered in the image, but it is a part of the graph calculation for layout purposes, or it could be *logically filtered*, meaning it is not rendered and it also does not affect the calculation of the graph layout. Our system supports both options.
Figure 2 – Grouping using colors and spatial structure.

Figure 3 – Visual Filtering of nodes by their degree (same graph from figure 2).
3.6 Geometrical Constraints – For more elaborate layouts, our system suggests adding geometrical constraints. Using such constraints, the user can visualize the graph in different layouts and control the position of the nodes. We currently allow several geometrical constraints. A user can fix the node's position, preventing it from moving. The user can add a constraint, forcing all or a group of nodes to be laid on a curve (a straight line, a circle, a rectangular) or inside a given region, such as a circle or a rectangle. This operation falls into the Reconfigure category in [2].

Figure 4 – Logical Filtering of nodes by their degree (of graph from figure 2).
3.7 Multi-resolution and Clustering - When handling large graphs, it is sometimes impossible to understand the "big picture" by viewing all nodes simultaneously. It is often very useful to partition the graph into clusters, and then visualize the graph of clusters (each cluster is a node). The partitioning can vary in the number of clusters, number of allowed nodes in each cluster, etc. This process can continue recursively; creating a multi-resolution representation of the graph. At the current stage our system does not support multi-resolution visualization of the graph. These operations are a combination of Reconfigure, Encode and Abstract/Elaborate categories in [2].
4. Layout Function

In graph visualization, the layout is the most influencing attribute of the visualization. It is basically the projection of the mathematical representation into the 2D canvas. Finding the "best" or "most beautiful" layout is a very hard problem, and unnecessary for ever-changing online graphs. It is enough to have a layout algorithm that supports the interrogative operations, allow the online nature of the graph and run with reasonable results and time. Therefore, we decided to use the classic Spring-Layout algorithm as our base layout algorithm and apply many improvements on it (that will be discussed in the next sections).

There are several parameters to the Spring-Layout algorithm, thus there are many possible implementations for it. In this section we examine several parameters and evaluate the strengths and weaknesses of some approaches. We start by examining how the forces are computed in the Spring-Layout model, and end by examining how the forces translate to movement.

4.1 Force Scheme – The original Spring-Layout algorithm, proposed by Eades, modeled the problem as a system of rings substituting nodes, and springs, substituting edges, connecting one ring to the other. This original system had forces only between connected nodes, or in other words, only spring forces:

\[
F_{spring}(v) = \sum_{(u,v) \in E} k_{uv}^{(1)} (d(p_v, p_u) - l_{uv}) \frac{p_v - p_u}{d(p_v, p_u)}
\]

Inherently, this means that no repulsion forces exist between non-adjacent nodes. For a graph that is made out of several strongly-connected components, it is highly plausible that all the different components will overlap. Clearly, this is not a recommended solution for the display of a large graph that is made of several strongly-connected components.

In order to prevent this problem, a new force was added to the system. The system is now composed of electrically-charged elements, instead of rings, with springs attached to them, as defined by the edges. The new force that governs the elements positions for graph \(G=(V,E)\) is:

\[
F_{final}(v) = F_{spring}(v) + F_{repulsion}(v)
\]
\[
F_{\text{repulsion}}(v) = \sum_{(u,v) \in V \times V} \frac{k_{uv}^{(2)}}{(d(p_v, p_u))^2} \frac{P_v - P_u}{d(p_v, p_u)}
\]

We will discuss several issues in the final force equation.

4.2 Spring's Length – As in the real world, the springs in the Spring-Layout model come in different sizes and attributes. Some springs allow longer distance between the nodes than other springs do. This behavior is determined by the \( l_{uv} \) parameter, the spring's desired length. If this value is constant for all nodes, then the springs will tend to be in the equal length.

In some cases, it may be useful to model \( l_{uv} \) in inverse-proportion to the weight of the edge \((u,v)\) in a weighted graph. The result may be that nodes which share a "strong" edge are relatively closer than a pair of nodes sharing a "weak" edge.

4.3 Spring's stiffness – The parameter \( k_{uv}^{(1)} \) is used to define the stiffness of the spring. The stiffness is a parameter affecting how much force the spring imposes on its adjacent nodes in order to return to its desired length. If \( k_{uv}^{(1)} \) is constant, then all the springs are exactly the same (stiffness-wise). If \( k_{uv}^{(1)} \) is smaller than 1, then the force imposed by the spring, is smaller. This, in turn, allows the spring's length to be longer than \( l_{uv} \).

The spring's stiffness can vary between different edges/springs. In some cases it may be useful to set different stiffness values according to a specific function. Remember that the higher the stiffness value is, the larger the force that affects the nodes. Lets review 2 scenarios where this parameter is useful:

**Spring's stiffness as a function of nodes' age** – Let's assume that the graph in hand is a result of a BFS algorithm traversal from a given node, and every node is assigned an integer that describes the graph-distance of this node from the root node. This is essentially a layered-graph. If we set the spring-stiffness value to be inversely proportional to the smaller age between the two nodes of each edge, the graph will be laid with longer springs next to the root, and shorter springs in the outer-most layers. This is useful when we want the attention of the user to be directed at the root node of the BFS, and its close environment.
Spring's stiffness as a function of nodes' degree – When exploring a graph, we usually give greater consideration to "important" nodes. We can define a node as important when its in-or-and-out degree is higher than a threshold value. When we relate the stiffness parameter to node degree, we can make the important nodes more stable in the layout than other nodes. Another effect of this approach is that springs connected to "important" nodes will tend to be longer. Two effective ways to implement this relation between node degrees and spring stiffness are formulated here:

\[ k_{uv}^{(1)} = \alpha^{\text{deg}(u)+\text{deg}(v)-2} \]

\[ k_{uv}^{(1)} = \alpha^{\text{deg}(v)-1} \], where \( 0 < \alpha \leq 1 \) in both implementations, making the springs less rigid.

In the first implementation the spring force is symmetrical on both nodes connected to the spring, while the latter is a detour from the physical world model because different forces are activated on each of the two nodes.

4.4 Repulsion Force – In the real world, the electric repulsion force is dependent on the distance between the charged elements. This is beneficial for a graph layout model as well, because we would like to have the repulsion influence closer nodes more than distant nodes. We formulate the repulsion force as followed:

\[ F_{\text{repulsion}}(v) = \sum_{(u,v) \in V \times V} \frac{k_{uv}^{(2)}}{(d(p_v, p_u))^2} \cdot \frac{p_v - p_u}{d(p_v, p_u)} \]
Like the spring-force, the repulsion-force parameter, $k^{(2)}_{uv}$, can be either a constant or a function of other parameters. Setting $k^{(2)}_{uv}$ to have a constant value is equivalent to the situation in which all nodes have the same electric charge (same in size and sign).

Given the scenarios described in (4.3), we can gain by choosing $k^{(2)}_{uv}$ to be a function of some parameters. In this scenario we run a BFS from a source node, and assign age values to each node, we can get an output layout where the source node is much spaced, while the nodes with higher age value are more crowded.

$$k^{(2)}_{uv} = \frac{1}{\text{age}(v) + 1}$$

In the scenario in which we would like to have "important" nodes more spaced than less important nodes we can have: $k^{(2)}_{uv} = \text{const} \cdot \text{deg}(v)$.

Up until now, we only saw "unnatural" repulsion force. The natural force (Columb's law) is depended on both electric charges, while a single charge only creates an electric field. $F_{Columb's} = \frac{1}{4 \pi \varepsilon_0} \frac{q_1 q_2}{r^2}$. We can model this attribute and then have important nodes laid out distant from each other. $k^{(2)}_{uv} = \text{const} \cdot \text{deg}(u) \cdot \text{deg}(v)$.

**Figure 9 – Repulsion as factor of node's degree. High degree nodes have greater repulsion.**

**4.5 Repulsion Range** – Let's assume that a graph has already been drawn, and a new node is inserted into the graph. In a case where the new node is assigned new
coordinates outside the convex-hull of the original graph, the new node will suffer from repulsion forces originating from all the nodes in the graph, pushing the new node in the same direction, away from the graph. Unless the springs connecting the new node are relatively powerful, the node will remain far from its adjacent nodes. Moreover, in a case when the graph is made out of several strongly-connected components, they can push each other indefinitely.

A repulsion force that is inversely proportional to the square distance between the nodes, limits the effect described in the first scenario, but not the second. Choosing a threshold distance, from which no electrical repulsion force exists, is a solution for both scenarios:

\[
F_{\text{repulsion}}(u,v) = \begin{cases} 
0 & d(u,v) > \text{threshold} \\
F'_{\text{repulsion}}(u,v) & \text{otherwise}
\end{cases}
\]

Making the threshold for electric-repulsion force a function of some parameters, can also be very useful, for example, when we want to attract attention to specific nodes in the graph. The graph drawn will have more space around important nodes, whether they are defined by their age (Layered graphs) or by their degree. We can set a dynamic repulsion range using the following:

\[
\text{threshold}(u,v) = \text{min_repulsion_range} + \deg(u) \cdot \text{radius}
\]

where \text{Min_repulsion_range} and \text{radius} are constants.

4.6 Translating Force to Movement - By now, we discussed how to formulate the forces among the nodes and springs, but one main question remains: How does the force translate to the movement of a node? These movements will result in the dynamic graph layout visualization. We describe two possible techniques:

4.6.1 Direct force-movement translation – This approach directly translates the sum of forces on the node to its movement. \( \vec{p}_x(v) = p_x(v) + F_x(v) \). A similar equation exists for the y-component. In this approach all the nodes move only by the forces activated upon the node in the last iteration. One can view this translation as if each iteration consists of 3 stages: "activating" the system and calculating the sum of forces, moving the node accordingly and then "shutting-down" the system.

4.6.2 Force-Speed-Movement - Another, more realistic, approach is to distinguish between force and speed. A force is activated on a node and affects its speed.
(speed/size and direction). In turn, the speed controls the next location of the node.
\[ \ddot{v}_s(v) = v_s(v) + F_s(v); \quad \ddot{p}_s(v) = p_s(v) + \ddot{v}_s(v). \]

It is sometimes useful to add friction to the system, in order to eliminate a situation that a node, which once had forces affecting it but now does not, moves indefinitely. This can easily be done by adding a damping parameter \( \alpha \) to the speed equation:

\[ \ddot{v}_s(v) = \alpha v_s(v) + F_s(v), \]

where \( 0 < \alpha < 1. \)

5 Iterative Solver Algorithm

As previously mentioned, we are not interested in finding the best layout in term of number of edge-crossing or other heuristic attribute. We require a layout algorithm that will support the interaction and investigation operations, while handling with the online, ever-changing nature of the graph with reasonable time and esthetic results.

We base our layout algorithm on the popular force-based approach. A variation of the basic algorithm will allow handling large graphs by solving the layout problem locally among other improvements.

Our algorithm is a variation of the Spring Layout algorithm. The graph is reduced into a system of electrically charged rings, connected to each other by springs. Let \( A(edge, node) \) be the attraction-force function, and \( R(node, node) \) the repulsion-force function. The algorithm works as followed:

```
Online-Constrained Layout (Graph G)
1. Update the accumulated forces of each node.
2. Accumulate forces.
   2.1. for each spring(edge)in G, e = (u,v),
       2.1.1. F_total(u) = F_total(u) + A(e, u);
       2.1.2. F_total(v) = F_total(v) + A(e, v).
   2.2. for each ring (node) v in G,
       2.2.1. pair_list = Calculate_Feasible_Pairs(v)
       2.2.2. for each pair p=(u,v) in pair_list
           2.2.2.1. F_total(u) = F_total(u) + R(u,v)
3. for each node v
   3.1. new_coordinates(v) =
       Movement_Function(v.coordinates, F_total(v))
   3.2. If node is limited to constraint C
```
3.2.1. \( v\text{.coordinates} = \text{getClosestLegalPoint}(C, v, \text{new\_coordinates}(v)) \)

Else

3.2.2. \( v\text{.coordinates} = \text{new\_coordinates}(v) \)

Note that our algorithm distributes the forces, and does not gather them.

5.1 Constraint Satisfaction – This is the core of the constrained layout mechanism. Each constraint implicitly defines a position, such as all nodes must be on a curve, or in a specific region. The function "getClosestLegalPoint", gets the current position of the node and its constraint, and projects the point on to the closest new coordinates of the node, so that the coordinates are consistent with the node's constraint.

We implement this function as projection of the node to some geometric primitive – a line, a curve, a region, a point, etc.

5.2 Decay Factor: As briefly mentioned before, there are many variations to the force-based graph layout scheme: the new coordinates can depend on the force activated on the node in earlier iterations (simulating acceleration), or only in the current iteration. The computation of new location can also be relative to some global step-width, like in Reingold-Fruchterman[12].

In our implementation, there is a dependency on the forces of the last iteration. If the size of the new sum-of-forces-vector is bigger than some constant value, the vector is clipped to be in the maximum step-width.

\[
\|F(v)\| > \text{maximum\_step\_width} \begin{cases} \text{no} & F(v) \\ \text{yes} & \hat{F}(v) \times \text{maximum\_step\_width} \end{cases}
\]

To reduce the number of oscillating nodes, we compute a decay factor for each region of the graph. Active regions will have \( \text{decay\_factor} = 1 \) while static regions will have \( \text{decay\_factor} \rightarrow 0 \). By doing so, and multiplying the sum-of-forces-vector of each node, by its node's \( \text{decay\_factor} \), we actually get a local effect similar to Simulated Annealing. \( F_{\text{final}}(v) = F_{\text{total}} \times \text{decay\_factor}(v) \)
The local decay factor is computed based on the SceneBin data structure that will be presented shortly.

5.3 **Oscillating nodes** - In some cases, a node may not find its final position even after more than 100 iterations, and will alternate between several positions indefinitely. The main reason for this phenomenon is that the movement calculation is not continuous, but discrete. In this section we will discuss why this case is possible, and the ways to eliminate or minimize the re-occurrences of such cases, which resemble harmonic motion in Physics.

5.3.1 **Harmonic Motion in Attraction-Only models** - Un-positioned nodes can occur even in a model with only a pair of nodes connected by an edge, when there is no repulsion force between them. Let's assume that in iteration $i$ the nodes are too far away. In the following iteration, each node will move toward the other. This may result in the nodes being too close to each other, so now the spring will cause the nodes to move away from each other. If the forces are very gross and the movement is relatively large, the nodes can always miss their equilibrium point.

5.3.2 **Harmonic Motion in Repulsion-Only models** - This situation is less common and usually occurs when some nodes are static – either by a constraint or by high node degree which fix their position. Consider the following case: three nodes, $a$, $b$, and $c$ are position on a straight line respectively. Nodes $a$ and $c$ are static for some reason, and node $b$ is closer to $a$ than to $c$. If the repulsion forces from $a$ and $c$ are strong, node $b$ might find itself jumping from the vicinity of node $a$ to the vicinity of node $c$ indefinitely.

As presented, Harmonic Motion can occur both in attraction only, and repulsion only models, therefore it's clear that it may occur in a model containing both.

5.3.3 **Solving Harmonic Motion** - As mentioned before, this problem originates from the non-continuous nature of the layout algorithm. We present some possible solutions:

- Approximating a continuous layout algorithm by using smaller steps – Determining a maximum distance for jumps is one option to deal with oscillation. When this value is small enough, the movement will be closer to continuous and the nodes will have a better chance to find their equilibrium point. One disadvantage of this solution is the global movement is much slower than before.
• Low-force systems – By having high force parameters for repulsion and attraction, the nodes are highly charged with energy, and tend to jump from place to place. By reducing the force acting on each node, the system becomes closer to continuous as the movement is less jittery, and uses smaller steps. Unfortunately, this solution also lengthens the number of iteration to reach equilibrium.

• Simulated Annealing (Harel & Davidson[11] or Fruchterman & Reingold[12]) – Another solution, taken from Simulated Annealing, is to steadily reduce the force parameters (or similarly, the maximum jump size of a node, respectively) as a function of time. This approach solves the problem efficiently without over lengthening the number of iteration to reach the final layout. A disadvantage of this approach is that in interactive programs, the "relative time" issue is not well defined.

• Movement lower threshold – By determining a threshold on the minimum movement, we can eliminate most of the Harmonic Motion occurrences. If the step determined by the layout-algorithm is less than some value, the node does not move. This means the nodes will not necessarily reach their optimal position. Another problem that arises from this solution is how to determine the threshold value. In order for the threshold to have meaning, it should be a function of some visualization parameters like the desired edge length or Bounding-Box size.

6 Speed-Up and Approximations
In this part we describe a set of suggested speed-ups and approximation. Each speed-up can be used with or without the other speed-ups, but most of them rely on the SceneBin data structure.

6.1 The "SceneBin" Data Structure- Theoretically, the repulsion force should be calculated between every pair of nodes in the graph. This will result in a running time complexity of O(v^2), which is too slow for large graphs. The SceneBin data structure allows the repulsion to be computed only between certain subset of pairs, thus reducing run-time. Our goal in using this data structure is to reduce the number of overall queries, and not the amount of time each query takes, which is the usual approach taken by spatial data structures. The computation time of repulsion force
between a single pair is relatively fast, and this structure is essential in avoiding O(v^2) time in the update procedure of the layout. The SceneBin is based on familiar data structures like QuadTrees and Grids, with some variations.

The 2-dimensional scene is divided into a number of bins. All the bins have uniform size, and together they form a grid of bins. The SceneBin data structure also holds the graph's ever-updating bounding-box. Nodes are assigned to a bin according to their relative position inside the graph's bounding-box. Each bin contains a hash-table of all the nodes it contains. Each node holds the bin number it currently belongs to. After changing position of a node, we can update the SceneBin structure in O(1).

6.2 Finding all feasible pairs—The first use of the SceneBin is to efficiently find the feasible pairs of nodes for repulsion force computation. For a given node v, when we want to find its relevant pairs for the repulsion calculation (see step 2.2.1 in the layout algorithm), we consider all the nodes in the same bin as v, and those in adjacent bins, according to v's repulsion force range. Notice that this approach can only work if the repulsion has a finite range.

```
Calculate_Feasible_Pairs(Node v) : Pairs of Nodes
1. pairs <- null
2. currBin = getBin(v)
3. range = Force_Range(v)
4. bins_list = Possible_Adjacent_Bin(currBin, range)
5. for each bin in bins_list
   5.1. for each node u in bin
      5.1.1. pairs <- pair (u, v)
6. return pairs
```

In our experiments, the SceneBin affects the run-time and interactivity of the program. The results show that SceneBin is much more influential in large graphs. In the experiment we performed, we counted the number of times the repulsion force was computed for the whole graph. This gave us the number of feasible pairs computed using the SceneBin.

This experiment was done on a graph using 16 bins (a 4x4 SceneBin).
<table>
<thead>
<tr>
<th># of Nodes</th>
<th>max number of iterations ((V^2))</th>
<th>number of iterations using SceneBin</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>25</td>
<td>18</td>
<td>0.72</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>60</td>
<td>0.6</td>
</tr>
<tr>
<td>20</td>
<td>400</td>
<td>260</td>
<td>0.65</td>
</tr>
<tr>
<td>50</td>
<td>2500</td>
<td>1498</td>
<td>0.5992</td>
</tr>
<tr>
<td>75</td>
<td>5625</td>
<td>2887</td>
<td>0.513244</td>
</tr>
<tr>
<td>100</td>
<td>10000</td>
<td>2752</td>
<td>0.2752</td>
</tr>
<tr>
<td>150</td>
<td>22500</td>
<td>3548</td>
<td>0.157689</td>
</tr>
<tr>
<td>201</td>
<td>40401</td>
<td>5889</td>
<td>0.145764</td>
</tr>
<tr>
<td>250</td>
<td>62500</td>
<td>9565</td>
<td>0.15304</td>
</tr>
<tr>
<td>300</td>
<td>90000</td>
<td>14336</td>
<td>0.159289</td>
</tr>
</tbody>
</table>

6.3 Approximating the repulsion force – This approach was motivated by Frishman and Tal[14]. Similar to the previous algorithm, we compute the repulsion force between each pair of nodes in the same bin. However, for other bins, we approximate the repulsion force they activate by calculating the repulsion force between the node and the center of gravity (COG) of each bin.

In this approach, the maximum number of pairs for each node, is the number of nodes of its bin, plus the number of bins in the scene. If we assume that the nodes are spread uniformly in the scene we get the following theoretical result:

\[
\text{num\ of\ pairs} = \left(\frac{\text{number\ of\ vertices}}{\text{number\ of\ bins}} + (\text{number\ of\ bins} - 1)\right)^2.
\]

For graph of 300 nodes and a 4*4 SceneBin, the theoretical number of iterations, if the nodes are uniformly distributed, is 1208. This is a significant speed-up from the results in the table above, which is ~14,000.

Note also that this approach does not need to assume a finite repulsion force range.

Naturally, the method's main drawback is the error caused by the approximation. In practice, this approach tends to cluster the nodes in each bin together. In addition, when the nodes are not spread uniformly in the scene, the speed-up is less significant.

This approximation can be taken even further, when repulsion is calculated only between each node and the COG's of all the bins, resulting in O(E) + O(Num of bins) complexity for the whole layout iteration.

Computing the COG is done incrementally – when a node enters or leaves the bin, the bin's COG is updated. This is done in O(1) operations, since each bin also holds the sum of all its members coordinates. Updating the COG is a matter of adding a coordinate or subtracting a coordinate from the sum.
6.4 Computing local decay factor – As stated in (5.2), it's useful to add a decay factor to the layout. By doing so locally, our algorithm is in fact similar to a local Simulated Annealing algorithm.

Using the SceneBin for computing the decay factor is very useful, because we can update each bin's decay factor upon the first insertion/removal of one of its nodes. Each bin holds an integer number in the range [1-1,000] representing the number of iterations the bin has been left unchanged. When a node enters/leaves the bin this integer is reset to 1, otherwise, in each iteration this integer is incremented until it reaches 1,000, when the decay factor is set to 0. The decay factor of each node is:

\[
decay\_factor(v) = \frac{1}{\text{num\_bin\_unchanged(get\_bin(v))}}
\]

The movement vector of each node will be multiplied with this value, resulting in the final movement vector.

6.5 Stochastic Update – In a graph of thousands of elements (nodes and edges), the user does not usually pay attention to each element, because the user is either interested in the bigger picture, or the graph is too rich with details. Taking this into consideration, we can speed-up the layout algorithm by not updating all the nodes in each iteration. We present here two approaches for implementing stochastic update speed-up.

**Repulsion Force Calculation Stochastic Update** - The repulsion-force calculation is the most time consuming stage of the layout algorithm, so it seems logical to insert the stochastic update to this stage only. This can be done by "tossing a dice" to decide whether a node will take place in the calculations or not. The most straightforward way of doing so, is to decide on a probability-threshold of a node to participate in the calculations. For example, assuming we have a uniform-distribution random number generator, we generate a random value for each node, and compare it to the threshold that was decided earlier.

In our experiments we discovered that if the probability of a node to participate in the computation was less that 80%, the layout was unstable.

**Complete Layout Algorithm Stochastic Update** - The obvious drawback of the previous approach is that it conflicts with the logic of the layout algorithm. It causes a node to have an effect in respect of attraction, but not with repulsion. This drawback makes the repulsion force, in effect, weaker. The second stochastic update approach tries to overcome this drawback.
In this approach, we decide in each iteration which nodes will be affected in the iteration and only these nodes participate in both repulsion and attraction calculations. By doing so we actually perform a random "Logical Filtering" in each iteration (see 3.5).

These two approaches suffer from similar problems. The results are considerably better with the original repulsion-force computation algorithm, than with the repulsion force computation speed-up proposed in 6.3. In addition, in a naïve implementation, where every two nodes have equal probability of not being a part of the computation, the layout can become very unstable when "important" nodes are ignored during one layout iteration, while on the next iteration they take part in the calculation. This effect is even strengthened if the repulsion/attraction forces are non-uniform and depend on variables like node degree.

On the other hand, these approaches provide a speed-up of 10-20%, without severely damaging the graph layout and its stability. We believe that the stochastic update approaches can be more useful if the distribution is non-uniform, meaning the probability of a node to be chosen is dependent on its degree.

**6.6 Experiment** – We compare run-times of 4 different implementations:

1. Eades – An implementation of the original algorithm.
2. Stochastic 1 – Stochastic update of the repulsion force calculation over the original Eades algorithm.
4. Approximation of repulsion using the SceneBins using only COG for repulsion calculations.

We compared the average iteration run-time for updating the nodes position using the four algorithms. Two different graph types were tested – A tree and a general graph. The algorithms were also compared with respect to the number of nodes in the graph.
Although the repulsion-force approximation using the SceneBins gives the best speed-up and transforms the problem from polynomial to linear to the number of edges, we
can only recommend using this approximation for a coarse layout estimation, or using it in conjunction with geometrical constraints, due to the aesthetic features of the resulting layout. This is similar to what Frishman & Tal have reported[14], while clustering the graph and then running the layout algorithm on the coarsen graph.

The Stochastic 2 speed-up gave an expected 20% speed-up for the tree, and about 10% for the general graph. Stochastic 1 speed-up only worked in the tree scenario, probably because the graph had a high average node degree.

In a different experiment, we found that using Stochastic 2 with the SceneBins structure aiding to find feasible pairs for repulsion-force calculations (6.2), gave us the best looking graph, with a speed-up of about 30% from the naïve Eades implementation. The additional speed-up of finding feasible pairs fast was the result of setting a maximum range for the repulsion-force, thus significantly reducing the number of calculations, and allowing quick retrieval of potential candidates.

Figure 12: Average runtime for a tree layout iteration. Comparison between Eades, Stochastic 2 and Stochastic 2 with find feasible pairs speed-up.

7 System Architecture
Our framework is intended to be a framework for displaying and interrogating online constantly-changing constrained graphs. The changes occur either because of user actions or because of the graph changing (adding more nodes or removing nodes e.g.
when clustering or filtering). The framework consists of three modules: Searching agent, Graph Creation Agent and the Visualization module (see figure 13).

7.1 **Searching Agent** – The Searching Agent is responsible for retrieving information from a database. This can be any sort of data source – a file, an internet site, a relational DB etc. The agent is responsible for running a BFS-like search on the data source. This search produces the nodes for the graph, and the edges between them. The agent places all those elements in a shared queue, for the Graph Creation Agent to work on. The agent is implemented as a separate thread, so low response rate from the data source will not affect the visualization and interactive characteristics of the system.

7.2 **Graph Creation Agent** – This agent is responsible of retrieving the elements from the shared queue, and then building the actual graph's data-structure accordingly. The agent first adds all the new node-elements by creating a graph node for each one. Then it adds directed edges between the newly added nodes and any existing nodes in the graph according to the data in the queue. While creating the nodes and the edges objects, the agent adds all additional information that will be relevant for the Visualization Module, like vertex age/layer. The edges that are not linked to any existing node are stored in a list that is scanned after the agent adds new nodes.

7.3 **Interrogative Module** – This module is responsible for drawing, interacting with the visual representation of the graph and determining the layout of the graph. We used the basic drawing capabilities and data structures are from the JUNG package. JUNG[27] is an open-source java project for visualizing network and graphs. The layout algorithm is novel with some reference to JUNG's implementation of Spring Layout. It is implemented as a thread, and updates the nodes' coordinates accordingly. In our case, the thread continues to work indefinitely, so we can support user-interactivity and immediate response to any dragging, filtering or any other parameter changing event.

Our Visualization module supports all the graph interrogative operations that were described in part 3, apart from multi-resolution operations (like clustering that will be implemented in the future). The Interrogative module also supports on-line changes to both attraction and repulsion function parameters of the layout algorithm.
8 Test Cases: Graph Investigation in Scientometric domain.

In this part we describe two test cases we used to examine our framework, which are relevant to the Scientometric domain. Lastly, we show another test case of simple web-crawling.

8.1 Graph Exploration in Citation Analysis- By analyzing the graph of co-citations between essays, we can find essays that are "Authorities" – i.e. which are most referenced or reviewed. This tool allows the user who wants to learn a new scientific topic, to find the most relevant and important works in a topic and map the field of research.

For this test, we implemented a Search Agent which searches the CiteSeer database[28]. CiteSeer is a project of Penn State University, which indexes millions of articles in Computer-Science, and their citations. Our agent also allows the user to search for essays by keywords, using CiteSeer's searching capabilities. These can then be used as starting points to build the graph.

The following diagram shows the suggested flow use:
More insight can be reached by examining the graph created. For example, the graph may be used in order to map new research domains and "place" them at the barrier between two "old" research domains. Running more complex algorithms on the graph will produce more insight on the domain.

We believe that our system can provide great help for a researcher. It is a common problem in science to find out the most innovative and important works. Our system helps the scientist in his first steps of research. Let's have a look in an example.

Let's assume we are interested in learning the topic of Stereo Vision. In order to do so, we need to find out which scientific works are most important in this field of study. We also need to find some good surveys, to provide us with general knowledge of this field. Lastly, we would like to know what are the latest works and achievements in this field.

We start off by entering a search query to our application: "stereo vision image reconstruction analysis". We then insert the best results from CiteSeerX to our system as starting nodes. We let the system perform a BFS search, and expand the presented graph until it reaches the size of several hundred nodes. We can now start investigating it.

We start investigating by hovering over nodes in the graph. This identification process allows us to identify works that are irrelevant for our questioned research topic. Usually, these works appear in the search graph because they were referenced as a general science problem that could benefit from the essay's results.
After identifying the irrelevant works, we quickly group them together, and filter them out. Having an operator that gets a selected node, and signals all the nodes that are connected to the original node is very helpful in this process. In the end of this process, we group all those works into a group of "Highly related" publications by coloring them with the same color.

In the next stage, we filter the nodes according to their inward degrees. This is a clear indication of the "authority" level of the works. Intersection between the most referenced and the most relevant works can be done visually, since we grouped the relevant works together with a single color. In this investigation we found out that the four most referenced and related works are by Tomasi & Kanade, Belhumer & Mumford, Faugeras and Longuet-Higgins.

Now that the relevant essays are flagged, we can easily find the best surveys by filtering out nodes according to their outward degree, and intersecting them with the group of related works. In addition, sorting the relevant works by their publication year can provide us with some insight of how this research topic has developed over the years.
Interrogative Visualization of Graphs

Yoav Srebnik
Interrogative Visualization of Graphs

Yoav Srebnik
8.2 Graph Exploration in Social-Networks (Co-Authors) – Displaying and analyzing social networks is a subject of much interest. We chose scientific author networks. We can find who the scientist's favorite colleagues are, whether the scientist prefers working alone, in a small or large teams etc.

In our implementation of Social-Network visualization, we used the DBLP database[29] for data of co-authors work. Each node represents an author, and an edge will appear between two authors if they have published a work together.

Using this tool we easily found out "social scientists" who regularly work with many other scientists. We also saw a proof to the "Small-World network phenomenon" in which the social-network is made out of cliques connected to each other by "socially active" members, creating a network in which most nodes are not neighbors of each other, but can be reached from every node in a relative small number of hops/steps.
Figure 20: Social networks of an author (in red). One can see the different cliques the author is a member of.

One does not need to have a graph visualization system to find nodes of high in or out degree, group together cliques, perform clustering etc. But the user needs to know a priori the different attributes of the data, so he can activate the different graph algorithms. For example, by examining the image produced by the visualization system, the user can discover that the graph is indeed made out of cliques, that there are several nodes with a relatively high degree, and that some nodes have similar connectivity, and therefore can be clustered into one node.
Interrogative Visualization of Graphs

Yoav Srebnik
Figure 21-25: Displaying an online graph of authors, using as root Prof. Ayellet Tal. A comparison between an approximation and a full layout, a clustered graph demonstrating the "small-world network" phenomenon and a tree layout, similar to Erdos number display.

8.3 Exploration of web-sites graph

Although this test case is not related to the Scientometric domain, it was fairly simple to support this test-case using our application and code. We provide a basic web-crawler that creates an online, ever growing graph, and all the functionality to explore it.
Interrogative Visualization of Graphs

Yoav Srebnik
Figure 26-33: Using a web-crawler to create an expanding graph and coloring it. Using the semantic option of "get connected vertices" to explore the graph from the root vertex to all sub-vertices. A comparison between a slow radial layout, to a fast approximation of a radial layout using the force-spring scheme with constraints and SceneBin approximation.
9. Conclusion and Future Work

One of the greatest challenges today is to know what to search. If an investigator wants to find cliques in a graph, he does not need to have a visualization system; he can just run an algorithm on the graph representation. The hardest part is to know to look for cliques in a graph. A graph visualization system with investigative operations helps in finding out which questions to ask.

In this work, we presented the "Investigative Visualization" paradigm for graphs. We proposed several actions and key-operations for the Man-Machine interaction that support this paradigm and compare them to previous works. We demonstrated how a simple layout algorithm, with some improvements can support exploration and investigation of medium-to-large online graphs with good running time and esthetic results. We discussed how the layout algorithm parameters can contribute to the graph's understanding. We showed how smart semantic operations, such as laying geometrical constraints and filtering can contribute to the understanding of the graph. We implemented a system that is based on this paradigm and examined several issues in its implementation. We also proposed a grid-like data-structure that facilitates many of the operations relevant to the layout algorithm.

In the future, we would like to add to our system some smart visualization features. We would like to provide a smart filtering option, which will allow the user to filter the graph by more than one field. We would like to support drawing different shapes of nodes in respect to their type. We would also like to support smarter layout algorithms, which support online, constantly changing graph, and multi-resolution.

We would also like to allow the user to relate different nodes from different graph visualizations. By doing so, we could display two graphs, for instance, one for authors, and one of citations, and some actions on one graph will affect the other. We believe that investigation using more than one graph could allow more and better insights from the data visualized. In addition, we would like to support some features for layered (labeled) graphs, where there are different types of nodes and edges. We also believe that some global graph algorithms can be very useful for analyzing large graphs, especially clustering algorithms.
Lastly, we are currently using the SceneBin data structure for spatial indexing. We would like to see how more complex data structures, like Quad-trees, handle the same problems, and affect the runtime of the layout algorithms.

10. References


http://citeseerx.ist.psu.edu/


Appendix – Screenshots

The SceneBin data structure of 4x4 bins.
A screenshot of our system displaying a tree of author networks.

Search for essays by keyword dialog.
Our system showing a graph of essays, with constraint that all vertices must be positioned on a circle.