Master Thesis

Visualization of Knowledge Representation & Distributed Activation Spreading in Long-Term Memory

Submitted at the Department of Informatics of the University of Bremen
in partial fulfillment of the requirements for the degree of
Master of Science in Digital Media

Author: Prabhakar Reddy Darakapalli
Matr.Nr: 1696276
Email: prabhakar@gmx.net
Date: July 29, 2005

Supervisors:
Dr. Thomas Barkowsky
Holger Schultheis

Evaluators:
Dr. Thomas Barkowsky
Prof. Dr. Kerstin Schill

Declaration

I herewith assure that this thesis was made exclusively by me, with the exemption of the official support by my supervisors. A complete list of the literature used in this thesis is included in this document

Ich versichere, daß die vorliegende Arbeit – bis auf die offizielle Betreuung durch den Lehrstuhl – ohne Fremde Hilfe von mir durchgeführt wurde. Die verwendete Literatur ist im Literaturverzeichnis vollständig angegeben.

Bremen, 29.07.2005

(Prabhakar Reddy Darakapalli)

Acknowledgments

This master thesis is my first step towards independent research with scientific methods and it would have been impossible with out the people who supported me. I would like to thank my supervisors Dr. Thomas Barkowsky and Holger Schultheis, University of Bremen, for their valuable guidance throughout the thesis. I would like to extend my thanks to Prof. Dr. Kerstin Schill, University of Bremen, for reviewing and evaluating this thesis

Contents

Chapter 1
Introduction

Visual representations of information have always played a vital role in human life. Human perception has the ability to perceive the spatial positions of visual elements and the relationships between them quickly. In accord with the proverb: “A picture is worth more than thousand words”, humans grasp the content of pictures much faster than understanding text. In this thesis, the term *visualization* is used in the sense of a graphical representation of data or concepts.

Graphical representations work as external aids to grasp the information of physical and abstract data of even large quantities. The importance of external aids in enhancing human cognitive abilities can be understood from Norman’s words, “The power of unaided mind is highly overrated. Without external aids, memory, thought, and reasoning all are constrained. But human intelligence is highly adaptive, superb at inventing procedures and objects that overcome its own limits. The real powers come from devising external aids that enhance cognitive abilities. How have we increased memory, thought, and reasoning? By the invention of external aids: It is things that make us smart.” [NOR93].

Developing visualization systems for abstract data is a challenging task as it requires finding suitable visual representations for the information and concepts represented by the data. These systems should give an insight into the abstract data, even for huge amounts, to support users in performing cognitive activities such as grasping the information in the data, identifying the features and patterns existing in the data, validating the data, making decisions etc. Thus, the development of visualization systems should take into account results from research regarding information visualization, human computer interaction, computer graphics, and user interface ergonomics. Only these visualization systems can work as efficient external aids to enhance the user cognitive abilities in working with the abstract data.

The development of visualization to the abstract data of a long-term memory model realizing the concept of retention and retrieval of knowledge in long-term memory is the main aim of my master thesis. This long-term memory model has been developed in R1-[ImageSpace] project of the SFB/TR 8 Spatial Cognition at the University of Bremen, Germany. The model assumes that the knowledge in long-term memory can be represented as interconnected units of knowledge (information) elements. The connections between the knowledge elements are a result of two kinds of relationships between them. It explains the knowledge representation as a network of nodes connected with links. The nodes or information elements have some activation value associated with them. The knowledge retrieval is explained with activation spreading process in the nodes. Every node has some activation which declines over time, and partially spreads to the nodes connected to it. The activation level of a node determines whether the node is retrieved or not. The model is implemented in LISP and its content and processes are made available to other applications in text format.

This thesis aims at developing a visualization tool to visualize this long-term memory model. The tool displays the knowledge representation using graphical visual structure and shows the knowledge retrieval via the spreading activation process in the visual structure with a suitable visual representation.

The purpose of the visualization tool is to enable the user to comprehend the information of the knowledge elements and the network structure of the model easily. Furthermore, it provides the user with necessary functionality to explore and navigate the model. Visualizing the long term memory phenomena of retention and retrieval based on the model enables the user to experiment with different conceptual models of long-term memory. This simplifies the process of checking the suitability of different conceptions which would be difficult if there were no visualization available.

The state of the art related to my topic originates from the general graph or network visualization area as the long-term memory model is depicted as a network visual structure. In the area of general graph or network visualization, there exist some tools like uDraw(Graph) [UDR05] for visualizing arbitrary knowledge bases represented as graphs or networks. Besides these tools, there exist several libraries to be used with programming languages such as Java and C++ to develop graph visualization applications. With respect to long-term memory models, however, currently no tools exist to realize a suitable visualization.

The visualization structure for the long-term memory model requires showing the knowledge representation as nodes connected with two kinds of relationships. In addition to the relationships, it needs animation possibilities to show activation spreading between the nodes. Furthermore, it requires to provide the user with a possibility to interact with the model and to allow navigation to explore the model.

The general purpose graph drawing and visualization tools are mainly useful to visualize static networks. These tools require the application in which they are embedded to do all the graph data handling and just to use them for the purpose of displaying the graph. These tools offer either no or very limited support for domain dependent visualization features such as navigation, animations in the graph, or user interaction with the graph. Consequently, with respect to the visualization of the long-term memory model, these tools can only be used at the front end, i.e., to solely display the model. Therefore, as there exists no specific tool and the general graph visualization tools available are not sufficient for visualizing the long-term memory model, I developed a visualization tool for the long-term memory model as part of my master thesis.

Organization of the Thesis

The remainder of the thesis is organized as follows. In the next chapter, I explain the concept of knowledge retention and retrieval in long-term memory and explain the long-term memory model to be visualized.

In Chapter 3*,* I explain important aspects of information visualization in general.

In Chapter 4*,* I present the issues to be considered to develop information visualization systems for network visual structures. First, I explain the visualization of information which can be represented as a hierarchy and as a network. Then, the chapter will be continued with an explanation of graph layout schemes. I conclude this chapter with an examination of 3D graph visualization.

In Chapter 5*,* I give a detailed explanation of the tool I have developed which is called *LTMVisual*. In the first part of this chapter, I describe how the LTMVisual environment ensures the visualization of knowledge representation and spreading activation processes in the long-term memory model. In the second part, I explain the implementation details

of the LTMVisual. I give design and development details of the LTMVisual architecture along with the technical details of the development platform.

Finally, I conclude in Chapter 6 with a summary of my work and conclusions with respect to the theoretical background I present in previous chapters. Furthermore, I describe the future work that can be done in relation to my work.

Chapter 2

Long-Term Memory: Retention & Retrieval

Humans retain certain information for long time and certain information for very short time. This phenomenon can be observed in day to day human life such as one being able to remember his or her best friend name life long in contrast to not being able to recall a newly introduced name after some time. Psychologists begin their explanation to this phenomenon by making distinction in human memory between *short-term memory* and *long-term memory*. The terms, memory being defined as retention of information over time, describe the fact that some information is retained for short time and some information is retained for long time [REE04]. Later different concepts have been proposed to explain the cause for this difference between these two memories. Such as, the short-term memory is information in active state, i.e., what one person is thinking currently and long-term memory is information in inactive state.

Another (interesting) phenomenon is *recall or retrieval of information.* The retrieval of information is explained with the concept of *working memory.* The working memory is treated sometimes as a separate area to process information and sometimes as another name for short-term memory. The working memory contains the information currently available to the system for processing. This information includes the information about current environment, inferences, current goal information etc. In addition, the working memory retrieves the traces of information from long-term memory to perform cognitive tasks [AND02].

Since this chapter is dedicated to explain the knowledge retention and knowledge retrieval in long-term memory, the remaining part of this chapter only concentrates on aspects of long-term memory (LTM).

2.1 LTM Semantic Network Models

There are different theories of how knowledge or information is retained in long-term memory. Several memory models have been proposed based on these theories [RAA02]. A general class of these models describes the information storage as a semantic network. These semantic networks are composed of nodes and links.

Collins & Quillian's model, by Collins and Quillian in 1969, is one of the first conceptual network model types [COL69]. This model represents knowledge as a hierarchical network containing interconnected units of information. The information units (concepts) are associated with properties and connections between them. In the hierarchical network, at the highest level there are a few general concepts and at the lower level there are many specific concepts. For example, the knowledge model shown in Figure 2.1 contains the general concept animal at the top of the tree and all specific categories are at the lower levels. The knowledge model shown in the figure encloses the knowledge such as a shark is an animal, a fish can swim, and a bird can fly etc.

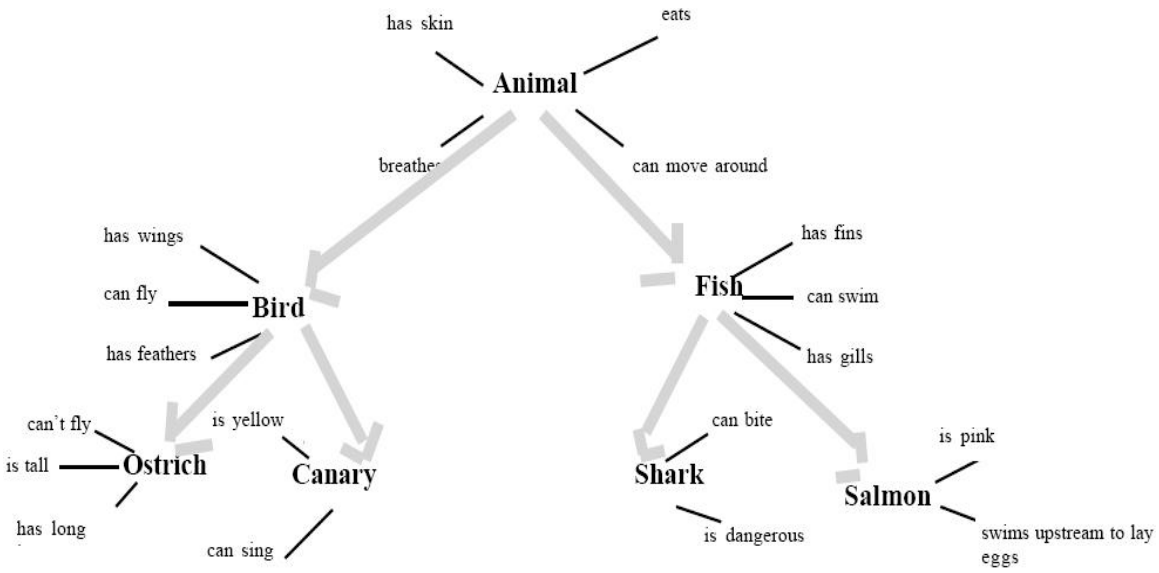


Figure 2.1: Collins & Quillian's Model of LTM knowledge Representation [COL69].

Later, the Collins & Quillian's model was modified and a new model called *Collins & Loftus* is introduced [COL75]. Contrasting to Collins & Quillian's, this model treats both the concepts and properties equally, i.e., it treats both of them as nodes. Links connect the

properties to concepts and to other properties. The links vary in length. The shorter the length between information units, the stronger is the degree of association between them.

The *Adaptive Control of Thought (ACT)* model has been proposed by John Anderson [AND83]. ACT is also a semantic network model, i.e., it has nodes, which represent concepts, and links between the nodes, which represent relations between the concepts. This theory represents the knowledge in a semantic network and has its memory processes defined on the network. It explains the retention phenomenon of the long-term memory in terms of network structures that encode facts and the network structure that surrounds these facts. According to this theory, traces are established in the long term memory from the transient copies of working memory cognitive units. These traces are not lost but the strength of them increases or decays based on the recalls.

The ACT model uses a *spreading activation process* to explain the retrieval phenomenon. The retrieval of information from the network is performed by spreading activation throughout the network. The spreading activation process determines the level of activity in long-term memory. The working memory cognitive units are the sources for the activation in the long-term memory information elements. Activation can spread from these elements to the associated elements in the network. Retrieval of information elements depends on their activation level.

2.2 The R1-[ImageSpace] LTM Model

The term *R1-[ImageSpace] LTM model* is used for this long-term memory model as it is developed as a part of R1-[ImageSpace] project of the SFB/TR 8 Spatial Cognition at the University of Bremen. This model illustrates the mental model of spatial information in terms of interconnected nodes of semantic network, partially, based on ACT and other similar models.

The description of this model is as follows. In long-term memory the persistent (spatial) knowledge is stored as a graph similar to a semantic network. Every node in the graph represents some information which is linked to other nodes (i.e., information) associated with it. The connections between the nodes depict either a hierarchical relationship or factual relationship. Similar to ACT model, knowledge retrieval from long-term memory is realized by spreading activation processes. Every node, N, has some activation which

declines over time, partly spreads to nodes linked to N, and determines whether this node is retrieved. If during reasoning additional information is needed regarding some nodes, these nodes are activated in long-term memory and subsequently spread their activation to other nodes. Retrieved, i.e., transferred to working memory, are those nodes which after spreading are activated beyond a certain threshold.

This semantic network model is associated with the following assumptions:

- Links: Each link has a specified weight, the weight of links increases with usage. The links spread the activation from one node to another node.
- Nodes: each node has a specified activation level. Different nodes contain different activation values.
- Activation Spreading: A node spreads the activation to the nodes which are directly linked to it. On spreading activation, a node's activation level decreases. On receiving activation, a node's activation level increases. The more the links are spreading the activation, the less the activation will spread to any one of the nodes receiving the activation.
- Activation Stimulation: along with the activation received from other nodes, a node can get activation from external stimulation (e.g., because working memory request information regarding a certain node)
- Network connections: the semantic network is connected either with hierarchical and factual links. When the nodes have a hierarchical relationship then they are connected with parent-child links. When the nodes have a factual relationship then they are connected with fact links

With the assumptions stated above, the drawing of a semantic network appear as a network of hierarchical information structures, or in other words, as a graph of interconnected tree structures. Each node in the network represents a knowledge element. The links connecting these knowledge elements belongs to either of the type's parent-child links or fact links. The links conveying a hierarchical relationship between the knowledge elements are the parent-child links. The links, which convey the factual relationships, are the fact links. The knowledge elements or the information elements are associated with a numeric value to represent their activation level. In addition, the edges are also associated with a numeric value to represent the weight. The retrieval of knowledge from the knowledge model is carried out with the process of distributed spreading activation

Chapter 3
Information Visualization

Information can be defined as "all ideas, facts, and imaginative works of the mind which have been communicated, recorded, published and/or distributed formally or informally in any format"[LIS98]. In simple terms, information is a collection of facts or data. Data, on its own has no meaning, but becomes information when it is interpreted. *Information Visualization* can be described as the usage of computer supported, interactive, visual representations of data to facilitate cognition [STU99]. Information visualization enhances the ability of observers while performing cognitive activities such as solving mathematical problems with abstract, complex, and multidimensional data. In addition to, it augments the interaction with the information using visual representations and extends a person's working memory and long term memory in the process of thinking.

The information visualization has numerous of advantages. It provides an ability to comprehend huge amounts of data. With the visualization, the important information even from more than a million measurements can be made immediately comprehensible. Visualization makes the problems with data, if any, to become immediately apparent and thus enabling the user with the data validation. It allows the perception of emergent properties that were not anticipated before the visualization. Further more, it facilitates the understandability of both large-scale and small-scale features and patterns existing in the data.

Information Visualization as a distinctive field of research area has less then fifteen years history and has rapidly become a far-reaching, inter-disciplinary research field [CHA99]. However, the research related this field originates from several fields and dates back to several hundred years. [PLA86] is considered to be among the earliest to use abstract visual properties such as line and area to represent data visually. [BER67] published a theory in the semiology of graphics which identifies the basic elements of visualization and describes a framework for their design. [TUF83] published a theory of data graphics emphasizing on how to maximize the density of useful information. [CLE88] published a

book called Dynamic Graphics for statistics with a particular interest on how to visualize data with several variables. The new generation of user interfaces with powerful graphics capabilities to allow complex interaction with large amounts of information facilitated the development of information visualization as a distinctive emerging field.

The different stages in the process of developing an information visualization system for a given data are the Collection of data, preprocessing of the data, the display hardware and graphics rendering algorithms, and the human perception and cognitive system. These stages involve a number of feedback loops, for example, the data collection may need to be repeated to gather more data after considering the perceiver special interests while developing the visualization system.

The four important issues to be considered while developing information visualization systems are human perception, visual elements, mapping data to visual structures, and factors influencing the visualization. The remaining part of this chapter is divided into four sections each section explaining one of these issues.

3.1 Human Perception

Consideration of the properties of human perception is an important aspect in information visualization systems. The systems have to use the visual structures of data so as to bring those properties bear. A fundamental cognitive principle for information visualization in human perception is whether the processing of information is done deliberately or preconsciously. Some visual information is processed automatically by the human perceptual system without the conscious focus of attention [TRI86]. This type of information processing is called *automatic, preattentive, or selective processing*. Otherwise, it is called *controlled processing* of information. Controlled processing is detailed, serial, low capacity, slow, and conscious. On the other hand, automatic processing is superficial, parallel, high capacity, fast, and unconscious. An example of automatic processing is the *visual popup* effect that occurs when a single red object is noticeable from a group of white objects. Many features can be preattentively processed, including length, orientation, contrast, curvature, shape, and hue [TRI88]. However, the automatic processing occurs for a single feature in most cases.

3.2 Visual Elements

The visual elements, which encode data in information visualization systems, can be categorized into four types based on their dimensions. They are the elements with:

- Zero- or none- dimensional, Points
- One dimensional, Lines
- Two dimensional, Areas
- Three dimensional, Volumes

Point marks and line marks can be used to show graph and tree topology. They show objects and the relations among objects. The position of objects in a graph and Tree topology can create the gestalt properties such as proximity and closure [COL00]. As these are easily comprehensible perceptional features, it is convenient to encode the information like the clustering of samples and partial trends with position [STU99].

However, the visual elements and their positioning alone can not encode much information. Graphical properties, such as color, do encode information and help in the observer's cognitive process. There are several graphical properties to encode information. [MAC95] proposes the crispness (the inverse of the amount of distance used to blend two areas or a line into an area), resolution, transparency, arrangement (e.g., different ways of arranging dots). [MAC95] also proposes usage of color and dividing the color into color value, hue, and saturation.

Table 3.2.1: Visual features that enable automatic processing [HEA95].

Number	Color
Line Orientation	Closure
Length	Direction of motion
Width	Flicker
Size	Binocular luster
Curvature	Stereoscopic depth
Terminators	3D depth cues
Intersection	Lighting direction
Intensity	

In addition to the graphical properties, time can also be used along with visual elements. Human perception is very sensitive to the changes in the positions, i.e., animation can encode any type of data [STU99]. Animation gives the observer an adequate ability to keep track of changes in visualization. It is rather inefficient to rely on object states than conveying the change and the identity of visual elements across the visualization with the aid of animation.

Indeed, all the graphical properties found in the perception literature are potential candidates to use for visualization. Table 1.2 shows several properties that support automatic processing with visual perception.

3.3 Mapping Data to Visual Elements

The transformation of data to visual elements plays a vital role in information visualization. Mapping data to visual form to encode information has to be carried out in a way that preserves the data. However, ignoring the unimportant data is also crucial in information visualization. A mapping is said to be effective when it is faster to interpret, can convey more distinctions, or leads to fewer errors [STU99]. The direct mapping of data to visual form might be inefficient and difficult as the data can be in several formats basing on where it is produced. So, it is a good idea to transform the data into a meaningful format before mapping to visual form. One such a meaningful format is arranging the data into a set of structured relations. This can be done by finding different variables existing in a given data, different states of the data and values of the variables in each of the states. When this information is arranged in the form of a table, the variable names and state description becomes the column or row headers and the value of each variable for each state becomes an entry in the corresponding cell of the table. Having the data in the form of table makes it easier to develop a scheme for visualization and to map to a visual form.

The usage of Meta-data enhances the mapping process. Meta-data is the descriptive information about data. Meta-data aids in simplifying the process of developing a visualization scheme. An important kind of Meta-data is the structure of data, for example, the structure of a data table when the data is arranged in the form of a table, the

information about the rows and columns are the potential types of Meta-data. The detailed process of mapping raw data to data tables has been explained in [STU99].

3.4 Factors Influencing the Visualization

The limitations in graphics hardware, the amount of data, and the complexity of the visualization are three sources of problems to be answered to increase the effectiveness of visualization systems.

Graphics Capabilities

The representational limit of graphics is an important factor influencing the quality of visualization. The most important aspect of visualization is the usage of space. Space is very limited and moreover it is very useful encode the structural information, it is important to decide which data variables have to be shown with an impact on the data variables that are not shown. Since it is crucial to use space efficiently, some view transformations are helpful in positioning the data variables on the limited size of display screens. There are three common view transformations: Location Probes, Viewpoint Controls and Distortion [STU99].

Location probes use the location of visual elements in the display space to show additional information about the data being visualized. A detail on demand pop-up window giving details about the context is a good example for location probes. Viewpoint controls are the transformations that use affine transformations to zoom, pan and clip the view port. These transformations allow the observer to view more details with zooming and to navigate the data through view port changes. Making zoom rapid and easy to invoke helps the observer to not keep much cognitive effort to remember the invisible information at a given instance. Distortion is another transformation that modifies a visual structure to create focus plus context views. For example, a hyperbolic transformation of plane graph with nodes and edges maps to a circle or elliptical, so the nodes that are far from the root shrink.

Amount of Data

The size of data set is another important factor that has lot of influence on the effectiveness of visualization system. Developing visualization schemes with out considering the data volume may cause worst results in the visualization when the data set size becomes large. The improvement in the Visual efficiency is quite significant when the problems related to large data visualization are minimized. The first problem to mention is the Occlusion which happens when two or more visual elements overlap each other, causing bad perception of information. Large data increases the occlusion as it increases the number of visual elements shown and in turn increasing the probability of overlapping in them. Aggregation is another problem in large data visualization. When the visual elements are drawn over each other it is hard to determine the number of them and results in misinterpretation of data.

In order to overcome these problems [MAT97] proposed two solutions: Visual abstraction and clustering. Considering the limited human cognitive capabilities, it is often unnecessary to show all the information at once. Visual abstraction discards some information in a given instance of visualization. The purpose is to reduce the volume of information with a preservation of its meaning. It is essential to know which details contain more information to carry out these visual abstractions.

Typically, the data that describe the real world contains groups of samples sharing similar properties. Most of the samples inside such group do not bring any new information and thus they can be filtered out without any significant damage to the meaning of the data. In such a group, all those similar samples can be discarded and replaced by a single larger and simpler element. Such a representation is a good starting point for visual abstraction and a promising solution for large data information visualization. Clustering of data helps to obtain an estimation of presence of the groups containing similar elements. It starts with dividing the data samples into clusters, i.e., assigning similar samples to same cluster [MAT97].

The precision of estimation of groups depends on clustering algorithm and on the data itself. There are two types of clustering algorithms depending on the direction of their progress [SER04]. Top-to-bottom algorithms group the samples by partitioning the whole set into decreasingly smaller areas until a certain level of precision is reached. They are less time consuming, but might produce wrong results by accidentally drawing the

Partitioning border between similar units. In contrast, the bottom-to-top algorithms start on the lowest level of abstraction, grouping pairs of similar units and clusters together. For the purposes of visually effective large data visualization the bottom-to-top approach is better choice [SER04].

Complexity

Visualization complexity has lot of influence on the Comprehension of information. Thus the complexity has to be minimized for an effective visualization. The number of data dimensions displayed simultaneously is an indication for the cognitive complexity. Having multivariate variables which require several colors and patterns to denote results in an increase in the number of dimensions to a data. The higher the complexity of data, the higher the number of visual elements required for visualization and hence the higher the cognitive effort on the part of the observer. The classification of combinations of visual attributes which can be processed by an observer in almost instantly vs. those which require a slower visual search can result in quantified guidelines [RIC97]. Such guidelines help in designing visualization schemes to convey multidimensional information effectively.

Chapter 4

Information Visualization in Networks

A *graph* is defined as a set of nodes and a set of edges connecting the nodes. In mathematical terms, a graph is referred to as a collection of points and a collection of lines connecting a subset of the points. The terms *point* and *vertex* are often used as synonyms for node. The terms *line, link,* and *arc* are often used as synonyms for edge. A simple graph is shown in the fig 1. It is to be noted that the change in the length of edges, the orientation of graph and shape of nodes do not change the meaning of a graph. Figure 4.2 depicts the same graph shown as in Figure 4.1, but with a different size and different orientation, i.e., the nodes C, D, and E have been moved to different place. Both graphs, irrespective of their appearance, represent the same information.

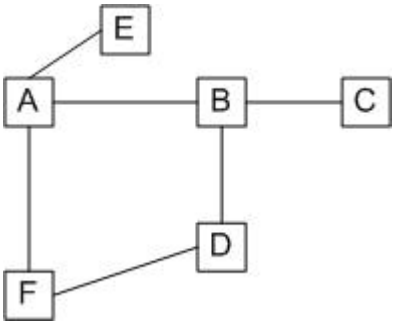


Figure 4.1: A Sample Graph 1.

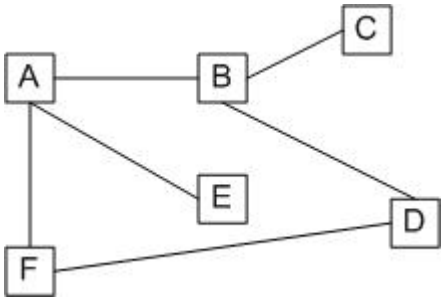


Figure 4.2: A Sample Graph 2.

The graphs can be either directed or undirected. When a graph contains one or more directed edges, typically drawn by placing arrowheads at the ends, it is called *directed graph,* otherwise it is called *undirected graph.* The graphs shown in Figure 4.1 and Figure 4.2 are undirected graphs and the graph shown in Figure 4.3 and Figure 4.4 are directed graphs.

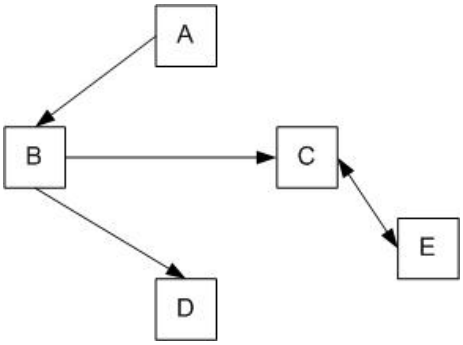


Figure 4.3: A Sample Directed Graph.

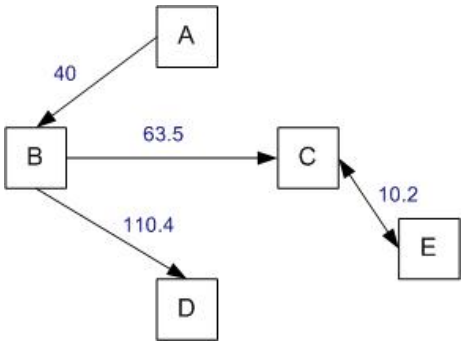


Figure 4.4: A Sample Network.

The term *network* is used to denote a graph with weighted edges. The graph shown in Figure 4.4 has edges with weights, so it can be called as a network. However, often both the terms graph and network are used as synonyms. A *tree* is a connected acyclic graph, i.e., a special kind of graph in which all the nodes are connected and no cyclic links exist.

Networks play a special role in information visualization as many real world domains can be represented as node-link based networks. For example, the domain of WWW, web pages as nodes and references as links, can be represented as a network. A dictionary, internet addressing, library catalog, telephone systems, computer programs etc are other potential examples for such domains.

Information visualization systems for such networks are designed to satisfy the domain specific needs or goals. In addition to domain specific goals, those systems try to achieve some general network visualization goals. The general visualization goals, particularly to mention, are utilizing the display space efficiently, facilitating the user to build a mental model of the structural information in a visually pleasing manner with a low cognitive effort, and conveying content information related to maximum number of individual nodes at once.

This chapter, information visualization in networks, as the name suggests concerns the issues related to developing visualization systems for network structures consisting of hierarchical information structures.

4.1 Network Information Elements

In a network visual structure, various properties associated with various visual elements are used to convey the information to the user. Nodes, links, node labels and edge labels are primary visual elements. Size, color, time and sound are the primary parameters to be used with the visual elements of a network to convey information.

Size and color are the properties of a node or link which can be synchronized with the underlying data. For example, different colors for links can be used to denote different kind of links. Different sizes for nodes can be used to convey different information such as number of links or the position of a numeric data value of node in a given range of values.

When the data at each node or link is varying through time, it is crucial to select a proper time point to show the data. The variations can be shown as distinct state changes or smooth animation. In a visualization of time varying data, it is important that the user is provided with an ability to control the speed of visualization to be able to understand the changes in data.

Sound is an independent channel to convey information. Sound can be associated with a node which is played when a node is selected or activated etc. during animations, a sound can be used to convey the state changes of a network structure.

4.2 Visualization of Hierarchical Data

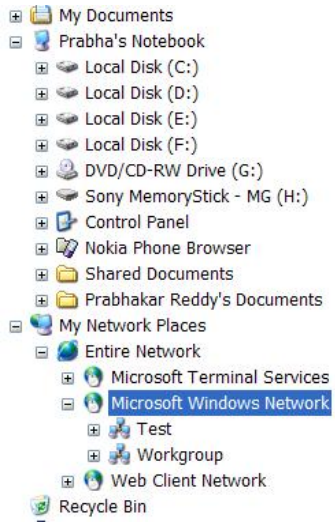
Hierarchical structures of information, in a knowledge base, contain hierarchical relationships between the data items. The data items can have parents, siblings, and children. The data items can be termed nodes and the relations between them links. These kinds of hierarchical information structures contain two types of information: structural information associated with the hierarchy, and content information associated with the nodes and links. It is easy to comprehend a hierarchical structure as long as it is small, but it gets difficult as the structure becomes large. There exist different visual structures to encode and to facilitate the navigation of hierarchically structured information.

Outline, node-link diagram, and *tree-map*, shortly described in the following paragraphs, are the three popular visual structures used to visualize hierarchical structures.

The outline method is very popular in file browsers. It explicitly provides the content information with the labels of nodes and structure information with indentation of nodes. Each level starts with an indentation. A snapshot of windows file browser based on outline method is shown in the Figure 4.5. As shown in the figure, each new node adds a new line to the hierarchy. So the number of lines required to display is linearly proportional to the number of nodes in the hierarchy.

Outline is a good visualization structure as long as the hierarchy has fewer levels and few nodes. It becomes inadequate to convey the structural information for hierarchies having number of nodes more than a few hundred [BRI91]. Even with a few hundred nodes it requires scrolling, opening and closing of the levels. The navigation of the structure gets hard as the number of nodes increases. So, for a user, it becomes hard to form a mental model of the whole hierarchy structure [CHI91].

Hierarchical Node link diagrams, which are called trees, are an important category of effective visual structures to encode hierarchical information. Tree visual structure is very good at conveying the structural information. The data cases and sub data cases are shown as nodes and are connected with links.



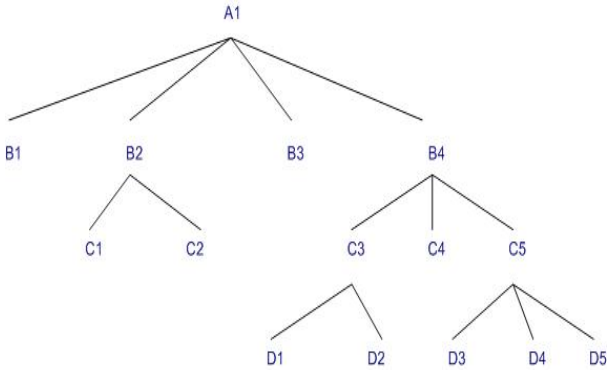


Figure 4.5: A Sample Outline.

Figure 4.6: A Sample Tree.

A sample tree structure is shown in Figure 4.6. The way space is used to position the nodes is very important in using a tree. The tree shown in the figure is the typical tree, where the Y-axis is used to show the tree depth and X-axis is used to show the separation of nodes. But trees can also be drawn by using X-axis of a square or radius of a circle to show the depth and Y-axis or angle to separate nodes respectively. Figure 4.11 shows a tree drawn using radius to show depth and angle to separate nodes. There have been many sophisticated techniques are presented to improve the visual efficiency and aesthetic aspects of trees [JAR99].

As it can be observed in the Figure 4.6, Tree visual structures makes inefficient use of space, i.e., most of the pixels in a area covered by tree contribute to the background. In a typical tree drawing more than 50% of the pixels are used as background. This inefficient usage of space is not a problem as long as the tree size is small, but for large trees it results in difficult navigation and hidden information because of the limited size of displays. Because of their structure, only text labels of limited length can be used with nodes. Additional information with each node, such as increasing the length of label, causes dramatic increase in the usage of display space.

As the information is hidden within the individual nodes of the hierarchy, for the structures generated by both the outline and node-link methods, it is difficult for users to navigate and to extract information from large hierarchical structures. In contrast to outline method, trees are very useful to show any information associated with links such as weight.

Tree-map is a solution proposed by [BRI91] with an objective of utilizing the display space efficiently. Relatively new, the tree-maps method uses enclosure to visualize hierarchical structures. Figure 4.7 shows the same hierarchical structure as in the Figure 4.6, but with a tree-map.

Tree-maps are constructed via recursive subdivision of an initial rectangle into sub-rectangles. Each individual rectangle represents a leaf node and gets a size reflected from the size given to that node. For instance, the size can be a weight assigned to a node or a file size when the node represents a file. The subdivision alters for each level, first horizontally and then next vertically and so on until all the leaf nodes are represented by a rectangle. Visual properties such as color, texture, border, blinking etc. can be used with each rectangle to convey additional information.

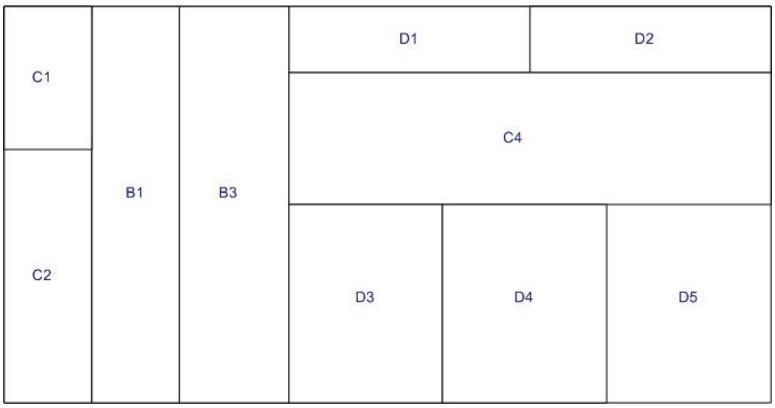


Figure 4.7: A Sample Tree-Map.

The Figure 4.8 describes a tree-map with more than 1000 nodes. When the same structure represented with a tree with a visually distinguishable node placement, it would have occupied several times of display space compared to this tree-map.

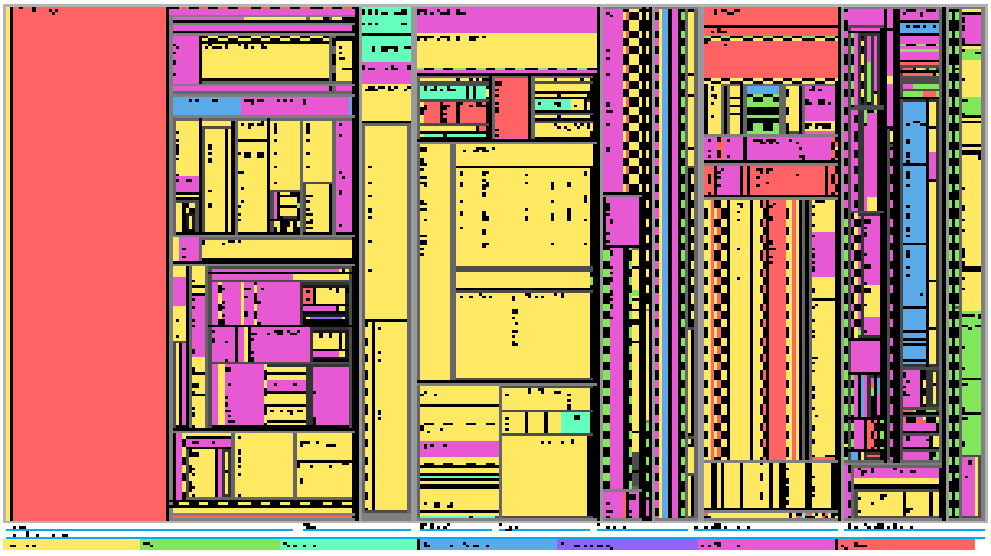


Figure 4.8: A Complex Tree-Map with more than 1000 Nodes [BRI91].

Despite its advantage regarding utilizing display spaces efficiently, tree-map has some drawbacks. Tree-maps demand extra cognitive effort on the part user to grasp the structure of the hierarchy being visualized. Tree-map worst case scenario is displaying a hierarchy where each parent has the same number of children and each leaf node is of the same size. In such a scenario the tree-map would appear as a grid of equal sized

rectangles and result in a maximum loss of structural information. Nested tree-map, where each group of siblings is surrounded by another rectangle, is a solution to describe hierarchical information in such a scenario. But it results in an inefficient usage of display space. Coloring of each group of siblings different from others can convey the structural information, but then the color can not be used to denote the properties of individual nodes.

The bottom line is: To display large hierarchical structures which have more than a few hundred nodes, what ever may be the visual structure it is hard to satisfy all the visualization goals in a single visual structure.

4.3 Visualization of Network Data

Networks consist of a set of nodes, links and additional data associated with nodes and links. In contrast to hierarchies, networks may contain complex cycles and internal hierarchies. The important issues to consider in network visualization are positioning of nodes, handling of edges, handling of the scale of graph when the number of nodes and links increasing, interacting with the network and navigating through large networks of information.

Positioning of nodes is a key issue in network visual structures. It greatly affects the comprehension of the knowledge base structure behind the network. There are several approaches to position the nodes. One approach to position nodes is to use the spatial properties associated with nodes, if nodes have any. For example, consider a network of all the universities located in Europe where each university is represented by a node. When the whole Europe map is considered as the background then the locations of universities could become the positions of nodes and lead to the meaningful network. However, sometimes using spatial properties of the nodes has adverse effects such as clustering of edges to the extent where the user is not able to distinguish between different edges.

Another approach is to define criteria for deciding what constitutes a desirable node positioning such as minimal edge crossing, edges length always less than a value etc. after defining criteria, graphical layout algorithms can be used to produce a well organized layouts from the network properties satisfying the defined criteria.

The second issue is how to handle the links as the network size is increasing to convey the actual information that is meant to be described by the links. Links can be colored to convey the type of link. Group of links connecting same pair of nodes can be replaced by one link. Furthermore, all the links can be dropped and node properties such as height and width can be used to convey the links information. In such a case, the height and width of nodes can represent the number of incoming and outgoing links, respectively. Besides the edge representation, routing of edges plays an important role in handling the edges. In some cases a straight line from the source to target for each link would be enough. But, sometimes, the edges have to be routed from source to edge such that they do not intersect any node.

The positioning of nodes and handling of edges are interdependent on each other and both carry some information in a network. If positioning corresponds too perfectly with the linkage relationship and if links carry only the structural information then the links add little information. On the other hand, if there is no relationship in the positioning, the linkages appear very random and obscure. In a good visualization node positioning and links add information to each other [KIM88]. Detailed information on node positioning can be found in section 4.4

The problem of handling the size of a network when the number of nodes is very large is to be addressed in network visualization. Aggregation is one of the approaches to handle large size networks. The idea is to represent multiple nodes neighbored to each other by a single node when the user's point of view is distant. Aggregation can be applied to a network with lot of hierarchical structures where parts of the hierarchy are replaced by a single node. In [STE93], aggregation is applied to an example concerned with changes to a large section of a computer program. The example software comes with a natural hierarchy of subsystems, modules and files. Within a module all the file-file links are retained. The links between a file and files in another module are replaced by a file-module link. The links between files in one module and files in another module are replaced by a module-module link. The whole aggregations result in a reduction of number of links by a factor of 32. Furthermore, retaining only 1% of the strongest links reduces the number of links by a factor of 100. So finally, 8 million links in a large software project are reduced to 2500 links. Hence, aggregation is a suitable technique to handle the scale of large networks and aggregation can be applied at the hierarchies within the network or a group of neighborhood links and nodes.

Interaction and navigation in a network is an important issue which has to be kept in mind while developing a visualization scheme for network structures. Not only are networks of interest important in terms of structure, but also in terms of content information of each node or link. For users, having the ability to examine the content of each node and relate them to the network is of crucial importance. Navigation is necessary to cope with the size and complexity of networks. Users can navigate in a network structure either by being able to move the view port to the desired portion of the network or by being able to move the network structure itself by moving the desired portions to the view port. Besides the provision of navigation to the desired portions of network, that is, controlling the locations, it is important to provide the user a provision to keep the track of locations in a way to recognize the locations. Locations in a network can be controlled with the help of functions such as scrolling, forward, backward, zoom in, zoom out of the view port.

Recognizing locations in a complex network can be implemented with the provision of navigation aids. A small map can be used in the view port with a path that was followed to reach current position to enable path retracing. The path retracing with navigation aids helps to re-establish the context that led to current position. Showing the nodes near the view port larger compared to the nodes far from view port can facilitate the recognition of current static position.

4.4 Graph Layout Schemes

A *graph layout* is assigning coordinates to the nodes and edges in a graph. The layout is critical in graph visualization. If nodes are placed randomly the information will become a confusing jumble. There has been substantial research on the problems of graph layouts in computer science and several layout schemes have been published. Some schemes concentrate only on position of nodes while others concentrate on bending of edges. There are several standard constraints on the positioning of the nodes including crossing avoidance, congestion avoidance, bend minimization and edge length minimization etc. the different schemes have been optimized for different criteria.

Many layout schemes try to minimize the problem of edge crossings. So as to avoid a visual impression with the crossings that does not reflect the actual structure. Edges

passing underneath nodes may divert the user's attention from the important graph structure.

Besides the problems in static layouts, incremental layouts pose a new problem to the layout schemes. When the layout of a graph is to be repeatedly changed based on the changes in the knowledge base underlying the graph then such kind of layout is termed incremental layout. The incremental layouts invoke the redrawing of graphs repeatedly based on some criteria. Layout schemes that redraw graphs in the same way on each invocation can help the user to form a mental model of the graph structure, otherwise laying out differently on each invocation confuses the user.

One popular strategy that can be used with most layout schemes, for an effective visualization, is the *fisheye view* which tries to attempt to give a useful balance of local details and surrounding context. The idea of fisheye views is to highlight the structure and node details of a graph near the view port and at the same time showing overall structure of the graph. Objects close to the centre of view port are magnified greatly, and this magnification drops rapidly for the objects which are farther from the centre of view port. Such a view tries to achieve smooth integration between both local detail and global context of a graph by repositioning and resizing nodes and edges. However, even for graphs with a few hundred nodes, the advantage of this approach are lost as the areas away from the focus become too congested to be understandable.

To summarize, the graph layout schemes compute the layouts basing on some criteria and the following are some important criteria:

- To use space efficiently
- To avoid distractions which are caused by edge-edge and edge-node crossings, node-node overlaps
- To maintain a symmetry in the structure of complete graph.
- To facilitate the rapid extraction of both structural and content information of graphs with low perceptual and cognitive loads
- To facilitate easy and efficient interactive control over the presentation of information in graphs
- To draw a esthetically pleasing graph

The remaining part of this section gives a brief introduction for some popular graph layout schemes.

Force-Directed Layout Scheme

The algorithms based on force-directed schemes simulate the behavior of physical systems where the nodes behave as mass points repelling each other and edges behave as springs with attracting forces. The algorithms aim at finding an equilibrium state of system for which the total forces on each node is equal to zero. The Force-directed scheme is suitable for drawing undirected graphs with cycles [THO91]. Figure 4.9 describes a complex graph drawn with algorithm based on a force-directed scheme.

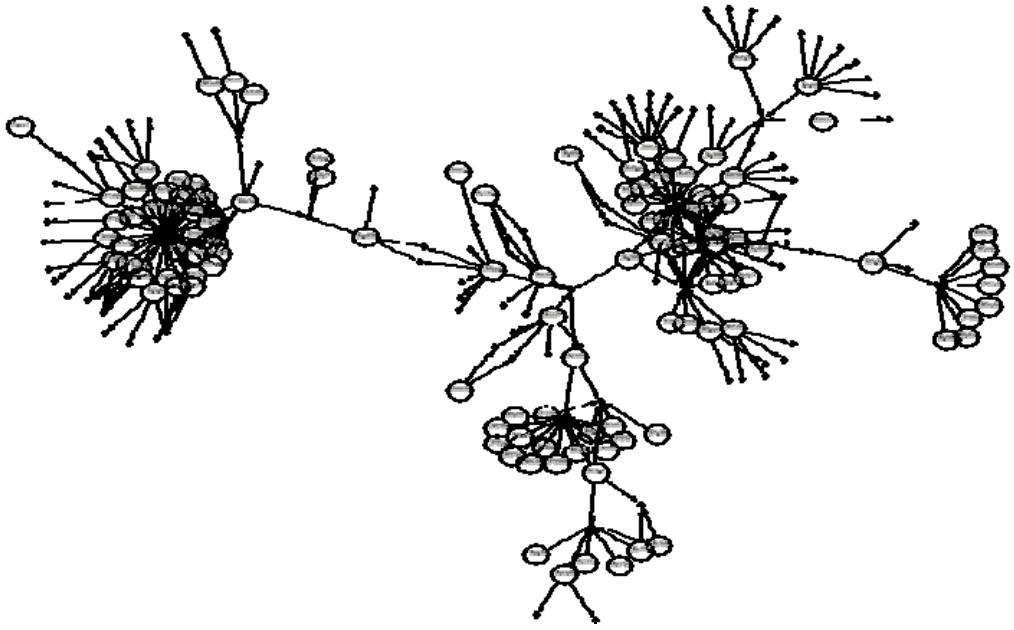


Figure 4.9: A sample Force-directed layout.

Hierarchical Layout Scheme

The algorithms based on the hierarchical layout scheme aim at highlighting the direction or flow in a directed graph. The nodes are placed from top to bottom in hierarchically arranged layers. Within each layer, the nodes are rearranged in such a way that the number of line crossings is minimized. Figure 4.10 shows a sample graph generated with

an algorithm based on Hierarchical Layout scheme. For few nodes, hierarchical schemes considered to be giving pleasant results.

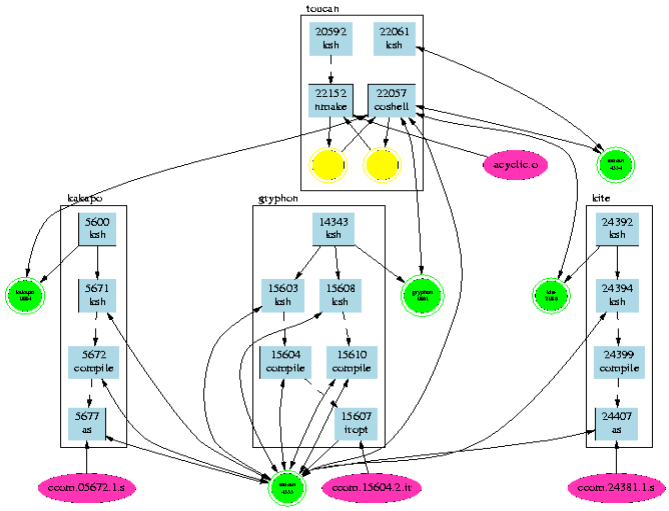


Figure 4.10: A sample Hierarchical layout.

Radial Layout Scheme

This scheme is suitable for a network when the whole network can be depicted as a single tree. The algorithms based on this layout scheme places the root node of the tree in the centre of the layout and lays out the other nodes in concentric rings around the focus node as shown in Figure 4.11.

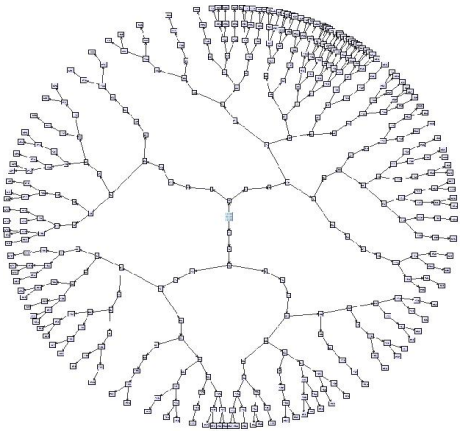


Figure 4.11: A sample Radial Layout.

Each node lies on the ring corresponding to its shortest network distance from the root node. Immediate neighbors of the root node lie on the smallest inner ring, their neighbors lie on the second smallest ring until the most distance nodes form the outermost rings. The angular position of a node on its ring is determined by the sector of the ring allocated to it.

Circle Layout Scheme

The algorithms based on circle layout scheme emphasize the group and tree structures with in a network. Then they analyze the network to create node partitions from the connectivity of the network. Each node partition is arranged as a circle. All the circles are arranged in a radial tree layout fashion. A sample network created with an algorithm based on circle layout is shown in Figure 4.12.

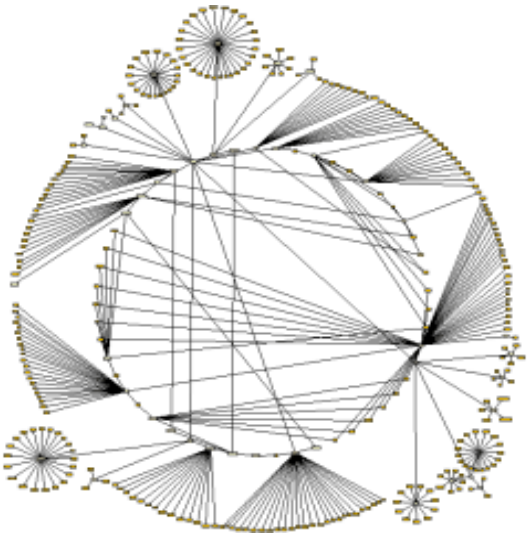


Figure 4.12: A sample Circle layout.

Orthogonal Layout Scheme

This scheme is suitable for the algorithms used to generate medium-sized graphs. It generates compact graphs with no overlaps, few crossing and few bends. The algorithms based on this scheme generate the graphs in three phases. The first phase starts with a calculation of edge crossings. Second phase calculates the edge bends. Third phase uses

the results from both the first phase and second phase to give the final coordinates to the graph elements. A sample graph generated with an algorithm based on this orthogonal layout scheme is shown in Figure 4.13.

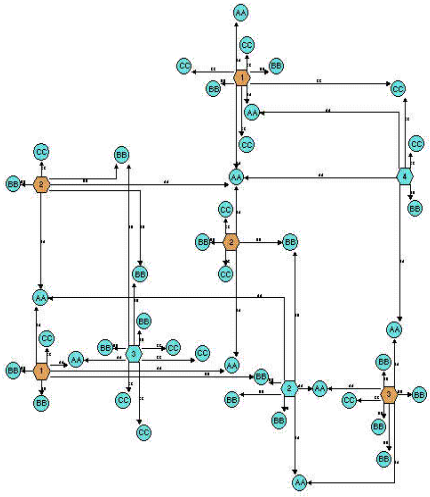


Figure 4.13: A sample Orthogonal Layout.

4.5 3D Graph Visualization

Adding additional dimension to two dimensional (2D) visual structures results in three dimensional (3D) visual structures. This additional dimension in 3D projecting from view port towards infinity creates a large visible workspace and encodes more information in the same view port compared to 2D. 3D visualization provides the focus plus context views, using the third dimension, by making nodes seen in the foreground bigger in the volume compared to the volume of nodes seen in the background of a view port. By changing the foreground or by rotating the graph, the focus can be changed. 3D is clearly more effective for visualizing physical data that includes 3D spatial variables difficult to map to 2D. But, when it comes to abstract data, 2D has a long and effective history.

Spatial navigation, layout and information representation are the three issues that can be considered to differentiate between 2D and 3D visualization of graphs. The different geometric operations available in 2D presentation of graphs are translation in x and y dimension, scaling about a point and rotation about a point orthogonal to the screen. In addition to the operations available in 2D, 3D presentation allows translation and rotation

in three different directions. Laying out graphs in 3D is easy to satisfy the minimal edge crossings criteria compared to the laying out in 2D. Edge crossings are less important in 3D layout due to human ability to perceive depth with which they do not perceive the crossings as intersections. The symbols used for information representation differs in 2D and 3D presentations.

In contrast to 2D, 3D involves several design difficulties. It poses a challenge to construct objects which can be equally understandable from any view point. 3D implementation involves consideration of many parameters including lighting, texturing, occlusion, shadowing, and three dimensional movements. So, it requires significantly more processing power and more work on the part of designers and developers.

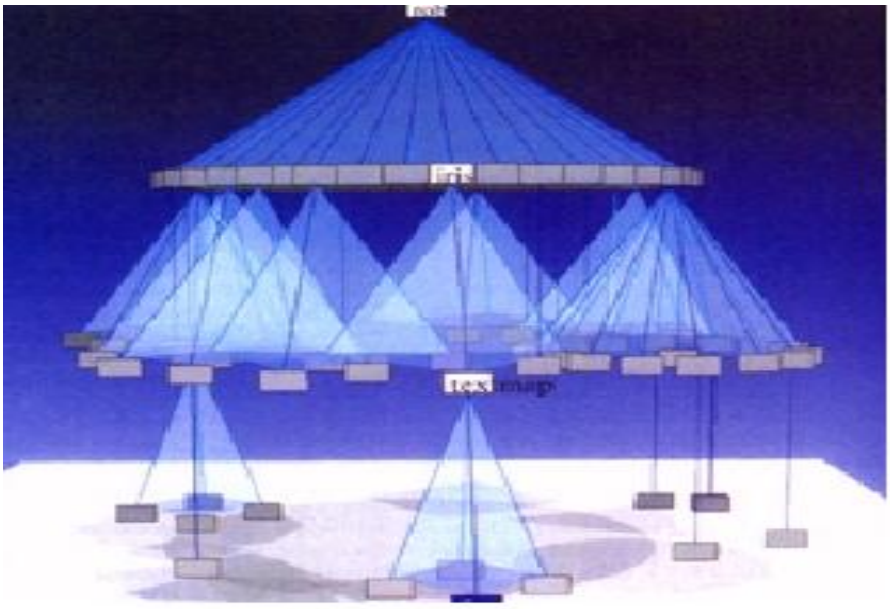


Figure 4.14: A sample 3D cone visualization

The difficulties in design and requirement of high processing power made 3D less popular in abstract data visualizations such as graphs compared to 2D. In graphs, which are nothing but node-link diagrams, as the number of nodes and links increases the handling of graph becomes hard. However, with the advent of high processing computers in mass-market and with the advancements in 3D-design software, 3D visualization is likely to become more common [STU99].

3d cone tree [ROB91] is a popular three-dimensional representation for hierarchical structures to provide interactive navigation through the hierarchy. The nodes are ordered on a 3D cone as shown in Figure 4.14. Selection of a node invokes rotation in the cone and brings the active node to the foreground. Cam trees are a variation of cone trees which are oriented horizontally as opposed to vertically.

Chapter 5

LTMVisual: Visualization & Development

LTMVisual stands for long-term memory visualization and is the name of a tool I have developed as part of my master thesis. Briefly, LTMVisual is a software application which provides an environment to visualize knowledge representation structures and activation spreading process in long term memory based on a model, referred to as *long-term memory model,* developed in the R1-[ImageSpace] project (for details, refer to chapter 2). LTMVisual, a two-dimensional graphical user interface, helps the user to view, analyze, and interact with the knowledge model.

To summarize the long-term memory model, it is a network of interconnected trees containing nodes. The nodes are connected either by parent-child links or fact links. The nodes are associated with a numeric value of activation and the edges are associated with a numeric value of weight. The nodes spread activation to other nodes in the network via the links.

The scope of LTMVisual is to grasp the content and processes in long-term memory from a text based long-term memory model available from another tool in the R1-[ImageSpace] project. LTMVisual parses the text based information representation of the long-term memory model, builds an internal representation of the model, and renders the model onto the screen. Along with the visualization of static model, LTMVisual also enables the user to visualize the process of spreading activation in the network model. Besides visualization, the user is provided with a meaningful interaction with the long-term memory model.

This chapter is divided into two parts. In the first part, I explain the visualization in LTMVisual environment. In the second part, I explain the implementation details, design, and development, of LTMVisual.

5.1 Visualization in LTMVisual

5.1.1 LTMVisual Environment

The LTMVisual plain window environment is shown in Figure 5.1. Considering the ergonomic aspects of graphical user interfaces, as shown in the figure, all the important and frequently accessed functions for the visualization are assigned to controls on two toolbars. One toolbar is oriented horizontally while the other one vertically. The horizontal toolbar is comprised of *process controls, progress bar*, and *filters*. The process controls, from left to right on the toolbar, are *start*, *suspend*, *resume*, and *stop* buttons. As the name implies these buttons help in controlling the whole visualization process right from the beginning to the end. The start and stop buttons correspond to the start and end events of visualization process and are used only once for visualization of a long-term memory model. However, the controls suspend and resume can be used several times during the visualization to suspend and to resume at any point of time. Overall, the process controls, give the user a moderate control over the process of visualizing the long-term memory model.

Contrary to the buttons, the progress bar is not an interactive control, but rather a control to show the progress of visualization. The progress bar comes into play only when the long-term memory model is read from a text file. In such a case, the progress shown in the progress bar maps to the number of data nodes in the text file.

The filter controls, named *f1*, *f2*, *f3*, and *f4* have a vital role in LTMVisual. These controls help to inspect particular knowledge in the network by showing only a part of the long-term memory model data. The four filter buttons correspond to *hide fact links (f1), hide parent-child links (f2), hide non-selected nodes (f3), and reset all filters (f4)* functionality.

The vertical toolbar is comprised of *zoom controls* and *speed control*. The zoom controls enable the zooming of the long-term memory model to be able to view details of the parts of network or the entire long-term memory model structure. The speed control enables the control of speed of visualization mainly during the activation spreading process.

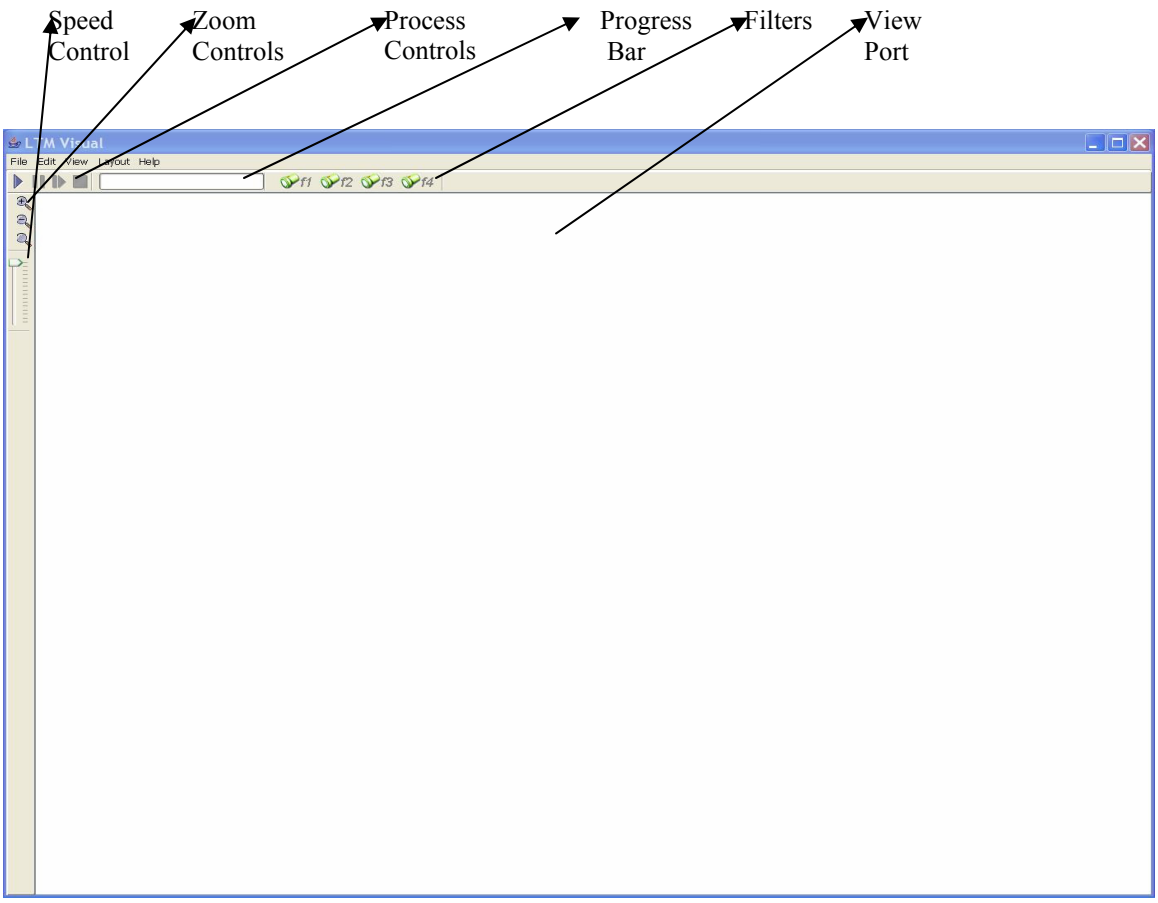


Figure 5.1: The LTMVisual Environment.

As it can be seen from Figure 5.1, nearly all of the space in LTMVisual main window is assigned to the view port. The view port is displayed with no scroll bars as long as the network model fits into the display size of screen. However, the scroll bars automatically appear as soon as the size of the network model increases the size of the view port.

Along with the controls on the tool bars, the layout menu has a crucial role in setting up the LTMVisual environment. The Layout menu as shown in Figure 5.2 contains the settings related to the way the network model is displayed. The layout menu is enabled until the start of visualization of network model and is disabled once the visualization starts. The layout menu contains all the settings related to the view port layout. Several possible settings are the layout schemes, the type of nodes, whether to include the numeric value of activation in the node labels, and whether or not to show the incremental layout (see chapter 4) during the build process of network model.

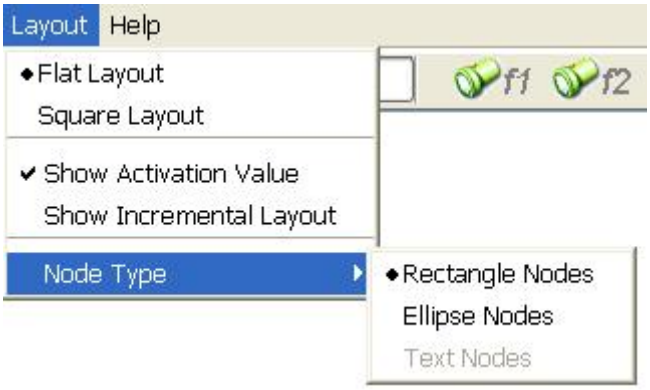


Figure 5.2: The LTMVisual Layout Menu.

5.1.2 Long-Term Memory Model Representation

LTMVisual arranges the long-term memory model nodes as a network of trees. With in each tree, the nodes are separated into different levels based on their parent in the tree. Furthermore, each level is divided into different layers where the number of layers in a level equals the division of number of nodes in that layer with maximum number of nodes allowed per layer. The trees are placed in horizontal direction. The parent-child links connect the nodes within a tree and the fact links connect the nodes within a tree and nodes between trees.

The primitive visualization parameters space, color, size, and text are used to encode the long-term memory model information in the network model. The space encodes the structural information of the long-term memory model, i.e., the spatial positioning of nodes conveys the structure of trees and the structure of the entire network.

The edges are colored based on their type. The parent-child edges are drawn with blue color and the fact edges are drawn with gray color. Furthermore, the color is used in nodes to show the activation value. The nodes are drawn with red color and the transparency value is used to differentiate their activation values. The transparency value is mapped to a range of numeric activation values. Hence, the nodes with a value high in the range of activation values are displayed with thick red color and the nodes with a activation value on the lower end of the range are displayed with very light red.

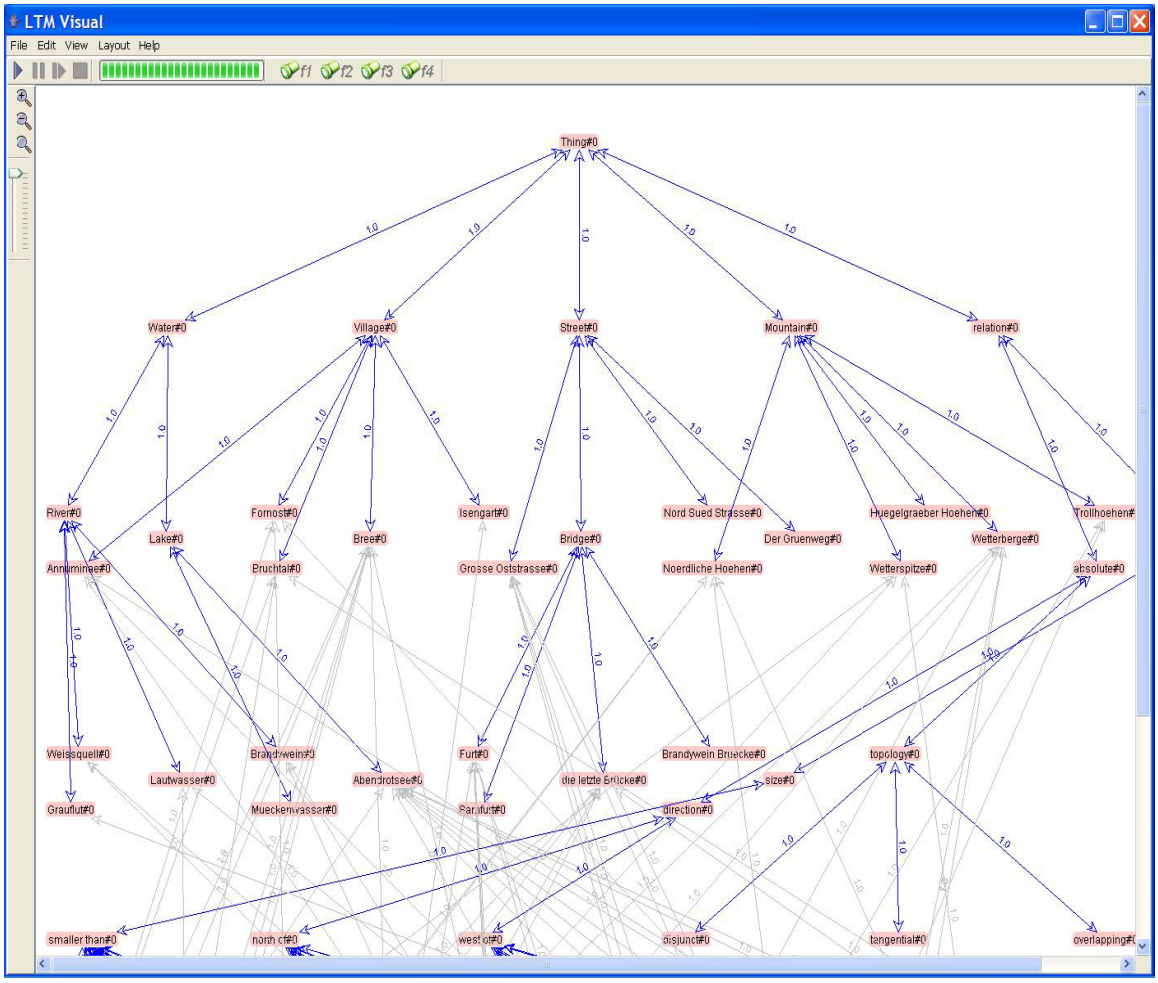


Figure 5.3: The LTMVisual Hierarchical information display.

The size parameter is used to differentiate between different edge weights. The edge widths are mapped to a range of numeric weight values. As the weight of an edge is increasing its width also increases.

Text conveys the information in a straight forward way. The edges are labeled to show the weight along with the edge width. The edge widths aid in comprehending the edge weight information by having different widths for different weights. Whereas, the label provides the exact value of weight an edge has. The long-term memory model represents real world spatial information. So, conveying the name of every node with labels eases the comprehension of information about what knowledge the long term memory contains at a given time. For example, in Figure 5.3, the root node “Thing#0” has the children “Village#0”, “Water#0”, “Street#0”, “Mountain#0”, and “Relation#0”. When users find

such a text based labels at root level in a tree, it becomes easier for them to interpret what the whole tree is about and to guess what would be the remaining nodes of the tree etc.

Along with the node label, text is used to show the node activation value. Two provisions have been provided to view the activation value a node has. One is to select the show activation value setting in layout menu at the start of visualization. When this menu item is selected the network is displayed with the nodes having labels composed of the name and the activation value. This is useful when the user wants to see the activation values of all the nodes every time. However, the increase of node's size and in turn the increase of tree size causes inefficient usage of space. The other provision is, the activation value of a node is shown in the tool tips and it is activated on placing the mouse pointer on the node. This is good as long as it is not required to remember the activation values of more than a few nodes. Otherwise, it requires a great cognitive effort of the user to remember the values as only one node activation value can be seen at any given time.

5.1.3 Distributed Activation Spreading

LTMVisual uses time based animation to show the knowledge retrieval from the long-term memory model in the form of distributed activation spreading process. As long as the long-term memory model is static, the edges and nodes are also static. But during the activation spreading, if a node spreads or gets the activation, all the edges connecting to the target nodes are replaced by ant-walking edges. The nodes participating in an instance of spreading activation are highlighted with dash-pattern boarder to be distinguishable from other nodes in the network. Besides, in the group of nodes, the single node which gets or spreads activation from or to a group of nodes is highlighted with green color along with dash-pattern boarder. This facilitates the highlighting of the node from the group of nodes in an instance of spreading activation process. The group of nodes participating in an activation spreading process only contains unidirectional links, i.e., one side arrow headed edges. The direction of the links describes the direction of activation flow.

An instance of activation spreading process in the long-term memory model is shown in Figure 5.4. As it can be observed form the figure the edges which spread activation are unidirectional and all other edges which remain static are bidirectional. The edges spreading activation are of dash-pattern and are animated to appear as ant walking pattern

edges. As it shown in the figure, at the current instance of activation spreading, the root node “Thing#0” is spreading activation to the nodes “Water#0”, “Village#0”, “Street#0”, “Mountain#0”, and “Relation#0”. The node “Mountain#0” is at thick red color foreground compared to the other nodes color to show that it posses a very high activation value compared to all other nodes. In the figure, the tool tip of the node “Water#0” contains the activation value of 0.5.

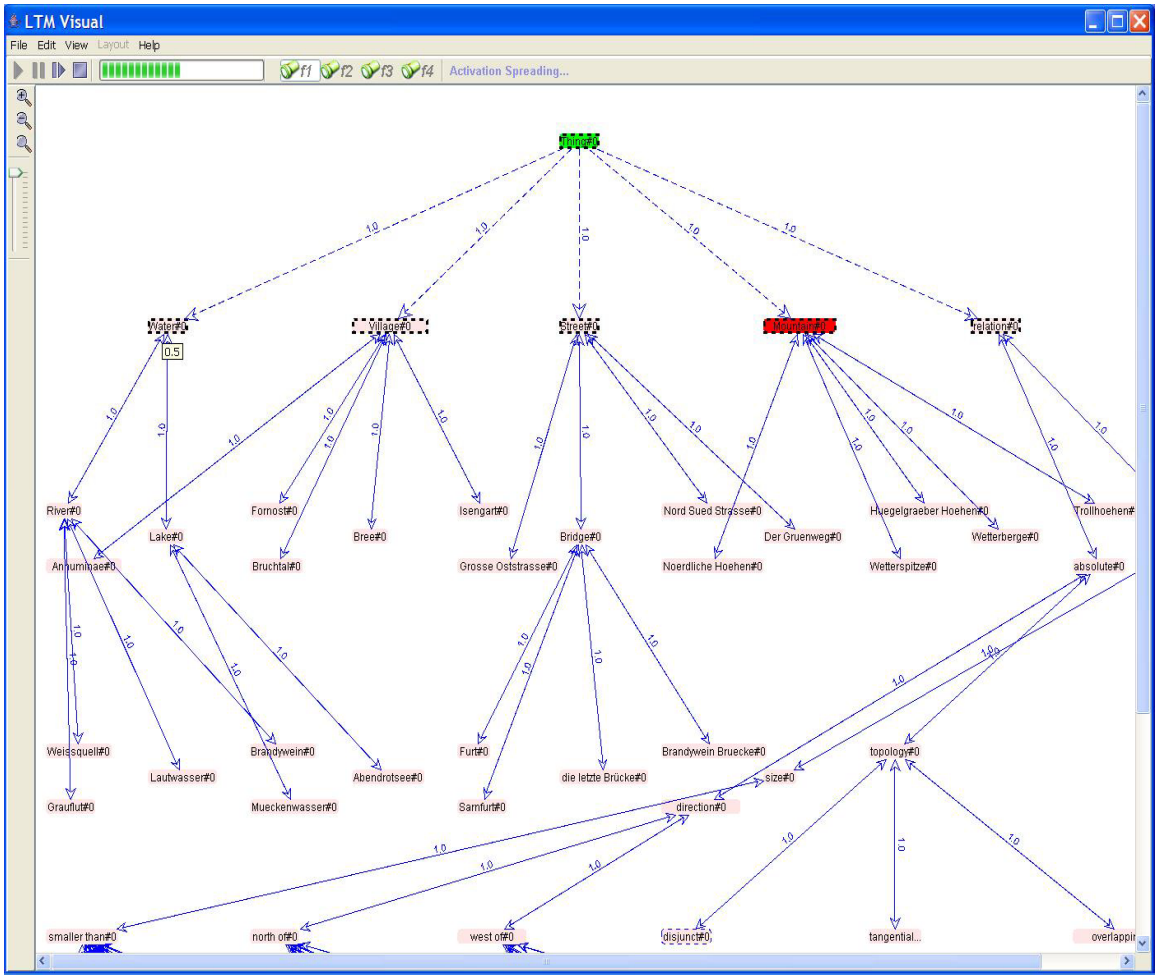


Figure 5.4: An instance of knowledge retrieval with activation spreading process

5.1.4 Information Navigation & Browsing

It is important to provide navigation facilities to explore and to be able to comprehend the information from the visual structures of a large long-term memory model. LTMVisual

has provisions to the user to facilitate the navigation features such as moving the view port, changing the size of network models, and inspecting one or more knowledge elements and their connectivity in the network model. These navigation features enable the user to switch between local details and surrounding context of large network models.

The concept of filters is incorporated in LTMVisual to accomplish the inspection of long-term memory model and is available via the filter buttons (f1, f2, f3, and f4; see above).

The hide fact links filter (f1), as the name implies, hides all the fact links and leaves only the parent-child links in the network model. This facilitates the user to focus on the structural information in the network as hierarchical relationship is only represented by parent-child links. Figure 5.5 shows a sample long-term memory model with the hide fact links filter (f1) on.

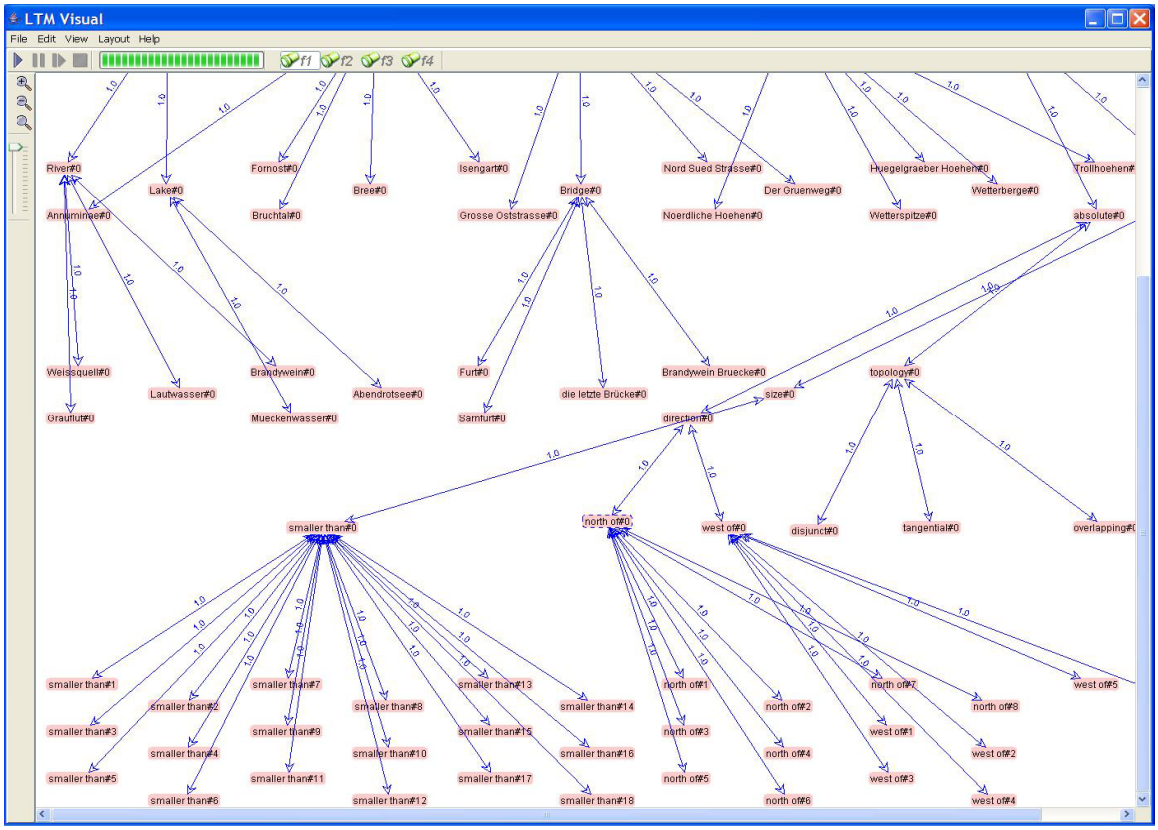


Figure 5.5: The LTMVisual network structure with the hide fact links filter (f1).

The second filter related to the structure is the hide parent-child links filter (f2). As it is implied by the name, it hides the parent-child links and shows only the fact links of the long-term memory model. By this, the user is provided with a facility to move attention from the structural information to the information related to fact links relationship. Figure 5.6 shows a sample long-term memory model with the hide parent-child links filter (f2) on.

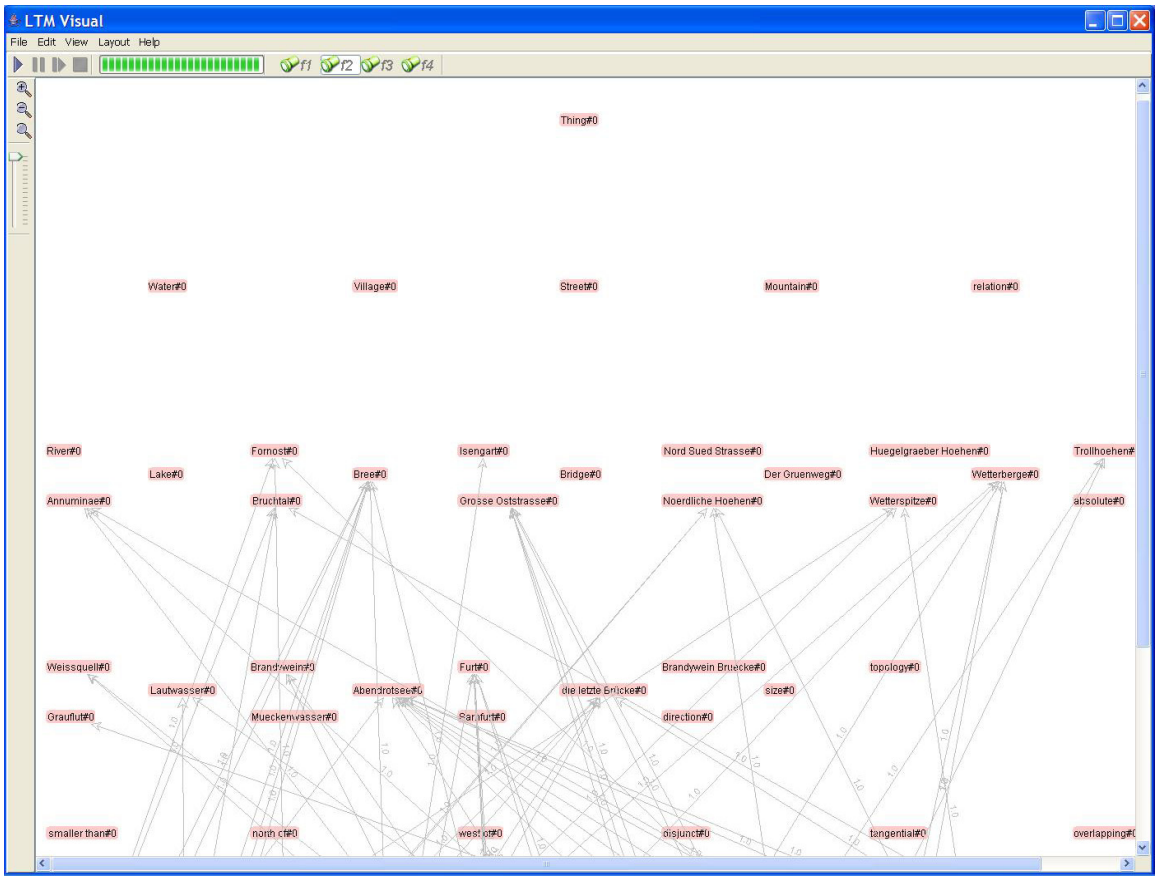


Figure 5.6: The LTMVisual with hide parent links filter (f2) on.

The above described hide fact links (f1) and hide parent-child links (f2) filters are concerned with overall structure of long-term memory model. They neither change the view of long-term memory model nor enable the focus on the sub parts of long-term memory model. In contrast, the hide non-selected nodes filter (f3) is mainly concerned with focusing on distinct parts of long-term memory models.

The hide non-selected filter (f3) is to be applied after the selection of one or more nodes in the network model. The application of this filter prepares a set of nodes containing the nodes selected and the nodes which are connected to the selected nodes. In addition to the set of the nodes filtered, all the links existing between them are shown on the view port. The selection of nodes can be done either by pressing ctrl and mouse left click over the nodes to be selected or by drawing a rectangle over the set of nodes to be selected.

A user can focus on specific parts of the network with selecting the nodes of interest and visualizing the network consisting of the selected nodes. Figure 5.7 shows a network model with the hide non-selected nodes filter (f3) on.

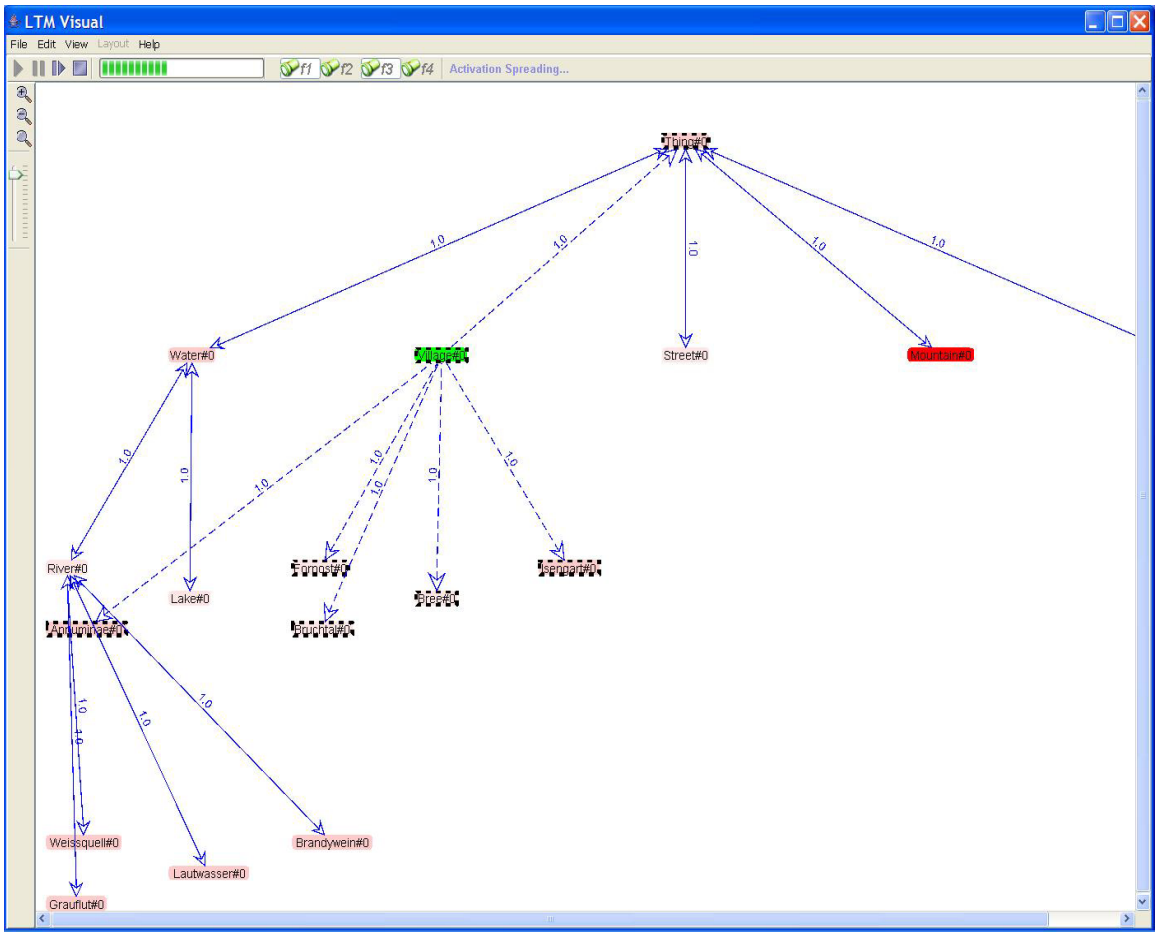


Figure 5.7: Activation spreading in a filtered network with the hide non-selected nodes filter (f3) and the hide fact links filter (f1) on.

The three filters can be applied in combination. For example, the application of filters f1 and f3 would show only the selected nodes and nodes connected to them with parent-child links. The application f2 and f3 would show only the selected nodes with fact links. Finally applying the combination of all the three filters would show only the selected nodes and nodes connected to them without any links. The filter reset filters (f4) is provided to reset all the filters and to show the whole network model. The filters can be applied while navigating a static knowledge representation or while visualizing the activation spreading in a long-term memory model.

The Figure 5.7 shows the activation spreading in a the long-term memory model shown in the Figure 5.5 after the application of the filters f1 and f3 with the selection of nodes “Village#0”, “Water#0”, “River#0”, and “Bree#0.

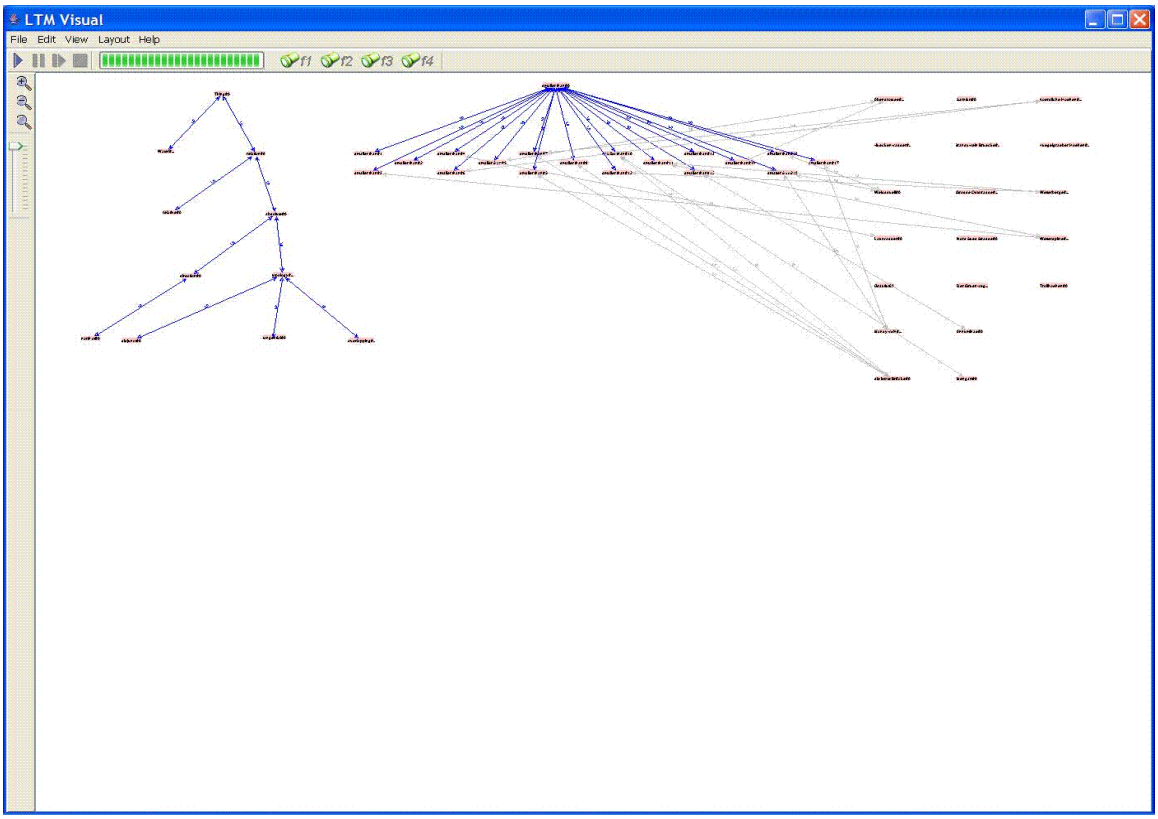


Figure 5.8: A network zoomed in to fit into the view port.

While enabling the examination of long-term memory models with the help of filters, LTMVisual has two other provisions to navigate the long-term memory model and to switch between local details and surrounding context. They are the zooming and

scrolling. Although the scrolling is primitive as it is only possible with the scrollbars of the view port, it is enough to navigate the whole network with out any problem.

When visualizing large networks, network models of size several times larger than the view port, it becomes hard for users to get an idea of overall network structure. The zoom out functionality can be used to make the whole network to fit into the view port. Although, the default layout is big enough to see the contents of a node clearly, some times it may be necessary to zoom out a network structure to see clearly the complicated parts of it, if any. In addition to the zoom in and zoom out buttons, the zoom has been provided to set the zoom to default level with a single click. A network which is zoomed in to fit into the view port is shown in Figure 5.8.

5.2 Design & Development of LTMVisual

5.2.1 Architecture of LTMVisual

LTMVisual is designed and developed to be an efficient and flexible application. To achieve this goal, the design of LTMVisual has been carried out with an examination of existing design patterns. A *design pattern* describes a proven solution for designing an application with an emphasis on providing a solution to the surrounding context problems of the application. After examining several design paradigms, I decided to use the *Model-View-Controller (MVC)* design pattern as a constructing paradigm for LTMVisual. The Model-View-Controller, shown in Figure 5.9, is a popular object oriented design paradigm which is defined by the SmallTalk environment [GOL89]. The goal of the MVC design pattern is to separate an application between the data it has, the way it presents the data, and the way it allows interaction with the data. To achieve the separation, MVC proposes to divide an application into three components, i.e., Model, View, and Controller which will be described in turn:

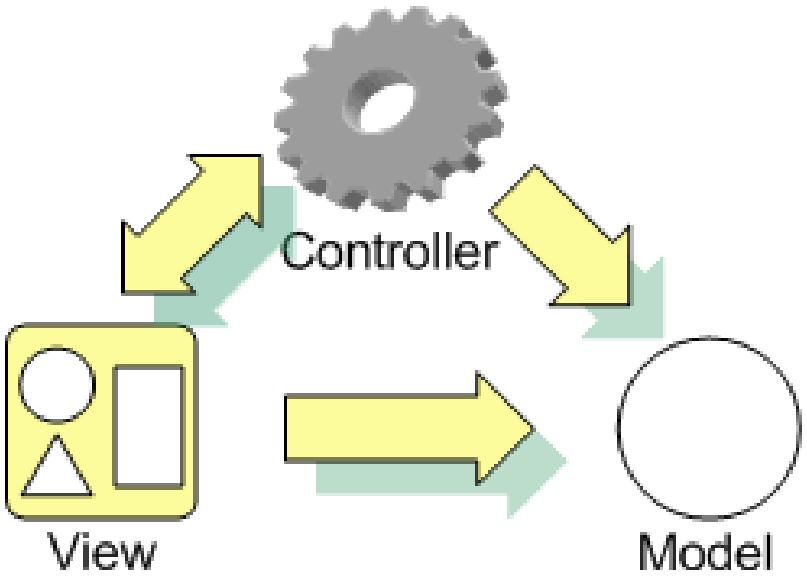


Figure 5.9: Overview of a Model-View-Controller Architecture [ECL05].

- Model: The core of an application which maintains the state and data that the application contains. It knows nothing about the way the data is displayed. A model is associated with one or more views. Whenever there is a significant change in the data of a model then the model updates all the views registered to it.
- View: It is the user interface which displays the information contained in the model to the user. The view has to be a registered object with the model to access the information.
- Controller: It handles the user interactions with the view. It transforms the user interactions on the view to the actions to the model.

The functionality of an application, when an application is designed founding on MVC architecture, is distributed between the model which contains the core functionality and the view which describes how the application's graphical user interface is designed. The main advantage of separating the data and display is that the data can be manipulated with out changing the display, and the display can be updated with new data only when the data is valid.

The arrow heads in the Figure 5.9 shows the direction of access between the components. The controller and view have unidirectional access to the model. Whereas the controller and view, themselves have a bidirectional access. In fact, it is to be observed that the

access is not always the same in all MVC architecture based applications. It is very specific to the application. An application architecture having a model object which takes the responsibility of updating the views can result in a bidirectional access between model and view in the architecture.

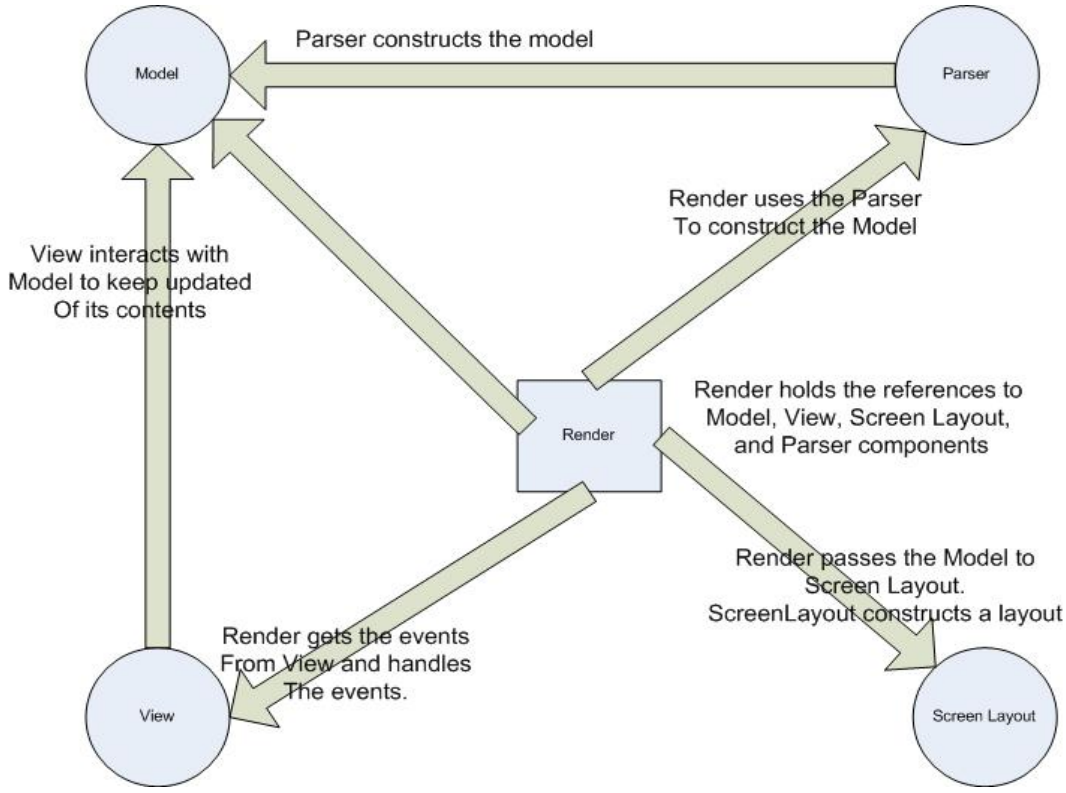


Figure 5.10: The LTMVisual Architecture.

The LTMVisual architecture developed from the notion of Model-View-Controller design pattern is shown in Figure 5.10. As shown in the figure, the LTMVisual is mainly comprised of the components *model*, *view*, *parser*, *screen layout*, and *render*. The term *component* is used in the sense of naming the encapsulation of a specific functionality. The separation of functionality between presentation layer and data layer made LTMVisual to be an application that can be easily changed to work with different user interfaces including 3D. The view component can be replaced with another view component to get a different user interface for LTMVisual.

As described in the figure, the render is the key component in LTMVisual gluing all other components together. The model contains all data structures required to hold the LTM

long-term memory model information. Render uses the Parser to parse the incoming text based node data and uses the data structures to hold the parsed data. With respect to the display, render passes the model to the screen layout component. The screen layout, in turn, generates a layout for the model to display on the screen. On the request from render, view generates all the graphical user elements to represent the information in the model. Finally, render displays the long-term memory model using the graphical user elements from view and layout information from screen layout component. Section 5.2.3 contains a detailed description of these components

5.2.2 LTMVisual Implementation

LTMVisual is programmed in java programming language with java standard edition 1.4.2 (J2SE). Java, being an object oriented programming language, encapsulates all the functionality into classes. The Java platform supplies *Java Foundation Classes (JFC)*, which encompasses a group of classes for building graphical user interfaces. JFC has two packages for user interface design named *swing* and *Abstract Windowing Toolkit* (AWT). Swing, relatively new and based on AWT, is more sophisticated and provides many built-in features compared to AWT. The classes in the swing package form the base classes of the LTMVisual user interface programming. All user interface components available in the swing package are developed with the Model-View-Controller architecture explained in the previous section. The swing package provides the controls right from simple buttons and labels to the complex panels and tables.

In addition to JFC, I looked into several open source java based graph or network visualization libraries for realizing graph structure. There exists several of this kind where each one is having emphasis on specific features. Some well known libraries I have considered are JUNG, HyperGraph, ZoomGraph, GVF, GINY, and JGraph. A brief description of the main focus in each of these libraries is:

- JUNG: stands for Java Universal Network/Graph Framework and provides a common and extendible language for the modeling, analysis, and visualization of data that can be modeled as a graph or network.
- HyperGraph: handles hyperbolic space from a purely mathematical point of view. And it provides a Swing panel that shows a graph or network view on the hyperbolic plane.

- ZoomGraph: with an emphasis on zooming it provides a zoomable interface to large graphs allowing the visualization of large graphs with a smooth zoom.
- GVF: GVF which is an abbreviation for Graph Visualization Framework is a set of java packages which serve as a foundation for applications that either manipulate graph structures or visualize them. These packages contain several modules for graph input, graph management, layout, and rendering. GVF has built in algorithms for layouts such as hierarchical, force-directed etc.
- GINY: is a graph library concentrating more on graph layout algorithms. These Algorithms include graph traversal algorithms like *all pairs shortest path* and graph layout algorithms like spring layout and hierarchical layout.
- JGraph: is an active open source library, robust, popular, written completely in java, and a complete graph component. It is java swing based component developed with Model-View-Controller architecture. It provides features such as drawing graphs, zooming, folding, undo, drag and drop etc.

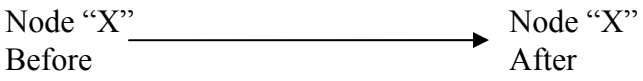
More or less most of the above mentioned graph libraries satisfy the requirements of LTMVisual. But I have chosen JGraph, because of its wide support available from a very active open source community and the ease to use the architecture. One major advantage of JGraph is its flexibility in embedding into an application. It can be used as a simple view providing graphical user elements or completely as a component handling all the functionality related to graph drawing. Besides its suitability for the needs of LTMVisual, having experience in using JGraph in one of my previous projects also made me to use JGraph library.

5.2.3 LTMVisual Components

In order to enhance the organization of classes in LTMVisual, all the classes are categorized to belong to one of the packages named *LTMModel*, *LTMRender*, and *LTMView*. However, from the functionality point of view, the classes belong to one of the components introduced in the section 5.2.1, which are parser, render, model, view, and screen layout. Each of these components is examined next. Appendix A contains the documentation of some core classes pertaining to these components.

5.2.3.1 Parser Component

The parser component contains the functionality required to read the long-term memory model data either from a text file or from another application generating the text representation of the knowledge structure and processes in the long-term memory model. The long-term memory model data is encoded in text format where each line of the text contains the data with respect to one node. The long-term memory model data is contained in nodes. Each node contains the data of its own and the data related to its connectivity with the remaining part of the Network model. The parser class is the core class of parser component. The format of each text line is as shown below



As shown above, each text line containing a node data is separated into two strings as node before and node after strings. These two strings are separated by the symbol “->”. The node before contains the current data of a node and node after contains the new data of the same node. Both the node before string and node after string contain the node data in the same format. Each string contains data as slot and value pairs where the slot represents an attribute of the node. Parenthesis “( )” are used to enclose a set of data such as weights, parents etc. the character “:” indicates the start of new slot. Space is used to separate the slots and separate the data within a slot. The string type data such as identifiers and names are enclosed in quotation marks.

The long-term memory model text data contains three different categories of instances. Each of these instances is explained below with example data.

First category: The below line of string specifies that new node with the identifier “Water#0” is added to the long-term memory model with the activation, trust, and base activation values equal to 1

“NULL -> (:IDENT "Water#0" :NAME "Water" :PARENTS NIL :PWEIGHT NIL :CHILDREN NIL :CWEIGHT NIL :FACTS NIL :WEIGHT NIL :ACTIVATION 1 :TRUST 1 :BASEACTIVATION 1) “

Second category: The below line of string specifies the node with identifier “Water#0” is removed from the long-term memory model.

“(:IDENT "Water#0") -> NULL”

The third and last category: The below line of string describes a change in the children of node identified as “Thing#0”. The node with identifier “Thing#0” gets the children with identifiers “Water#0”, “Village#0”, “Street#0”, “Mountain#0”, and “relation#0” with each link weight value equal to 1.

“(:IDENT "Thing#0") -> (:CHILDREN ("Water#0" "Village#0" "Street#0" "Mountain#0" "relation#0") :CWEIGHT (1 1 1 1 1))”

Along with the information in each line of text, to make it easier for parsing, the text data also contains the information (meta-data) about a block of lines of text. This information is encoded in the form of tags. The different tags used are “[BUILD]”, “[SPREAD]”, and “[STIMULATION]”. Each tag represents the type of data contained in the following lines of text to it until the next tag or until the end of the text. For example, the tag “[SPREAD]” represents the data contained in the following lines of text to it as the activation spreading data.

In relation to the user interaction, the render component invokes the parser component to read the text data. The parser uses the tags to decide the kind of information it is going to read, reads the text data line by line, and parses each line. The parsed data is passed to model component, which fills the data structures underlying it to hold the structure and spread activation in the network.

5.2.3.2 Model Component

The model component contains the data structures to hold the long-term memory model data and functionality to manipulate it. *NetworkModel*, *LTMNode*, and *LTMLink* are the core classes of LTMModel component.

A node is identified by its unique identifier given in the input text data to the parser. A link is identified by its predecessor identifier and successor identifier. The LTMNode

class maintains all the information pertaining to a node such as name, activation, base activation, trust parents, children, facts, and weights. The LTMLink class maintains all the information pertaining to a link such as predecessor identifier, successor identifier, link type, and link weight.

The model component holds the long-term memory model as a network of interconnected trees. A single node without any children is treated as a tree only having a root node. So a given node in the long-term memory model is a part of a hierarchical structure or an individual node not being part of any hierarchical structure. An object of LTMNode holds the data of a single node. An object of the NetworkModel class stores all the nodes in the form of a list of trees. The list is called *nodes*. The nodes list is sorted based on depth of the trees in the decreasing order of tree depths where the depth of a tree is equal to the longest distance between its root node to a leaf node in terms of the number of nodes. In the nodes list, a tree with N number of levels is preceded by either a tree with N number of levels or more than N number of levels. The individual tree structures are rebuild on adding new nodes, on deleting already existing nodes, and on updating the links of already existing nodes. The list of trees is sorted when a tree structure is added to the list or when a tree structure is rebuild.

The nodes list maintains the hierarchical structure with a trees based organization. However, the linkage information related to both parent-child links and fact links is maintained in another list of the LTMLink class objects. The list is called *links*. While parsing the text data, the connection information contained in each node is divided either as parent-child links or as fact links. All the links are added to the links list.

The NetworkModel object is not concerned with the way the long-term memory model it contains is displayed. Whenever a change in the structure of network model object takes place, it reports the change to the render component.

5.2.3.3 View Component

The view component is responsible for drawing all the graphical user interface elements on the view port. The *NetworkView*, *LTMGraphView*, *LTMEdgeView*, *LTMVertexView*, and *LTMGraphConstants* classes comprise the core classes of the view component.

LTMVertexView is a super class of all the vertex type classes. Every vertex type has its own view and rendering classes. The view class is responsible for defining attributes for a vertex and the rendering class is responsible to paint the vertex on the graph. All the view classes are sub-classes of vertexview class and the renderer classes are the subclasses of cellviewrenderer class.

A node rendering process takes the node shape, coordinates, background color, and foreground color values as parameters and draws the node. The foreground color corresponds to the activation value of a node. The foreground color transparency is computed using the activation value of the node. Edge drawing takes source and destination coordinates into account and draws a line of width basing on edge weight from source to destination.

Rendering of node labels is done based on the node types. For the nodes of type ellipse or round rectangular the node label is drawn inside the node. Whereas for the nodes of type text cells the labels are drawn outside a small rectangle. The edge weights are treated as the label of the edges.

JGraph contains a *Map* data structure to hold all the attributes as a list of key-value pairs where the key is the name of an attribute and the value is the current value of that attribute. These attributes are nothing but all the display properties of nodes, links, and graph. An attribute value can be changed by passing the name and the value of the attribute to JGraph where the JGraph simply updates the value of the attribute in the map data structure. After changing all the attribute values based on the requirement, the issuing of paint commands would redraw the entire network structure with new attribute values contained in the map.

Zooming is handled by the JGraph object. The current zoom ratio is always saved and whenever it is required to reset the value, the JGraph object is given the value of the default zoom ratio.

The view captures all the user interaction with the graph. All the node drag, mouse click, keyboard click events of graph are captured by JGraph object and passed to the render component to handle.

5.2.3.4 Screen Layout Component

The screen layout component is responsible for computing the layout of the network model. It gets the nodes list from NetworkModel and computes the coordinates for every node based on predefined criteria. *LTMScreenLayout* is the core class of this component. It is flexible enough to implement new layout schemes simply by adding a new method to the class LTMScreenLayout.

Currently the LTMVisual is using a layout scheme developed on the following criteria. The individual hierarchical structures in the network are displayed as to convey the structure information easily. On the other hand, it tries to achieve edge length minimization as far as possible.

Here are the most important steps in the layout algorithm:

- Several constants are declared hold the default values for the layout like maximum number of nodes in each layer, the horizontal & vertical distance between nodes, horizontal & vertical distance between trees, distance between levels & layers etc.
- One multidimensional array is created to store the meta-data about each tree. One dimension equals the number of levels in the tree and the second dimension is equal to the number of different meta-data variables such as number of nodes in each level.
- The nodes list is analyzed to compute the meta-data and the meta-data is stored in the array declared in the previous step. This meta-data includes the number of trees, depth of each tree, number of nodes in each level for each tree, and the level with maximum number of nodes in each tree etc.
- The whole list of trees in the nodes list is traversed to collect the trees one by one. In each tree the nodes are collected into a collection data structure with breadth first enumeration, i.e., the nodes at lower level are collected first compared to next higher level.
- Finally, using the meta-data contained in the multidimensional array, the coordinates are computed for each node using the default values for the layout stored in the constants declared in the first step.

The constants holding the default values can be changed to get different appearance for the same layout. The layout algorithm does not take the edge crossings into account. With calculated coordinates, the tree would result in fewer edge crossings when only the parent-child edges are present. However, display of fact edges would result in increase of edge crossings between parent-child edges and fact edges.

The screen layout reports changes to the network structure render component to redraw the network, only when there is a change in coordinates between already existing values and newly computed values of at least one node.

5.2.3.5 Render Component

The render component handles the entire visualization process from parsing and rendering of the network structure to the animation in activation spreading and handling the user interaction in the main window. It is comprised of *controller* and *LTMMainWindow* as core classes.

It issues all the visualization commands to all four components and coordinates the interaction between them. When LTMVisual is started then the render simply displays the main window. The main window propagates entire user actions takes place in it to the render component. The render interprets the events and takes the corresponding action. When the user starts the visualization, it initializes the parser component to parse and store the network data in the internal data structures. After the completion of parsing, the render starts rendering the node data one by one. Each node data is send to model, where the model analyzes the node data and functions as explained in the model component section.

The controller class maintains the information related to layout settings and filter settings. The information specific to the drawing attributes is passed to the view component before view returns the graphical elements for the network model. Besides, the render component uses the filter settings while rendering the final network model. Only the required graphical elements representing the filtered nodes and edges are drawn on the view port.

For example, the show non-selected nodes filter's algorithm works by getting the selected nodes information from JGraph object. Based on this, controller class checks the network model to get all the nodes connecting to the selected nodes. While rendering the graph, the controller class only selects the set of nodes, i.e., both selected nodes and connected nodes to the set of selected nodes. Along with the set of nodes, all the links existing between them are also rendered to the view port.

Besides rendering the model, controller class takes care of the entire animation process during the activation spreading with the objects of the Java Thread class.

Chapter 6
Conclusions & Further Work

In the scope of my master thesis I developed a visualization module, called LTMVisual, for the long-term memory model of the R1-[ImageSpace] project. The LTMVisual is developed with the intension to enable the user to be able to comprehend the graph structure and the distributed spreading activation process of the long-term memory model. To ensure this, LTMVisual provides an environment for the user to visualize knowledge elements of the model in showing identities, the relationships between the elements, the graph structure formed by the relative position of elements, and the activation spreading between the elements. In this chapter, I discuss the LTMVisual visualization and present the further work that can be done to improve the LTMVisual environment by extending existing features and introducing new features. I carry out my discussion with respect to the background presented in preceding chapters. Furthermore, this chapter provides a summary of the LTMVisual tool.

6.1 Conclusions

LTMVisual is implemented by using a node-link diagram as the visualization structure to show the network structure of the long-term memory model. The node-link diagram, commonly used to visualize trees and networks, facilitates the user to grasp the structural information of network structures without much cognitive effort.

The long-term memory model data is mapped to the network visual structure with the aid of network information elements position, color, size, text, and animation. LTMVisual emphasizes the graph structure by depicting knowledge elements of the model as simple labeled nodes and relationships between them as colored links. The colored links are used to distinguish between factual relationship and parent-child relationship. The nodes are colored in red color with different transparency value to convey activation value.

LTMVisual provides navigation tools to enable the user to navigate to different parts of the network structure. Furthermore, it provides filters to enable the user to focus on a subset of knowledge elements and to explore the knowledge structures of the model. It shows the activation spreading by animating the edges as ant-walking edges, the direction of spreading by arrow headed links, and the nodes participating in an instance of spreading process by highlighting these nodes.

A single object with a contrasting color to a group of similar colored objects enables the human cognitive system with preattentive processing of its color information. Although the usage of color to show node activation values in LTMVisual does not enable the processing of activation information preattentively, it enables the automatic processing at group level information. A group of nodes having the color with nearby transparency values enables the user, preattentively, to identify the areas of the network with same activation levels.

Instead of color, size is another parameter that can be used with nodes to encode the activation value. As the dimensions of nodes depend on label lengths, usage of size for activation value would require placing labels outside the nodes to be able to draw nodes size completely based on their activation value. Placing the label outside a node and mapping activation values to size, however, results in an increase of nodes size and in turn an increase of the size of the entire network leading, in the end, to an inefficient usage of display space. So, the node size parameter is not used to show activation information However, the size parameter is used with edges width to show their weight value.

The position of nodes in the network visual structure greatly influences how effectively the long-term memory model structural information is revealed. The layout implemented in LTMVisual draws the hierarchical structures of the model as trees. A node position in a tree easily conveys the information about its parents, children, and siblings. However, the tree shaped layout causes inefficient usage of space. One solution is replacing each tree with a nested tree-map and treating the individual nested tree-maps as nodes. However, it badly influences the comprehension of structural information and limits the possibility of showing the content information of nodes effectively. The nested tree-maps do not show any links between nodes. As nodes spread activation via links, having links is necessary to show the activation spreading process. Without links it is only possible to

show discrete changes in the activation values of nodes without showing which nodes spread activation to which nodes.

It is important in information visualization systems that the user has the possibility to customize the environment. LTMVisual is customizable at different levels: the user can customize the shape of nodes, the type of data to be contained in nodes, and can select between different layouts. Furthermore, the user can make a choice about whether or not to visualize the incremental layout.

LTMVisual gives the user with a moderate control over the visualization process. During the visualization of incremental layouts and activation spreading process, sometimes it is necessary to suspend the process to view the content information of the nodes and links. In addition, the user has can control the speed of the visualization at any point of time during the visualization process.

The long term memory model, because of its notion of two types of relationships between the knowledge elements, has a high probability to result in a high number of links even in knowledge structures containing small numbers of knowledge elements. Thus, the intersection of edges is inevitable in LTMVisual network visual structures. Although this may sometimes cause difficulties in recognizing the network structure, it does not lead to misinterpretation of the information as the arrow heads of edges specify which two nodes are connected by each edge.

LTMVisual can be converted into a 3D visualization environment without rebuilding it from scratch. Because of its MVC architecture which separates the presentation layer from the data layer, LTMVisual can be converted into to a 3D visualization tool for the long-term memory model by simply re-implementing its view component.

6.2 Further Work

LTMVisual has been developed with sufficient features to create a suitable environment for the purpose of visualization. But, it still has a great scope to be developed further by extending existing features and by introducing new features. In the following I will point out some aspects of LTMVisual which would profit from further improvement.

Along with visualizing the knowledge structures based on the model, users can be given the possibility to modify a knowledge structure to simulate different alternatives for the same structure. In addition to modifying the structures, these structures can be made available to the LISP module of R1-[ImageSpace] project. The modification of the structures can be realized by implementing editing features such as data editing and data creation. The data editing could include editing the data of nodes and links such as activation, trust, and weight. The data creation could include creation of new nodes and links. Furthermore, the modification of structures can be extended by including the features to delete nodes and links. The concept of location probes is, explained in Chapter 3, helpful in implementing the data editing and data creation features.

The zoom functionality of LTMVisual has been implemented to perform the zoom operation by taking the entire network as a single entity. It can be enhanced by allowing the zoom to be restricted to parts of a knowledge structure, so that only required knowledge elements can be selected and zoomed without causing much increase in the overall size of the structure. Furthermore, the current zoom is discrete, i.e., each click on the zoom related buttons modifies the zoom of structure with a numeric value. This can be changed in a way to zoom the structure smoothly as long as the zoom buttons are pressed.

Since the layout is vital in visualizing complex networks, LTMVisual flexible layout implementation allows addition of new layouts easily. Some of the layout schemes explained in Chapter 4 would make an efficient usage of space. The radial layout and circle layout can be added to LTMVisual in addition to the hierarchical layout. These layouts would help in visualizing large knowledge structures based on the model without much scrolling.

In addition to the positioning of nodes with automatic layouts, LTMVisual allows personalized positioning of the nodes which enables the user to move the nodes with the mouse to improve the visualization of the network in certain situations where the automatic layouts are not sufficient. However, the positioning only stays until the next instance of increment in the layout. So, indeed, this can be improved by preserving the positions of the nodes used by the user to use in further increments of layout.

Edge routing is also important along with node positioning in graph layouts. In LTMVisual, the edges are routed as straight lines between their two nodes connected at

each end. The straight-line routing does not cause any problems with tree links as they have only a small possibility of intersecting other nodes. But the fact links connecting nodes within a tree and connecting nodes between different trees are vulnerable to cause the intersections of the nodes and links. Although it requires a complicated algorithm requiring considerable processing time for complex networks, the edge routing to reduce intersections helps to minimize the misinterpretation of links information.

The concept of filters can be extended by adding new filters to provide further navigation and exploration possibilities. Other potentially useful filters are, for instance, showing selected node and all of its children up to leaves, showing a set of nodes connecting a set of links satisfying a constraint on weight value, and showing a set of nodes with a constraint on the activation, trust, or base activation values.

Bibliography

[AND83]:
Anderson, J. R.; A Spreading Activation Theory of Memory, *Journal of Verbal Learning and Verbal Behavior*, 22, 261-295; 1983

[AND02]:
Anderson, J. R.; *Cognitive Psychology and its Implications, Fifth Edition*; 2002

[BER67]
Bertin, J.; *Semiology of Graphics: Diagrams, Networks, Maps*; London: University of Wisconsin Press; 1967

[BRI91]
Brian Johnson, Ben Shneiderman; *Tree-Maps: A Space-Filling Approach to the Visualization of Hierarchical Information Structures*; University of Maryland, USA; 1991

[CHA99]
Chaomei Chen; *Information Visualization and Virtual Environements*; Berlin: Springer; 1999;

[CHI91]
Chimera R., Wolman K., and Shneiderman B.; Evaluation of three interfaces for browsing hierarchical tables of contents; Technical Report AR-TR-539, CS-TR-2620, University of Maryland, USA; 1991

[CLE88]
Cleveland W.S., McGill M.E.; *Dynamic Graphics for Statistics*; The Wadsworth and Brooks/Cole; 1988

[COL69]
Collins, A.M., & Quillian, M. R.; Retrival Time from Semantic Memory, *Journal of Verbal Learning and Verbal Behavior, 8, 240-247*; 1969

[COL75]
Collins, A. M., & Loftus, E. F.; A Spreading Activation Theory of Semantic Memory, *Psychological Review, 82, 407-428*; 1975

[COL00]
Colin Ware; *Information Visualization: Perception for Design*; Morgan Kaufmann; 2000

[DAV99]
David A. C.; Guidelines for Designing Information Visualization Application; Institutionen för Datavetenskap, Linköpings Universitet, Sweden; 1999

[ECL05]
Eclipse GEF API Specification;
http://publib.boulder.ibm.com/infocenter/wsad512/index.jsp?topic=/org.eclipse.gef.doc.isv/reference/api/overview-summary.html; Last Visited 07/2005

[GAR98]
Gary Geisler; Making Information More Accessible: *A Survey of Information Visualization Applications and Techniques*; 1998

[GOL89]
Goldberg & Robson; *Smalltalk: the Language and its Implementation*; Addison-Wesley 1989;

[GRE00]
Greg Parker, Glenn Franck, Colin Ware; Visualization of Large Nested Graphs in 3D: Navigation and Interaction; University of New Brunswick, CANADA; 2000

[HEA95]
Healey C. G., Booth K.S., Enns J.T.; High-Speed Visual Estimation Using Preattentive Processing; *ACM Transactions on Computer-Human Interaction*; 1995

[HER01]
Herbert Schildt; *The Complete Reference Java 2 Fourth Edition*; Tata McGraw-Hill; 2001

[JAR99]
Jarke J. van Wijk, Huub van de Wetering; Cushion Treemaps: Visualization of Hierarchical Information; *IEEE Symposium on Information Visualization (INFOVIS99)*

[JGR05]
JGraph Home Page;
www.jgraph.com; Last Visited 07/2005

[KIM88]
Kim M. F., Steven E. P., George W. Furnas; SemNet: Three-Dimensional Graphic Representations of Large Knowledge Bases; 1988

[KOF35]
Koffka Kurt; Principles of Gestalt psychology; Harcourt-Brace, New York; 1935

[LIS98]
List Carla; an Introduction to Information Research; Kendall/Hunt, Dubuque, USA; 1998

[MAC95]
MacEachren, M.; *How Maps Work, Representation, Visualization, and Design*; The Guilford Press, USA; 1995

[MAT97]
Matej Novotny; Visually Effective Information Visualization of Large Data; *Research Center for Virtual Reality and Visualization*; 1997

[NOR93]
Norman, D. A.; *Things that Make us Smart, Reading, MA*; Addison-Wesley; p. 43; 1993

[PLA86]
Playfair, W.; *The Commercial and Political Atlas*, London; 1786

[RAA02]
Raaijmakers, J. G. W., & Shiffrin, R. M.; Models of Memory, In Pashler, H., & Medin, D. (eds.) *Stevens Handbook of Psychology 3rd Edition, Vol. 2: Memory and Cognitive Processes, P- 43-76*; 2002

[REE04]
Reed Hunt R., Henry C. Ellis; *Fundamentals of Cognitive Psychology*, McGraw-Hill Publications; 2004;

[RIC95]
Richard A. B., Stephen G. E., Allan R.W.; *Visualizing Network Data*; 1995

[RIC97]
Richard B.; Concept Demonstration Metrics for Effective Information Visualization Visible Decisions Inc; 1997

[ROB91]
Robertson G. G., Mackinlay. J. D. and Card, S. K.; Cone Trees: Animated 3D Visualizations of Hierarchical Information. 1991

[SER04]
Serengul Smith; Machine Learning Techniques;
http://www.cs.mdx.as.uk/staffpages/serengul/clustering.htm; 2004

[SFB05]
SFB-TR8 – R1 Project Page; http://www.sfbtr8.uni-bremen.de/project/r1/; Last Visited 07/2005

[STE93]
Stephen G. Eick, Graham J. Wills; Navigating Large Networks with Hierarchies; AT&T Bell Laboratories; 1993

[STU99]

Stuart K. Card, Jock Mackinlay and Ben Shneiderman; *Readings in Information Visualization: Using Vision to Think (Morgan Kaufmann, Series in Interactive Technologies)*; London: Academic Press; pp.1-62; 1999

[THO91]
Thomas M. J. Fruchterman, Edward M. Reingold; Graph Drawing by Force-directed Placement; University of Illinois, USA; 1991

[THO02]
Thomas Rieger, Francesca Taponecco; Interactive Information Visualization of Entity-Relationship-Data; *Interactive Graphics Systems Group, Technische Universität Darmstadt Darmstadt*, Germany; 2002

[TRI86]
Triesman, A.; Features and Objects in Visual Processing; *Scientific American*; 1986

[TRI88]
Triesman A , Gormican S.; Feature Analysis in Early Vision: Evidence from Search Asymmetries, Psychological Review; 1988

[TUF83]
Tufte, E. R.; The Visual Display of Quantitative Information; Cheshire: Graphics Press; 1983

[UDR05]
uDraw(Graph) Project Page;
http://www.informatik.uni-bremen.de/uDrawGraph/en/home.html; *Last Visited* 07/2005;

Appendix A

Documentation of Core Classes

This appendix contains the documentation of some core classes of LTMVisual. These classes include Controller, Parser, LTMNode, LTMLink, LTMScreenLayout, LTMGraphConstants, and NetworkModel. This documentation has been generated from the *source code* and *code comments* of the classes in LTMVisual using the *JavaDoc* feature in Java2 Platform, Standard Edition, v 1.4.2.

Class Controller

Package

```
java.lang.Object
  └ ltmRender.Controller
```

All Implemented Interfaces:

java.lang.Runnable

```
public class Controller
extends java.lang.Object
implements java.lang.Runnable
```

This class controls the entire visualization process and handles the events from LTMMainWindow. It contains the objects of Network Model class and NetworkView class. Furthermore, it contains the parser and LTMScreenLayout objects to handle parsing and layout, respectively.

Nested Class Summary	
class	**Controller.EdgeAnimator** Internal class to handle all the edge animation for showing the activation spreading process. Implements the Runnable interface.

Constructor Detail

Controller

```
public Controller()
```

Creates a new instance of Controller

Method Detail

getNewGraphCell

```
publicorg.jgraph.graph.DefaultGraphCell
getNewGraphCell(java.lang.String id,double actVal,
int nStatus)
```

Gets a suitable graph cell from JGraph. Uses the id, activation value to render the cell.

Parameters:

`id` -

`actVal` -

`nStatus` -

---

getVertexType

```
public int getVertexType()
```

Returns the type of vertex for graph.

---

setVertexType

```
public void setVertexType(int vertexType)
```

Sets the type of vertex to vertexType parameter.

Parameters:

`vertexType` -

---

setHideFacts

```
public void setHideFacts(boolean setFlag)
```

Sets the hideFactsOn flag to the value of setFlag.

Parameters:

`setFlag` -

---

setHideParents

```
public void setHideParents(boolean setFlag)
```

Sets the hideParentsOn flag to the value of setFlag.

Parameters:

`setFlag` -

---

setShowSelNode

`public void setShowSelNode(boolean setFlag)`

Sets the showSelNodeOn flag to the value of setFlag.

Parameters:

`setFlag` -

---

setShowAll

`public void setShowAll(boolean setFlag)`

Sets the showAllOn flag to the value of setFlag.

Parameters:

`setFlag` -

---

setLayoutType

`public void setLayoutType(int layoutType)`

Sets the layoutType variable to the value layoutType parameter.

Parameters:

`layoutType` -

---

setShowIncrementalLayout

`public void setShowIncrementalLayout(boolean setFlag)`

Sets the showIncrementalLayout flag to the setFlag.

Parameters:

`setFlag` -

ShowActValue

`public boolean ShowActValue()`

Returns the showActValue flag value.

---

setShowActValue

`public void setShowActValue(boolean showFlag)`

Sets the showActValue flag to showFlag value.

Parameters:

`showFlag` -

---

getNetworkView

`public NetworkView getNetworkView()`

Returns the current networkView object.

---

getNetworkModel

`public NetworkModel getNetworkModel()`

Returns the current NetworkModel object.

---

getParser

`public Parser getParser()`

Returns the parser object.

---

run

`public void run()`

The implementation of run method from Runnable interface.

Specified by:

`run` in interface `java.lang.Runnable`

---

visualizeNetworkModel

`public int visualizeNetworkModel()`

Goes through the nodes list read from text file and adds the data changes to the network model. It may add new node or update existing node.

---

doBuildNet

`public int doBuildNet(LTMNode nData)`

Adds the node data contained in nData to the networkmodel and renders the model if it changes in structure or data.

Parameters:

`nData` -

---

doSpread

`public int doSpread(LTMNode nData)`

Adds the spreading data to the network model and changes the corresponding nodes on the screen. Furthermore, it starts the animation of edges.

Parameters:

`nData` -

---

doStimulation

`public int doStimulation(LTMNode nData)`

Adds the stimulation data contained in nData to the network model. And, highlights the corresponding vertex in the graph.

Parameters:

`nData` -

---

getCellsToShow

`public void` **`getCellsToShow`**`()`

Returns the set of cells to show based on the user selection in graph. The set of cells includes the selected vertex, the vertices that are directly connected to the selected set, and all the links existing between the final set of vertices adds the vertex to the cellsToShow arrayList.

---

renderNetworkModel

`public int` **`renderNetworkModel`**`()`

It renders entire graph on to the screen it goes through all the nodes and links existing in the network model. With respect to the selected filters, it renders the required cells on to the screen. It gets all the vertices from View component, and the entire layout from screenlayout object.

---

addNodeData

`public int` **`addNodeData`**`(LTMNode nData)`

This function checks the type of data contained in nData and initiates corresponding action based on the type of data.

Parameters:

`nData` -

---

selectFile

`public int` **`selectFile`**`()`

Opens a dialog for file selection and sets the file path in parser object.

---

readFile

`public void` **`readFile`**`()`

Initiates the parser object to read the user selected file.

---

modelExists

`public boolean` **`modelExists`**`()`

It checks the existence of the model. If not, based on the confirmation from user it selects a file and sends the file to parser to build the model.

---

startRender

`public int` **`startRender`**`()`

Starts the rendering process.

---

suspendRender

`public int` **`suspendRender`**`()`

Pauses the rendering process.

---

resumeRender

`public int` **`resumeRender`**`()`

Resumes the rendering process.

---

stopRender

`public int` **`stopRender`**`()`

Stops the rendering process.

---

applyF1

`public int` **`applyF1`**`()`

Applies the Hide fact Links filter and renders the model.

---

applyF2

`public int` **`applyF2`**`()`

Applies the Show Parent Links filter and renders the model.

---

applyF3

`public int` **`applyF3`**`()`

Applies the Show Selected Node Links filter and renders the model.

---

applyF4

`public int` **`applyF4`**`()`

Resets all the filters.

Class LTMNode

Package

`java.lang.Object`
  `└` **`ltmModel.LTMNode`**

All Implemented Interfaces:

java.lang.Cloneable

Direct Known Subclasses:

LTMObject, LTMRelation

```
public class LTMNode
extends java.lang.Object
implements java.lang.Cloneable
```

This LTMNode class contains all the data related a node and related to its connectivity in a in the network model. It contains all the methods to add and modify the data of a node.

Constructor Detail

LTMNode

```
public LTMNode()
```

Creates a new instance of LTMNode.

Method Detail

clone

```
public java.lang.Object clone()
```

An overriden method of Object class. Makes a copy of LTMNode object.

Overrides:

`clone` in class `java.lang.Object`

Returns:

Object

See Also:

`Object`

---

isPosUpdated

```
public boolean isPosUpdated()
```

Returns whether or not the position of the LTMNode object is updated.

---

toString

```
public java.lang.String toString()
```

The overridden method of Object class.

Overrides:

`toString` in class `java.lang.Object`

---

setDataType

```
public void setDataType(int dataType)
```

Sets the data type of LTMNode object.

---

getDataType

```
public int getDataType()
```

Returns the data type.

setAttrFlag

```
public void setAttrFlag(java.lang.String attribute,
                        int flag)
```

Sets the value of attribute parameter to the value of flag.

---

getAttrFlag

```
public int getAttrFlag(java.lang.String attribute)
```

Returns the value of attribute value from attribute map.

---

getAttrFlags

```
public java.util.ArrayList getAttrFlags()
```

Gets the list of values of all attributes.

---

setAttrFlags

```
public void setAttrFlags(int flag)
```

Sets the value of all attributes to the flag value.

---

getUpdAttributes

```
public java.util.ArrayList getUpdAttributes()
```

Gets the list of updated attributes.

---

getPosition

```
public java.awt.Point getPosition()
```

Returns the coordinates of the LTMNode object.

---

getLevel

```
public int getLevel()
```

Returns the level of LTMNode object

---

getPosX

```
public double getPosX()
```

Returns the x coordinate value

---

getPosY

```
public double getPosY()
```

Returns the Y coordinate value.

---

setPosition

```
public void setPosition(java.awt.Point point)
```

Sets the coordinates to the point argument

---

setPosition

```
public void setPosition(double x,
                        double y)
```

Sets the coordinates to the x and y parameters

---

setLevel

```
public void setLevel(int level)
```

Sets the level

---

getName

```
public java.lang.String getName()
```

Returns the name of LTMNode

---

getId

```
public java.lang.String getId()
```

Returns the id of LTMNode

---

setStatus

```
public void setStatus(int status)
```

Sets the status

---

getStatus

```
public int getStatus()
```

Returns the status

---

getActivation

```
public double getActivation()
```

Returns the activation value of LTMNode

---

getBaseActivation

```
public double getBaseActivation()
```

Returns the baseactivation value of LTMNode

---

getTrust

```
public double getTrust()
```

Returns the trust value of LTMNode

---

getType

```
public int getType()
```

Returns the type of LTMNode object

---

setType

```
public void setType(int type)
```

Sets the type of LTMNode object

---

setName

```
public void setName(java.lang.String name)
```

Sets the name of LTMNode

---

setId

```
public void setId(java.lang.String id)
```

Sets the id of LTMNode

---

setActivation

```
public void setActivation(double activation)
```

Sets the activation value of LTMNode

---

setBaseActivation

```
public void setBaseActivation(double baseActivation)
```

Sets the base activation of LTMNode

---

setTrust

```
public void setTrust(double trust)
```

Sets the trust value of LTMNode

---

getParents

```
public java.util.ArrayList getParents()
```

Return the parents in an arraylist when there are no parents return an empty list

---

getChildren

```
public java.util.ArrayList getChildren()
```

Returns the children of a node in arraylist when there are no children, returns an empty list

---

getFacts

```
public java.util.ArrayList getFacts()
```

Returns the nodes connected with fact links in an arraylist when there are no facts, returns an empty list

---

getObjectAuen

```
public java.util.ArrayList getObjectAuen()
```

Returns the objectauen of a node in an arraylist when there are no objectauen, returns an empty list

---

setParents

```
public void setParents(java.util.ArrayList parArray)
```

Sets the parents of a node with the list of nodes in the arraylist

---

setChildren

```
public void setChildren(java.util.ArrayList childrenArray)
```

sets the children of a node with the list of nodes in the arraylist

---

setFacts

```
public void setFacts(java.util.ArrayList factArray)
```

Sets the facts of a node with the list of nodes in the arraylist

---

setObjectAuen

```
public void setObjectAuen(java.util.ArrayList objectOuenArray)
```

Sets the objectauen of a node with the list of nodes in the arraylist

---

copyUpdData

```
public void copyUpdData(LTMNode source)
```

Copies the data of updated attributes from the source node object to this node

---

copyFlags

```
public void copyFlags(LTMNode source)
```

Copy flags from source node this node

setParentIds

`public void` **`setParentIds`**`(java.util.ArrayList parIdArray)`

Sets the parents of a node with the list of nodes ids in the arraylist

---

addParent

`public void` **`addParent`**`(LTMNode parentNode)`

adds a parent to a node with the node parentnode in the arraylist

---

addParent

`public void` **`addParent`**`(java.lang.String parID)`

Adds a parent of a node with the node parent node id in the arraylist

---

clearParents

`public void` **`clearParents`**`()`

Clears the parents list

---

removeParent

`public void` **`removeParent`**`(java.lang.String parID)`

Removes the node with parId from the parents list

---

removeParent

`public void` **`removeParent`**`(LTMNode parNode)`

Removes the node parNode from the parents list

---

setChildrenIds

`public void` **`setChildrenIds`**`(java.util.ArrayList childIdArray)`

Sets the children nodes with the node ids in the array

---

addChild

`public void` **`addChild`**`(LTMNode childNode)`

Adds a child node

Parameters:

`childNode` -

---

addChild

`public void` **`addChild`**`(java.lang.String childID)`

Adds a child node specified by the id

Parameters:
`childID` -

---

clearchildren

`public void` **`clearchildren`**`()`

Clears the children collection

---

removeChild

`public void` **`removeChild`**`(java.lang.String childID)`

Removes a child node specified by childID

Parameters:
`childID` -

---

removeChild

`public void` **`removeChild`**`(LTMNode childNode)`

Removes the childNode

Parameters:
`childNode` -

---

setFactIds

`public void` **`setFactIds`**`(java.util.ArrayList factIdArray)`

Sets the fact ids

Parameters:
`factIdArray` -

---

addFact

`public void` **`addFact`**`(LTMNode factNode)`

Adds a fact Node

Parameters:
`factNode` -

---

addFact

`public void` **`addFact`**`(java.lang.String factID)`

Adds a fact node specified by factID

Parameters:
`factID` -

---

clearFacts

```
public void clearFacts()
```

Clears the fact collection

---

removeFact

```
public void removeFact(java.lang.String factID)
```

Removes the factNode specified by factID

Parameters:

`factID` -

---

removeFact

```
public void removeFact(LTMNode factNode)
```

Removes the factNode

Parameters:

`factNode` -

---

setObjectAuenIds

```
public void setObjectAuenIds(java.util.ArrayList objectAuenIdArray)
```

Adds the objectAuen specifed by objectAuenIdArray

Parameters:

`objectAuenIdArray` -

---

addObjectAuen

```
public void addObjectAuen(LTMNode objectAuenNode)
```

Adds the objectAuenNode

Parameters:

`objectAuenNode` -

---

addObjectAuen

```
public void addObjectAuen(java.lang.String objectAuenID)
```

Adds the objectAuenNode specified by objectAuenID

Parameters:

`objectAuenID` -

---

clearObjectAuen

```
public void clearObjectAuen()
```

Clears the objectAuen Collection

---

removeObjectAuen

`public void` **`removeObjectAuen`**`(java.lang.String objectAuenID)`

Removes objectAuen Node specified by objectAuenID

Parameters:

`objectAuenID` -

---

setParentWeights

`public void` **`setParentWeights`**`(java.util.ArrayList weightArr)`

Sets the weight of all parents with the weights in weightArr

Parameters:

`weightArr` -

---

setChildrenWeights

`public void` **`setChildrenWeights`**`(java.util.ArrayList weightArr)`

Sets the weight of all children with the weights in weightArr

Parameters:

`weightArr` -

---

setFactWeights

`public void` **`setFactWeights`**`(java.util.ArrayList weightArr)`

Sets the weight of all facts with the weights in weightArr

Parameters:

`weightArr` -

---

setObjectAuenWeights

`public void` **`setObjectAuenWeights`**`(java.util.ArrayList weightArr)`

Sets the weight of all objectAuen with the weights in weightArr

Parameters:

`weightArr` -

---

getParConnectionWeight

`public double` **`getParConnectionWeight`**`(java.lang.String parID)`

Returns the weight of the connection between this node and the node specified in parID

Parameters:

`parID` -

---

getParConnectionWeight

`public double` **`getParConnectionWeight`**`(LTMNode parNode)`

Returns the weight of the connection between this node and the parNode

Parameters:

`parNode` -

---

getChildConnectionWeight

`public double` **`getChildConnectionWeight`**`(java.lang.String childID)`

Returns the weight of the link between this node and the node specifed by the childID

Parameters:

`childID` -

---

getChildConnectionWeight

`public double` **`getChildConnectionWeight`**`(LTMNode childNode)`

Returns the weight of the link between this node and the childNode

Parameters:

`childNode` -

---

getFactConnectionWeight

`public double` **`getFactConnectionWeight`**`(LTMNode factNode)`

Returns the weight of the link between this node and factNode

Parameters:

`factNode` -

---

getAuenConnectionWeight

`public double` **`getAuenConnectionWeight`**`(java.lang.String auenID)`

Returns the weight of the link between this node and object auen node specified by auenID

Parameters:

`auenID` -

---

getAuenConnectionWeight

`public double` **`getAuenConnectionWeight`**`(LTMNode auenNode)`

Returns the weight of the link between this node and auenNode

Parameters:

`auenNode` -

Class LTMLink

Package

```
java.lang.Object
  └ltmModel.LTMLink
```

```
public class LTMLink
extends java.lang.Object
```

A class to encapsulate the data of a link. The links are identified by their predecessor and successor ids

Constructor Detail

LTMLink

```
public LTMLink()
```

Empty Constructor

LTMLink

```
public LTMLink(java.lang.String pred,
               java.lang.String succ)
```

Constructs a link with both ids

Parameters:

`pred` -

`succ` -

LTMLink

```
public LTMLink(java.lang.String pred,
               java.lang.String succ,
               double weight)
```

Constructs a link with weight along with ids

Parameters:

`pred` -

`succ` -

`weight` -

LTMLink

```
public LTMLink(java.lang.String pred,
               java.lang.String succ,
               double weight,
               int type)
```

Constructs a link with type and weight along with ids

Parameters:

pred -

succ -

weight -

type -

Method Detail

equals

```
public boolean equals(LTMLink edge)
```

Checks whether or not this link and the link in the edge parameter are same

Parameters:

edge -

---

getType

```
public int getType()
```

Returns the type of the link

---

setType

```
public void setType(int type)
```

Sets the type of the link

Parameters:

type -

---

getlabel

```
public java.lang.String getlabel()
```

Returns the name of LTMLink

---

getId

```
public java.lang.String getId()
```

Returns the id of LTMLink

---

getStatus

```
public int getStatus()
```

Returns the status of the link

---

setStatus

```
public void setStatus(int status)
```

Sets the status of the link

---

setLabel

```
public void setLabel(java.lang.String label)
```

Sets the label of the link

---

setId

```
public void setId(java.lang.String id)
```

Sets the id of the Link

Parameters:

`id` -

---

getWeight

```
public double getWeight()
```

Returns the weight of the link

---

setWeight

```
public void setWeight(double weight)
```

Sets the weight of the link

Parameters:

`weight` -

---

setSucc

```
public void setSucc(java.lang.String succ)
```

Sets the successor id

Parameters:

`succ` -

---

setPred

```
public void setPred(java.lang.String pred)
```

Sets the predecessor id

Parameters:

`pred` -

---

getSucc

```
public java.lang.String getSucc()
```

Gets the successor id

---

getPred

```
public java.lang.String getPred()
```

Gets the predecessor id

Class LTMScreenLayout

Package

```
java.lang.Object
  └ ltmRender.LTMScreenLayout
```

```
public class LTMScreenLayout
extends java.lang.Object
```

This class handles all the layout functionality of LTMVisual. A New layout can be added to this class simply by adding a new method. Currently it has two layouts implemented

Constructor Detail

LTMScreenLayout

```
public LTMScreenLayout()
```

constructor

Method Detail

getFlatLayout

```
public void getFlatLayout(NetworkModel networkModel)
```

Calculates screen positions for all the nodes in the network model with flat layout

Parameters:

`networkModel` -

---

getSquareLayout

```
public void getSquareLayout(NetworkModel networkModel)
```

Calculates screen positions for all the nodes in the network model with square layout, i.e., hierarchical layout default layout of LTMVisual

Parameters:

`networkModel` -

Class Parser

Package

```
java.lang.Object
  └ltmRender.Parser
```

```
public class Parser
extends java.lang.Object
```

This class handles all the parsing of files or text data available as an input to LTMVisual. Parses the data and keeps it in ltmNetwork object to be used in controller class. Implemented to parse the node strings either from a text file or from a network

Constructor Detail

Parser

```
public Parser()
```

Creates a new instance of Parser

Method Detail

parseFile

```
public LTMData parseFile()
```

Starts the parsing of a file available in filepath

---

getLTMNetwork

```
public LTMData getLTMNetwork()
```

Returns the ltmNetwork object

ReadFile

```
public int ReadFile()
```

Reads the network data from a text file

ReadNodeData

```
public LTMNode ReadNodeData(java.lang.String nodeString)
```

Reads the node data from the node string. The node string can be passed from a text file or via the network when LTMVisual is interfaced with the LISP module

Parameters:

nodeString -

---

getNode

```
public void getNode(java.lang.String strNode,
                    LTMNode node)
```

Parses the string in strNode parameter and adds it to the node object

Parameters:

strNode -

node -

---

getStrArray

```
public java.util.ArrayList getStrArray(java.lang.String str)
```

Parses the string containing the ids of parents, children, facts etc returns a string array with all the ids

Parameters:

str -

---

getNumArray

```
public java.util.ArrayList getNumArray(java.lang.String str)
```

Parses the string containing the values of weights of parents, children etc returns a double array containing all the values

Parameters:

str -

---

setFilePath

```
public void setFilePath(java.lang.String filePath)
```

Sets the file path with the path containing in filePath parameter

Parameters:

filePath -

---

getFilePath

```
public java.lang.String getFilePath()
```

Returns the path of the current file being parsed

Class LTMGraphConstants

Package

java.lang.Object

```
└org.jgraph.graph.GraphConstants
    └ltmView.LTMGraphConstants
```

```
public class LTMGraphConstants
extends org.jgraph.graph.GraphConstants
```

Subclass of the JGraph's GraphConstants class and contains all the values for network attributes

Constructor Detail

LTMGraphConstants

```
public LTMGraphConstants()
```

Method Detail

setCellHighlighted

```
public static final void setCellHighlighted(java.util.Map map,
                                             boolean flag)
```

Sets the selectable attribute in the specified map to the specified value. This determines whether or not a cell needs to be highlighted

---

setLinePattern

```
public static final void setLinePattern(java.util.Map map,
                                         boolean flag)
```

Sets the linepattern attribute in the specified map to the specified value.

---

getLinePattern

```
public static final boolean getLinePattern(java.util.Map map)
```

Returns the line pattern attribute value

Parameters:

`map` -

---

setPortHighlighed

```
public static final void setPortHighlighed(java.util.Map map,
                                            boolean flag)
```

Sets the selectable attribute in the specified map to the specified value. This determines whether or not a port needs to be highlighted

---

isPortHighlighted

```
public static final boolean isPortHighlighted(java.util.Map map)
```

Returns the status of the port whether it is highlighted or not

Parameters:

`map` -

---

isCellHighlighted

```
public static final boolean isCellHighlighted(java.util.Map map)
```

Returns the status of a cell whether it is highlighted or not

Parameters:

`map` -

---

setPortBackground

```
public static final void setPortBackground(java.util.Map map,
                                           java.awt.Color color)
```

Sets the background color for port

Parameters:

`map` -

`color` -

---

setPortWidth

```
public static final void setPortWidth(java.util.Map map,
                                      int width)
```

Sets the value of port width

Parameters:

`map` -

`width` -

---

setPortHeight

```
public static final void setPortHeight(java.util.Map map,
                                       int height)
```

Sets the value of port height

Parameters:

`map` -

`height` -

---

getPortWidth

```
public static final int getPortWidth(java.util.Map map)
```

Returns the port width value from the map

Parameters:

`map` -

---

getPortHeight

```
public static final int getPortHeight(java.util.Map map)
```

Returns the port height value from the map

---

getPortBackground

```
public static final java.awt.Color getPortBackground(java.util.Map map)
```

Returns the port background attribute value from the map

Parameters:

`map` -

Class NetworkModel

```
java.lang.Object
  └ ltmModel.NetworkModel
```

```
public class NetworkModel
extends java.lang.Object
```

This class contains the internal model of the LTM model. Contains a list of trees and edges the nodes are organized as trees. A single node is treated as a tree with a single root node

---

Nested Class Summary

class	**NetworkModel.TreeComparator** Inner class to compare two trees based on their depth implements the Comparator interface overrides the compare function to compare the trees based on their depth

Constructor Detail

NetworkModel

```
public NetworkModel()
```

Initializes the nodes and edges collections

Method Detail

isNetChanged

`public boolean **isNetChanged**()`

Returns whether there is a change in the data of the network model compared to the previous instance

---

setNetChanged

`public void **setNetChanged**(boolean val)`

Sets the network change status to val

Parameters:

`val` -

---

getNodes

`public java.util.LinkedList **getNodes**()`

Returns all the trees

---

getEdges

`public java.util.HashMap **getEdges**()`

Returns all the edges

---

clearNetworkModel

`public void **clearNetworkModel**()`

Clears the network model, i.e., clears both the nodes and edges lists

---

addNode

`public void **addNode**(LTMNode nData)`

Adds a node to nodes network. For the node without any parent, creates the tree and for the node with a parent, adds it to the corresponding parent in the tree

Parameters:

`node` -

---

updateNode

`public void **updateNode**(LTMNode nData)`

Updates the data of a node with nData and also updates the tree structure if there is a change in the parents or children information. Furthermore, sets the status of network model to changed status

Parameters:

`nData` -

---

copyUpdData

```
public void copyUpdData(LTMNode target,
                        LTMNode source)
```

Copies the updated attributes of source to target

Parameters:

`target` -

`source` -

---

getTreeNode

```
public                                  org.jgraph.graph.DefaultGraphCell
getTreeNode(java.lang.String nodeId)
```

Gets the tree node in the network model

Parameters:

`node` -

---

getNode

```
public LTMNode getNode(java.lang.String nodeID)
```

Gets the tree node in the network model with nodeID

Parameters:

`nodeID` -

---

nodeExists

```
public boolean nodeExists(java.lang.String nodeId)
```

Searches the networkmodel trees for the node

Parameters:

`node` -

Returns:

true if node exist otherwise false

---

addEdges

```
public void addEdges(LTMNode nData)
```

Reads all the edges in the node adds them to the edges list

Parameters:

`node` -

---

updateEdges

```
public void updateEdges(LTMNode nData)
```

Updates the edges collection with nData information

Parameters:

`nData` -

---

addEdge

```
public void addEdge(LTMLink edge)
```

Adds the edge to edges collection

Parameters:

`edge` -

---

edgeExists

```
public boolean edgeExists(LTMLink edge)
```

Checks whether an edge with same predecessor id and successor id is existing in edges collection

Parameters:

`edge` -

---

edgeExists

```
public boolean edgeExists(java.lang.String predId,
                          java.lang.String succId,
                          int eType)
```

Checks the edges collection if an edge with predId and succId with eType existing

Parameters:

`predId` -

`succId` -

---

removeEdges

```
public void removeEdges(java.lang.String nodeId,
                        int eType)
```

Removes all the edges starting from the node of the given eType

---

removeEdges

```
public void removeEdges(java.lang.String nodeId)
```

Removes all the edges starting from the node

---

removeNode

```
public void removeNode(LTMNode node)
```

Removes the node from nodes tree collection

Parameters:

`node` -

Appendix B

CD-ROM

A CD-ROM is accompanied with this thesis and can be found inside the back cover. The data on the CD-ROM is organized as follows: It contains three folders named LTMVisual, Doc, and Thesis

- The folder LTMVisual contains all the source code files of LTMVisual including the libraries.
- The folder Doc contains the documentation of all the classes of LTMVisual in HTML format
- The folder Thesis contains a soft copy of my thesis in PDF format.