Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Global Information Systems Group
Prof. Moira C. Norrie

Supervisor: Beat Signer

Diploma Thesis

Visualization of Trails and Tours

Corsin Decurtins, March 2002

Acknowledgments

This document is part of my diploma thesis in the Global Information Systems Group of Prof. Moira C. Norrie at the Swiss Federal Institute of Technology in Zurich.

I would like to thank Beat Signer for his supervision and advice. With his uncomplicated and helpful support, he contributed a lot to this work.

Another big thank goes to my fellows in misery Sabine Keuser, Ljiljana Vukelja, Urs Hardegger, Daniel Müller and Hugo Hugosson. Our coffee break discussions helped to keep a clear mind.

Special thanks go to Andrea Lombardoni (Systems Administrator of the Global Information Systems Group) for setting up a new machine for me in record time and to Jason Brazile for pointing out the worst mistakes in my report.

And last but not least, I would like to thank Bettina Schrag for keeping me attached to the world beyond bits and bytes.

Zurich, March 4, 2002

Corsin Decurtins

Eidgenössische Technische Hochschule Zürich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

Diploma Thesis, Winter Term 2002

Visualization of Trails and Tours

Corsin Decurtins (D-INFK)

Introduction

With the increasing number of documents available nowadays in databases and on the internet, the cross-linking of these documents becomes more and more important. In our group, various systems have been developed to address this issue. One of these systems is the Intelligent Caching Proxy, an HTTP proxy with a generic core that supports caching and prefetching based on access patterns assembled by the proxy. These access patterns can also be used to provide additional linking information for the documents.

Objectives

The aim of the project was to do surveys on both graph visualization techniques and existing hypertext systems and to design and implement a generic and extensible visualization framework for interlinked objects based on the results. A prototype application for web documents was to be developed based on this framework and the Intelligent Caching Proxy application.

Results

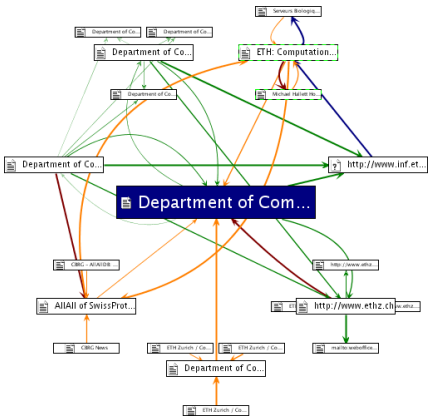
We present TrailGuide, an Applet add-on for web browsers. It provides a graphical user interface that supports visualization and manipulation of web documents and links. TrailGuide is based on a generic visualization component and an XML-RPC based protocol for the retrieval and update of graph data. The architecture of the application was specially designed to be extendibles for other document types.

Outlook

The TrailGuide application could be extended for the support of other document types. A very interesting candidate are database objects from the object-oriented database management system OMS developed in the Global Information Systems Group. Another issue is the implementation of additional graph layout algorithms.

Contact

Beat Signer
Global Information Systems Group
IFW D 46.2
ETH Zentrum
CH-8092 Zurich
Phone: +41 1 632 06 76
Email: signer@inf.ethz.ch
http://www.globis.ethz.ch/



Contents

1 Introduction

The internet is the biggest collection of information that has ever been established in the history of mankind. You can basically get information about anything from bleeding edge news about the latest development in theoretical physics to ancient vegetarian recipes of the romans.

So the problem nowadays is usually not to get access to a piece of information, but to find it and to sort out all the irrelevant information surrounding it. There have been many proposals on how to solve this problem. For this diploma thesis, we will focus on three of them: *Interlinked Objects*, *Visualization* and *Knowledge Transfer*.

1.1 Interlinked Objects

The most important innovation of the *World Wide Web* is the concept of *hypertext*. This concept has already been used in earlier systems, but the big break-trough was clearly the World Wide Web. Hypertext enhances conventional documents by adding the possibility to define links for certain parts of the document. If for example the name of a person appears in a document, it can be used as anchor for a link to a document with detailed information about this person or maybe a photography. If someone is reading the first document and wants more information about the person, he can follow the link to the more detailed document.

For experienced World Wide Web users of course, the concept of hypertext might look pretty simple. And it is actually simple, but its impact for the organization of information was enormous. Just imagine the World Wide Web without links. We could still access all the documents by entering the URL manually into the web browser. So the amount of information would still be the same but the troublesome access would make it much more difficult to find specific information.

The ability to link parts of documents to other documents is really nice. But just for now, we would like to use an even simpler model for hypertext. Let us interpret the World Wide Web as a collection of documents, where some documents are linked to other documents. Which part of the documents serve as anchors for the link is not important for the moment. The power of hypertext with respect to finding relevant information can even be seen with this very simple model. Once we have found a relevant document, we can find other relevant documents by following the links. So links are basically pointers to other documents with similar content.

This concept of *interlinked objects* can also be used for other information sources than the World Wide Web. A more abstract example is a database. Documents in this case would be queries to the database or the results of these queries respectively. We can set two queries in relation to each other by adding a link between these two objects. If a user now enters the first query, he can also follow the link and gets access to other relevant information. The difference to hypertext is, that the links are not part of the objects, not built into the objects. The link information exists in parallel to the objects. The big advantage of systems like this is that links can be added without modifying the documents. *Microcosm* is an example of such a system. It is described in [FHH90].

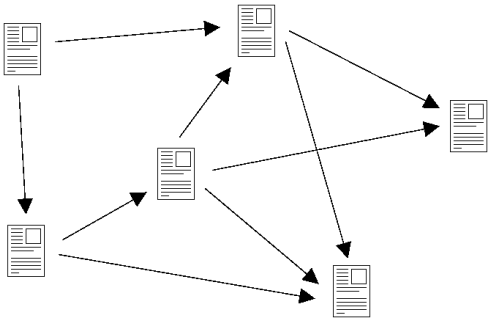


Figure 1.1: Interlinked Objects

By applying this concept of interlinked objects to a collection of documents, the information space becomes something like a web (the World Wide Web in case of the internet). For this thesis we use this abstract view of information sources. So for us the World Wide Web is just the special case of a general concept.

Another way of looking at this, is to see the links as a *non-deterministic path* through the information space. Starting at a document, we can follow a path using one link after the other to get to one document after the other. The paths might be authored or implicitly derived, deterministic or non-deterministic. Authored paths are also called *tours*. Whether they are deterministic or not, does not matter that much. The World Wide Web is an example of a non-deterministic tour. The links are authored, but there are usually multiple links for a document. An example for a deterministic tour would be a guided tour through a website. Your browsing history on the other side is an example of an unauthored path. The links between the documents are not explicitly defined. They can be implicitly derived from your browsing behavior. Unauthored paths are also called *trails*.

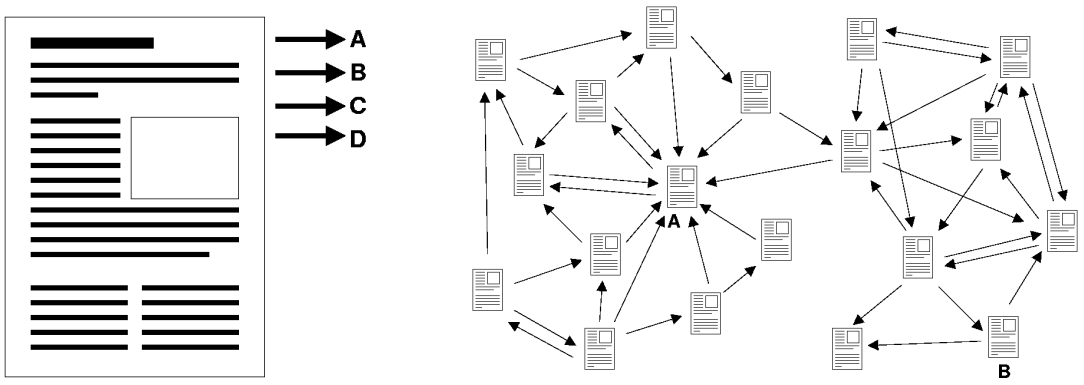
1.2 Visualization

In the last chapter, we have seen that the organization of information is a very important issue. Theoretical access to a piece of information is completely irrelevant, if there is no efficient way to find the piece of information. The same thing is also true for the presentation of information. The visualization of information has a big effect on what we perceive.

In figure 1.2(a) we present a typical view of a hypertext document. The document contains some content such as text and images and links to other documents. We do not actually see the linked documents but we can see the link and some kind of abstraction (e.g. the title of the linked document). This is also the traditional way of looking at web documents. The web document is displayed in the web browser along with the embedded links. The linked documents are abstracted by their anchor in the original document.

This view on hypertext documents is very useful, but it has its limitations. In our concept of interlinked objects, this *document view* is a very detailed one. You can see all the details of the document, but you are not able to get the big picture. The view is restricted to the current document along with its links and maybe the neighbor documents. This is a very local view where the global structure is not revealed.

In order to achieve a more global view, we have to take a step back from the painting. This is addressed by *visualizations*. Documents and links are abstracted by simpler objects (e.g. icons



(a) Traditional View (b) Visualization

Figure 1.2: Visualization of Documents and Links

and arrows) and a bigger area of the information space formed by the documents and links is displayed. Figure 1.2(b) is an example of a simple visualization.

The content of the documents and so the actual information is not visible anymore. But the visualization reveals implicit meta information about the documents. For example, we can assume, that document *A* is pretty important and relevant since a lot of links are pointing toward *A*. Document *B* can be assumed to be of less interest, since there is only one link to it.

We can also see two clusters of documents in figure 1.2(b). This can for example be explained by the fact that the documents of the left cluster are related to one topic and the documents of the right cluster are related to another topic.

To sum up, we can say that the meta information provided by such a coarse view to the documents can be very helpful in navigating the information space. But for the optimal benefit, we probably need multiple, different views to the documents. A coarse overview does not help, if we can not look at the documents themselves and a detailed view does not help that much, if we are not able to find other relevant documents.

1.3 Knowledge Transfer

We have seen in the last two sections that the information on where to find information is a very important resource. Let us assume that you want to work in a new field of knowledge, say *graph visualization*. You know that there must be plenty of documents in the World Wide Web with very useful information about this area. But before you can access these documents, you have to find them. You probably search for known web sites covering topics closely related to the desired one and use search engines. The beginning of this procedure tends to be very tedious, but once you have found a relevant document it usually gets easier.

You can imagine, that it would be very helpful, if someone had already assembled a collection of documents about the desired topic. This thesis for example is such a collection. Toward the end of it on page 65, you will find the bibliography with links to very useful papers on graph visualization. These papers again will have bibliographies with links to related papers. The papers and bibliography references are *authored links* and form a *tour* for the topic of *graph visualization*. This tour can be used by multiple users and can also be extended.

In the process of elaboration of this thesis, of course we also used this form of information collections. We found a few papers and they helped us to find other papers and so on. But we also gathered a lot of information by searching the internet using search engines and browsing around. The browser history might also be a very useful source of information for someone who wants to work into the topic of *graph visualization*. In contrast to the bibliography example, the browser history does not just contain the documents and the links (i.e. the path through the documents), but it also contains a weighting for the documents. It can be assumed, that I visited the relevant web documents several times and the irrelevant documents just once.

The other big difference of the browser history example to the bibliography example is the fact, that the browser history is private information. Even though it would be extremely helpful, it can not be accessed by other users.

1.4 Visualization of Tours and Trails

We suggest an application that combines all of the tree concepts described in the last sections: *Interlinked Objects*, *Visualization* and *Knowledge Transfer*.

The linking of objects is addressed by the *Intelligent Caching Proxy*, an application developed by Beat Signer. The current implementation supports caching of web documents and is used as HTTP proxy between the web browser and the World Wide Web. The core of the Intelligent Caching Proxy however is not restricted to web documents. It is described in the paper [SEN00] and supports caching and prefetching of any objects that can be identified by a unique id. So for example, it can also be used for the caching of queries to a database.

The link information is not assembled by analyzing the documents or database queries, but computed by analyzing the access patterns of a user. That corresponds to the browser history example. If the users accesses two documents one right after the other, than a link is added between the documents. The more often this happens, the bigger the weight of the link gets.

These *trail* links can be combined with authored link collections (i.e. *tours* that the users can define). So our model of the information space is a collection of documents (web documents, database results, ...) with several collections of links (trail and tour collections).

The *knowledge transfer* is achieved by sharing the link information between users. Both the generated trails and the authored tours can be shared within a community and all the members of this community can make use of the assembled meta information.

The objects themselves are accessed though the corresponding browser applications. It could be a web browser for web documents or a generic browser for a database. This corresponds to the *document view* mentioned earlier in this chapter. In parallel to this view, there is a *visualization* of the meta information assembled by the Intelligent Caching Proxy. The documents and links are visualized in form of a *graph*.

A person can use both views to access the data. He can click on a link in the web browser, enter a new query in the database interface or he can interact using the graph visualization. Changes in one view do also affect the other view. The manipulation of data on the other side is restricted. The original object browsers are used to manipulate the data itself, if that was intended. The graph view only allows to manipulate the meta data which means that a user can add new links or delete links from the authored collections.

In chapters 2 and 3, we present surveys on graph visualization techniques in general and existing visualization systems for hypertext. In chapters 4 to 7, we present the implementation of a generic framework for the visualization of interlinked objects and the *TrailGuide* prototype application for web documents.

2 Graph Visualization

The visualization of graphs is a very complex topic. This of course is related to the fact that graphs themselves are a very complex topic. Graphs can have many different properties ranging from very small graphs with a few vertices and edges to huge graph structures with hundreds and thousands of vertices and edges. Edges can be directed or undirected. The fanout[1] of vertices can also vary from one or two edges to a fully connected graph[2].

This variety of different graph types led to the development of even more visualizations for graphs. Some of them are quite general. Others are specialized for graph types with specific properties (e.g. trees or planar graphs[3]).

The visualization problem can be separated into two parts: The first issue is the *graphical representation* of vertices, edges or high-order structures like subtrees or clusters. The other issue is the specific *layout* for the chosen representation, i.e. the position of vertices and edges in space.

In this chapter, we would like to provide a small survey of a few graph visualizations along with their most important features and restrictions. The aim of this survey is not an in-depth discussion of the topic, but to provide a solid basis of decision for the choice of a suitable graph visualization technique for the *TrailGuide* application.

2.1 Graph Representation

Before thinking about placing vertices and edges into a plane, one has to know how to draw them, i.e. the graphical representation of vertices and edges.

2.1.1 Geometrical Abstraction

Graph objects can be abstracted by geometrical shapes. Vertices are represented by circles, ellipses, rectangles or more complex shapes. Edges are lines or arrows, straight or curved. See figure 2.1 for some examples. To increase the usability of the visualization, the geometrical shapes can be annotated with labels, colors, special icons and symbols and so on.

This is the most common graphical representation for graphs. It is widely used in the sciences, especially in mathematics and computer science.

2.1.2 Virtual Reality

The counterpart of the geometrical abstraction is virtual reality. If a vertex is an abstraction of a concrete object from the real world (e.g. a person), then the vertex could also be represented by the object itself or by a graphical abstraction of this object (e.g. a photograph of the person).

[1] The fanout is the number of edges that a vertex is connected to. It is also known as the *arity* of a vertex.
[2] A fully connected graph is a graph where every vertex is connected by an edge to every other vertex.
[3] Planar graphs can be visualized in the two-dimensional plane without intersecting edges.

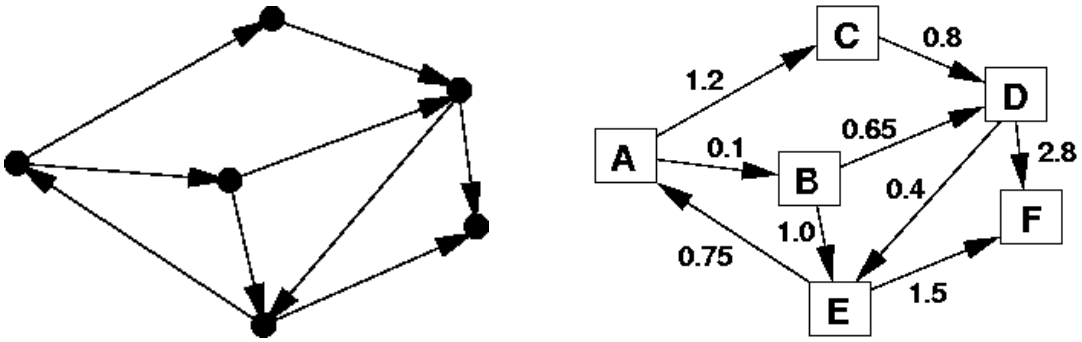


Figure 2.1: Graph Representation with Geometrical Abstraction

For many applications this is a more intuitive approach, since it corresponds much more to the real world. However, it does not work if the vertices represent abstract objects without real world counterparts. Very often, it is also difficult to find suitable graphical representations of the objects. What would be a suitable representation for a city? Should we use a picture of the whole city or just a famous building? The Eiffel Tower could be a suitable graphical representation of Paris. But what if a person does not know the Eiffel Tower or a particular city is small enough not to have any famous buildings? For these cases, the abstraction of cities with simple points and the city name is probably better.

2.1.3 Landscapes and Maps

At first glance, maps or landscapes do not differ much from normal visualizations. The difference is subtle, however. In most visualizations, the position of vertices and edges in a plane does not really have a significance. The objects are placed by a layout algorithm (see section 2.2) according to rules that try to optimize for aesthetics.

In maps and landscapes, the position of the vertices and edges in a plane play a significant role. Maps are two-dimensional visualizations on a plane or a sphere. In landscape visualizations, there is the height as an additional dimension.

The obvious example is of geographical locations (e.g. cities) where the dimensions of the visualization correspond to real world dimensions. A more abstract example is map visualization, where the vertices represent people, edges relationships between people, and the axis of the map correspond to weight and height of the people.

2.2 Graph Layout Algorithms

As mentioned in section 2.1.3, for landscape and map visualizations the position of a vertex on the plane or in space is defined by properties of the vertex itself. For all other representations, the position of vertices, edges and other graph structures (e.g. clusters) can be defined by the visualization algorithm.

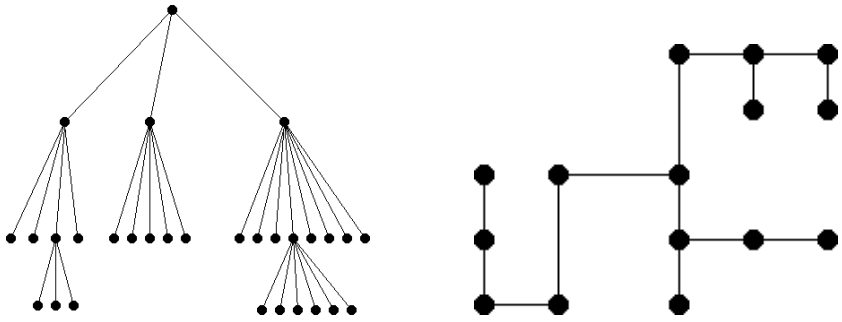
In this case, the goal for the graph layout is usually to optimize readability and aesthetic aspects. Some of these requirements can be specified in clear mathematical statements, e.g. the number of edge crossings should be minimal. Other requirements, specially aesthetical ones, are more difficult to put in words and even more difficult to put in an algorithm. Often, it is

also a question of taste and preference. Some people like the fanciness of three-dimensional visualizations, other prefer the simplicity of two-dimensional visualizations.

Graph layout algorithms are a complex topic. There are no absolute rules and no common opinion on which algorithms are the best. [BETT94] and [HMM00] provide a good overview on the most important graph layout and drawing algorithms.

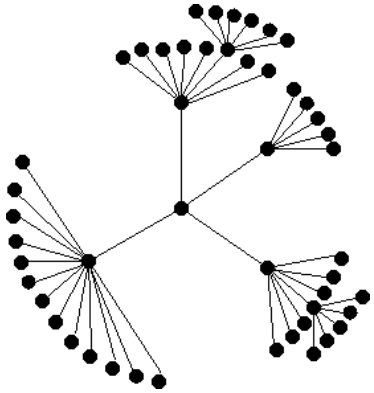
2.2.1 Two-Dimensional Trees

Two-dimensional trees are the most common visualizations for small trees. Vertices and edges of the tree are placed in a two-dimensional plane. The biggest advantage of two-dimensional trees is their simplicity and intuitivity. The representation of the graph entities is usually a very simple one: vertices are drawn as circles or boxes, edges are straight lines between the vertices. This visualization is so simple that it can even be used for manual drawing of trees. Despite this simplicity, there a several flavors of this traditional tree, which we will describe in the rest of this section.

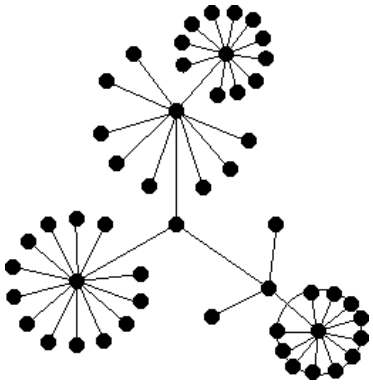


(a) Rooted Trees

(b) H-Tree



(c) Radial View



(d) Balloon View

Figure 2.2: Two-dimensional Tree Visualizations

Rooted trees are the classical tree visualization in computer science and mathematics. The tree is drawn upside down with the root on top and the leaves at the bottom. For every vertex, its children are placed one level lower. This simple rule is then applied recursively to all its subtrees. All the vertices with the same distance from the root vertex are placed at the same level and there are no edge crossings. See figure 2.2(a) for a simple example.

The *H-tree* visualization algorithm allows edges only to be drawn in horizontal or vertical direction. See figure 2.2(b) for an example.

The *radial view* of a tree is generated by putting vertices with the same distance from the root on a concentric circle around the root vertex. The radius of this circle increases with the distance of the vertices to the root. An example of a radial view is given in figure 2.2(c).

The *balloon view* visualization is quite similar to the radial view. The children are placed on a circle around their parent vertex, where the radius of the circle again decreases with the distance of the parent vertex from the root of the tree. Figure 2.2(d) is an example of a balloon view.

2.2.2 Three-Dimensional Trees

The main reason for using three-dimensional graph visualizations is the gain in space. More vertices and edges fit into the three-dimensional space than onto a plane. The disadvantage is, that the orientation and navigation is more complex. Because of that, three-dimensional visualizations are usually interactive and the user can modify his viewpoint.

All of the two-dimensional tree visualizations described in section 2.2.1 can easily be generalized for use in the three-dimensional space.

Cone Trees

Cone trees are generalizations of the balloon view tree visualization described in section 2.2.1. Every subtree is represented by a cone, where the cones of its children's subtrees are placed in.

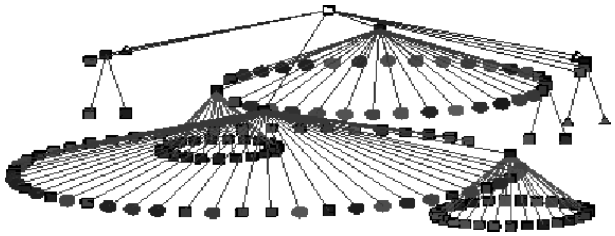


Figure 2.3: Cone Tree

Figure 2.3 (from [CK95]) is an example of such a cone tree visualization. The projection of a cone tree to a plane perpendicular to the cone's axis corresponds to the balloon view. Further information about cone trees can be found in [RMC91] and [CK95].

2.2.3 Force Optimization Algorithms

Graph layout algorithms based on *force optimization* are inspired by similar algorithms in physics and mechanics. The can be used for two- and three-dimensional visualizations. The basic idea is a system of objects in space that attract or repulse each other. The algorithm minimizes the corresponding forces on a global scale. In case of a graph, two vertices without a link between them repulse each other, two vertices with a link attract each other.

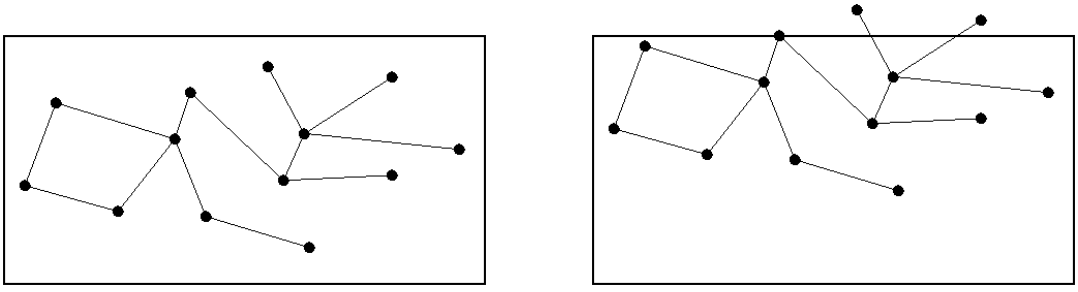
2.3 Optimizations for Large Graphs

An important issue in graph visualization is the ability to deal with large numbers of vertices and edges. Most of the visualization techniques do have the ability to merely display a large graph, but the visualizations often tend to be confusing.

To address this problem, people have developed various optimizations. With a few exceptions, they can be applied to any visualization technique. The general idea is to omit graph elements (see section 2.3.1), decrease them in size (section 2.3.2, 2.3.3 and 2.3.4) or to sum them up (section 2.3.5). Combinations of these strategies are possible.

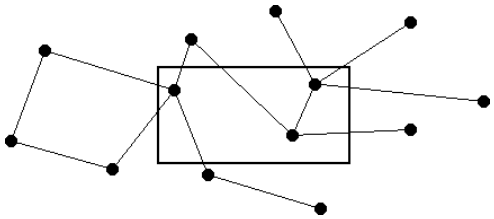
2.3.1 Zoom and Pan

A class of solutions to this problem are the so called *zoom and pan* techniques. The visualization thereby does not necessarily have to include all the vertices and edges. The view can pan to a certain area of the graph (see figure 2.4(b)) and all the vertices and edges laying outside a specified frame are not displayed. In addition to the panning, the view frame can also be resized. In the visualization this results in a zooming effect (see figure 2.4(c)).



(a) Full View

(b) Panning



(c) Zooming

Figure 2.4: Zoom and Pan

2.3.2 Focus+Context

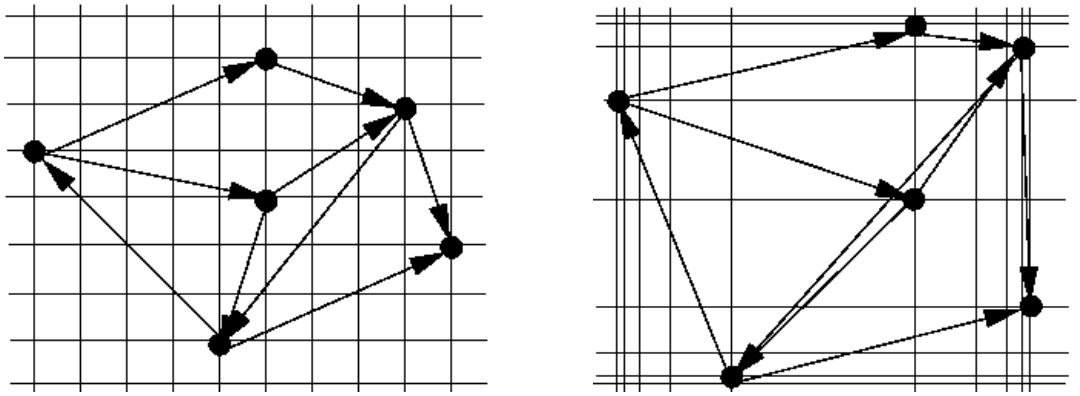
The biggest disadvantage of zoom and pan techniques is that you lose have context information when you are looking at a specific detail of a graph, which often causes disorientation and makes global browsing and navigation difficult.

Exactly this issue is addressed by *focus+context* techniques. With these visualizations, the whole graph is displayed, but the farther vertices and edges are away from the current center, the smaller they are displayed. This is a very natural approach, since it corresponds much to the way that the human eye percepts things.

Examples for focus+context visualization techniques are the *fish eye view* (section 2.3.3) and *hyperbolic trees* (section 2.3.4).

2.3.3 Fish Eye View

The fish eye view visualizes a graph as if it was seen though a very strong and non-perfect lens. Mathematically, the unit system of the plane is transformed by a distortion function in order to make a special region of the plane bigger and the rest smaller. This usually results in a nonlinear distortion as it can be seen in figure 2.5. A detailed paper about fish eye views is [SB92].



(a) Normal View (b) Fish Eye View

Figure 2.5: Fish Eye View

2.3.4 Hyperbolic Trees

Hyperbolic trees are another example of a focus+context technique. They are generated by placing a tree on a hyperbolic plane and projecting it to a flat two-dimensional plane. This results in a circle shape, where the vertices at the edge of the circle are smaller and closer together. There is more space and more level of detail toward the center of the circle. See figure 2.6 (from [LR94]) for an example of a hyperbolic tree. [LR94] and [LRP95] describe hyperbolic trees in more detail.

2.3.5 Clustering

Techniques to deal with a very large number of objects have also been developed in classical scientific information visualization. One of theses techniques is *clustering*. Objects can be summed up into clusters and in the visualization only the clusters are displayed representing all of its objects.

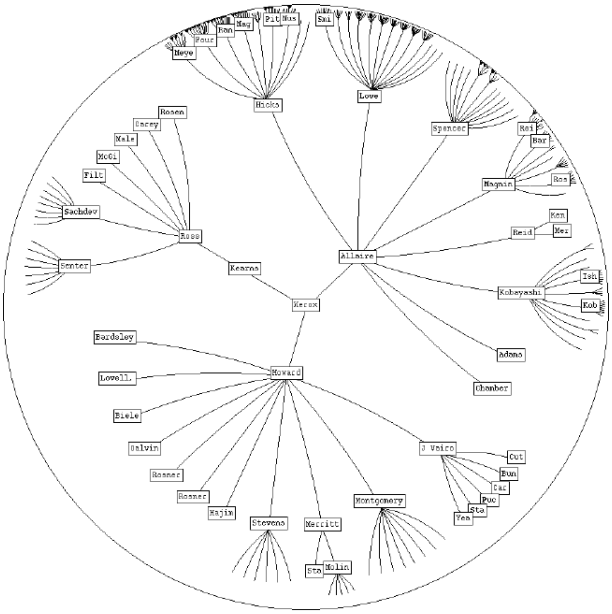


Figure 2.6: Hyperbolic Tree

This technique can also be used for the visualization of graphs, especially for trees. The clustering of vertices can either be computed with additional knowledge about the vertices or implicitly by the graph structure.

2.4 Conclusions

On a first look, three-dimensional graph visualizations seem to be the most interesting ones. They are able to visualize more data than the simple two-dimensional visualizations. The problem is, that the complexity of three dimensions is bigger and the risk is that this complexity might outweigh the advantages. Our original purpose of a visualization was to reduce the complexity of information and allow a more global view. With this requirement in mind, the simpler two-dimensional visualizations is probably the better choice.

Both *landscapes and maps* or *virtual reality* representations are unsuitable for the *TrailGuide* application. In the first case, the problem is the choice of a metric for the axis. Of course, one could use something like the size of the document or the number of hits. But these metrics are not very meaningful and the fact that two documents are close to each other in the visualization would not necessarily mean that their content was similar. Proximity functions for documents are known for information retrieval and classification problems from artificial intelligence. However, for this specific case the algorithms would probably be overkill.

One of the theses of our project is that two documents are related to each other if they are linked, i.e. if there is an edge between the vertices. Our proximity function can therefore be derived from the mathematical structure of the graph itself. Most of the graph layout algorithms take this structure into account. Thus we do not have to look for explicit proximity functions since the graph layout algorithms do the work for us.

Virtual reality visualization is also not very suitable. The problem here is the graphical abstraction of documents. There are no suitable icons that could represent a document other than the document itself. And this is exactly what we are trying to avoid with our project.

The *focus+context* optimizations are very interesting, since our intention is to visualize information spaces of infinite or at least huge size. Hyperbolic trees are a very interesting approach, but unfortunately there are patent issues for this technology. Therefore, our choice is the fish eye view. Clustering might also be of interest, specially if the clustering can be done using properties of the documents themselves. But we give clustering less priority than a solid graph layout algorithm along with a fish eye view implementation.

3 Hypertext Systems

The number of existing hypertext systems is actually pretty high. The interesting thing is that most of them are academic prototypes and have never actually been used in production environments. Even though our hypertext system was also not designed for production use, this is a disappointing observation. But questions about whether graphical visualizations of hypertext make sense at all or why they are not successful certainly are beyond the scope of this thesis.

The most important issues that we would like to address in this survey are graph visualization and the data source. The first issue has already been addressed in chapter 2. However, we are specially interested in how these theoretical principals have been adapted by various applications. The second issue is the source of the data for the graph. Where do the applications get this data from? Is the data specific to a user or can it be shared among the members of a community?

In a first section, we would like to give a quick overview of a few very interesting implementations. The aim is not to provide a complete survey, but to describe some applications with interesting and special features. In a second section, we would then like to have a more detailed look at some applications that are of special interest for our application.

3.1 Overview

In this section, we will try to give an overview of existing hypertext systems. The selection of the systems is not at all complete, but we think it is quite representative. A more complete survey can be found in [BTB+99].

A very simple 3D visualization is used by *Natto* [SM97]. The documents are visualized as labeled spheres and links between the documents are simple lines. This is nothing special yet. Interesting however is, how the documents are positioned in space. Two axis are determined by attributes of the documents. The third dimension can be manipulated by the user. Interesting and essential documents can be put at the top and non-relevant documents at the bottom.

Another application using a similar approach is *WebPath* [FS98]. The difference is that the vertical position is not determined by the user but by the system itself. The height of the documents increases with every visit, such that the most recently visited document is at the top. The importance of a document is therefore automatically determined by the number of visits. An additional, very interesting feature is a *fogging effect*. This effect is achieved by adding semi-transparent layers to the visualization. Lower documents (i.e. older) appear less obscure than the recent ones. The data for the visualization is taken from the browser history.

Narcissus [HDWB95] uses a force optimization algorithm for the graph layout (see section 2.2.3). Another interesting feature of *Narcissus* is the implicit clustering based on the calculated position of the documents.

Another application based on a force optimization algorithm is *WWW3D* [SBG+97] or its successor *HyperVIS* [BB96]. Whereas *WWW3D* limits itself to the visualization of the browser his-

tory, *HyperVIS* also includes documents that have not been visited yet. They are automatically derived from the current document by analyzing the links.

Applications based on the hyperbolic visualization described in section 2.3.4 have also been proposed. *Star Trees* is one of them as described in more detail in section 3.2.3.

The interesting feature of the *Open Text Web Index* [Bra96] is the rendering of the vertices. The vertices do not represent single web documents, but complete web sites. Attributes of these websites like the number of documents and links are displayed as shapes of a composite object. The placement of these three-dimensional objects on a plane is determined by the connectivity of the objects, i.e. two objects are close together if they have a lot of links to and from each other.

A completely different approach was implemented in *MAPA* [DK98]. It is used to display the structure of single websites. Documents are represented by small rectangular icons standing upright on a plane. The child documents are placed in a row behind their parent document. The use of this technique is pretty restricted. It basically only makes sense if the parent document represents a specific topic and the children are documents of this topic. This is why it is usually used to visualize well-organized company websites.

More in the direction of virtual reality are *Web Forager/WebBook* [CRY96]. Web documents are not abstracted, but appear on projection planes in three-dimensional space. Multiple web documents can be assembled into bookcases. The organization of the visualization as well as the collection of data is not done automatically. The application is rather a virtual, three-dimensional workspace that serves as a replacement for bookmark collections.

3.2 Case Studies

After this short overview, we would like to take a closer look at three applications that are of special interest in the context of the application we have in mind.

3.2.1 Webmap

Webmap is a rather simple hypertext visualization system. It was developed around 1994 and is described in [Doe94]. Figure 3.1 (from [Doe94]) shows a screen shot of the application.

The system interacts directly with the *Mosaic* web browser. It automatically collects data from the browser about the visited pages and can also load pages into the browser. Both the graph representation and the graph layout algorithms are very simple. Documents are represented by ellipses with numbers. Links are just simple lines. Additional information about the documents is displayed at the top and the bottom of the window.

The most important and interesting features of Webmap are the simplicity of the graph visualization and the automatic collection of data about the documents and links.

For the graph layout, the *rooted tree* algorithm described in section 2.2.1 is used. The tree is built by loading documents into the browser. If non-tree edges[1] have to be added, the layout is not affected and the edges are drawn into the existing tree layout. So the layout of the graph is defined by a simple layout of the spanning tree which is calculated incrementally.

[1] In a tree, there is exactly one path from every vertex to an other vertex. The direction of an edge does not matter. If a new edge would add an additional path, the graph would not be a tree anymore.

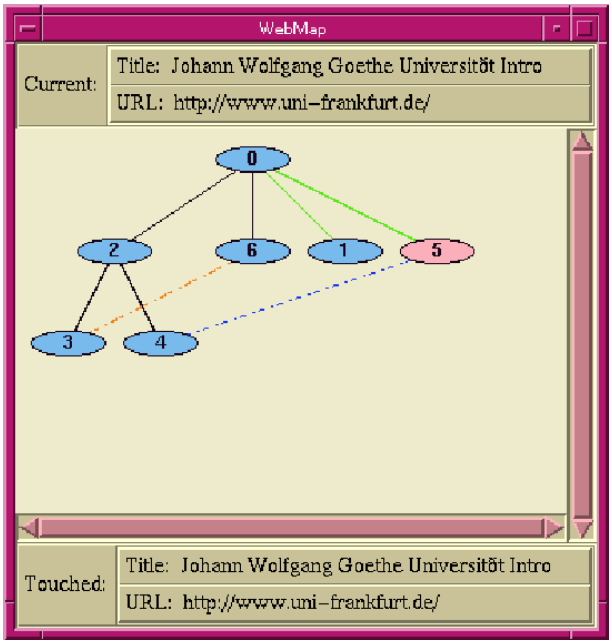


Figure 3.1: Webmap

3.2.2 WebBrain/PersonalBrain

WebBrain[2] by the company *The Brain*[3] is a web directory and search engine (see figure 3.2 for a screen shot). The special thing about this web directory is that the structure of the directory is visualized as a graph using a Java Applet.

The graph representation is again a rather simple one. The directory entries are represented by their title. The graph layout algorithm is a simple tree visualization. The root vertex, i.e. the currently selected category, is displayed in the middle of the window. At the bottom are the child categories of this vertex whereas its parent category is at the top. At the right side are vertices which are on the same level as the current root vertex, i.e. that have the same parent vertex. On the left side are alternative paths leading to the current vertex, i.e. basically the "other parents" of the root vertex. By clicking on a category in the graph window, the graph is updated with a nice animation and the links in this category appear in the lower frame of the HTML page. The graph data is retrieved from the directory and link database.

PersonalBrain is another product by the same company. The application gives the user the possibility to build a new view to his file system, the internet or other information collections. The vertices in this application can for example be documents on the local hard disk or documents on the web. The data for the graph is specified by the user rather than being generated automatically. The PersonalBrains can be shared among different users forming something like a company brain.

The interesting thing about PersonalBrain is that the user can organize his own information space. He can add elements to this information space, delete them and organize them in a structured way. On the other hand, PersonalBrain does also allow one to integrate "brains" from other people. This is especially interesting for knowledge transfer, e.g. if a user has to work in a new field. An expert in this field could just send him his brain (or a part of it).

[2]WebBrain: http://www.webbrain.com/
[3]The Brain: http://www.thebrain.com/

Figure 3.2: WebBrain

3.2.3 Star Tree

Star Tree[4] is a product of the company *Inxight*[5]. It is based on hyperbolic trees (see section 2.3.4) and is mainly used for site maps of larger websites.

Figure 3.3 shows two screen shots of the site map of the Inxight website. The hyperbolic tree can be manipulated by clicking in an empty area and dragging or by clicking on a vertex, which causes it to be moved into the center. By doing this, new vertices show up and others disappear at the border. By double-clicking on a vertex in the site map, it is moved into the center of the tree and the corresponding document is loaded in the web browser.

The most interesting feature of this application is the fact that it is able to display a large number of vertices and edges without making the visualization confusing.

3.3 Conclusions

The conclusions from the graph visualization survey are mostly confirmed. A simpler visualization might be better for the users. The title of the document, maybe along with an icon representing meta data (e.g. the size or the MIME type of the document), seems to be the representation of choice.

[4]Inxight Star Trees: http://startree.inxight.com/
[5]Inxight: http://www.inxight.com/

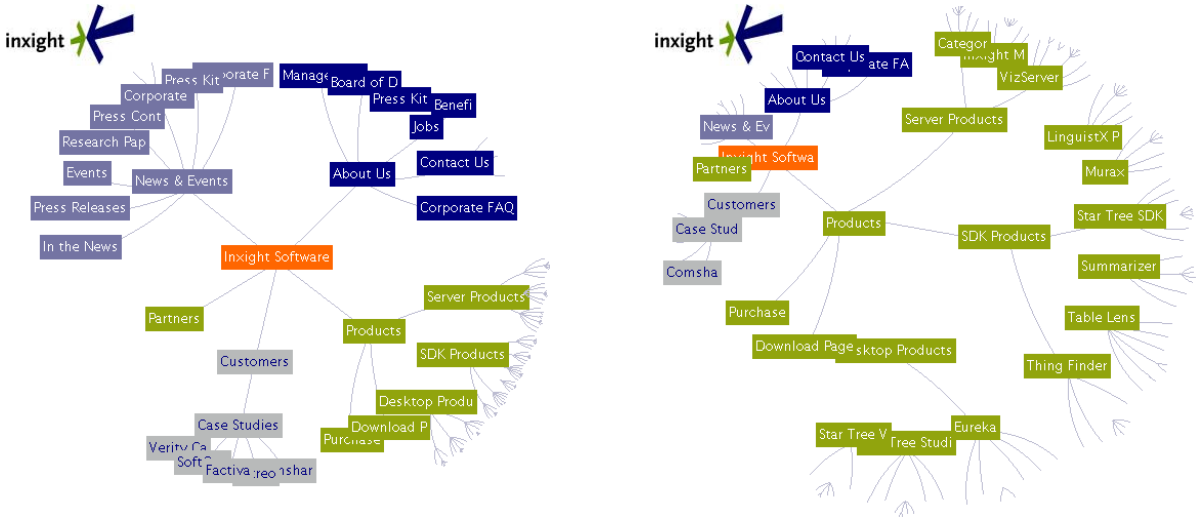


Figure 3.3: Star Tree

An observation we made is that the context of a document is a very important resource. The users literally tend to get lost without a proper visualization of the context. The *Webmap* application for example, that we like a lot for its simplicity, lacks of this context.

The persistence of the accumulated graph data seems to be another issue. Some applications we have seen assemble this data by analyzing the browser history. But this information is only available as long as the browser is running. The next time the browser is started up, the accumulation of graph data has to start from scratch.

Another interesting feature we have seen along the way is the *fogging* mechanism of the *WebPath* application. Important documents appear clearer in the visualization. Even though this is quite a simple mechanism, it helps a lot to differentiate the relevant and the irrelevant information.

4 Generic Visualization Framework

The aim of this component is to provide a generic and extensible framework for the visualization of interlinked objects. It is based on JGraph, a library for graph visualization that provides most of the functionality and which is described in section 4.1. However, we implemented some extensions to JGraph, implementing additional functionality specific to our requirements. These extensions are described in sections 4.2 and 4.3. Sections 4.4 and 4.5 describe how to integrate the framework into an application.

4.1 JGraph

JGraph[1] is a *Swing* Component for graph drawing. It was developed by Gaudenz Alder in a semester project in the Global Information Systems Group and a diploma thesis at the Electronics Laboratory, both part of the Swiss Federal Institute of Technology (ETH) in Zurich. The JGraph component is described in the paper [Ald02].

JGraph provides a nice and flexible framework for graph drawing that fits well into the Swing environment. The look and feel can easily be extended to meet almost any requirements. JGraph supports various applications from *graph editors* like *Graphpad*[2] (developed by Gaudenz Alder as well) to *graph visualization components* like our framework.

The main reason we chose JGraph is the fact that it perfectly meets our requirements. The basic representation of the vertices and edges is quite simple, but extensible. JGraph provides a lot of the basic functionality we need and the rest can easily be implemented by using and extending the JGraph API. The version of JGraph we use is *JGraph 1.0 Release Candidate 1*.

4.2 Model Classes

In order to support the special requirements of our graph visualization, we extend some of the classes from the JGraph library. Extensions have been made to the graph elements `DefaultGraphCell` and `DefaultEdge` and to the graph classes `JGraph` and `DefaultGraphModel`. All these extensions can be found in the `ch.etc.trailguide.framework` package.

4.2.1 Graph Elements

In JGraph, the default implementations for vertices and edges are `DefaultGraphCell` and `DefaultEdge`. We provide our own implementations `Document` and `Link` based on these classes. See figure 4.1 for the class diagram.

[1] JGraph: http://jgraph.com/
[2] Graphpad: http://jgraph.sourceforge.net/graphpad.html

The `Link` class has a unique identifier `id`, a collection identifier and a weight. The identifier for `Link` is generated from the identifiers of its source and target documents along with the collection identifier. The `Link` class provides a static method to construct this identifier.

The key features of the `Document` class are the identifier and label attributes that every document must have and a method `display`. This method is used to display the document that is represented by this object. The default implementation of `display` is empty, but it can be implemented for specific types of documents by subclassing `Document` and overwriting the method `display`. This is done by the TrailGuide application for web documents (see section 5.2), but it is not part of the generic visualization framework.

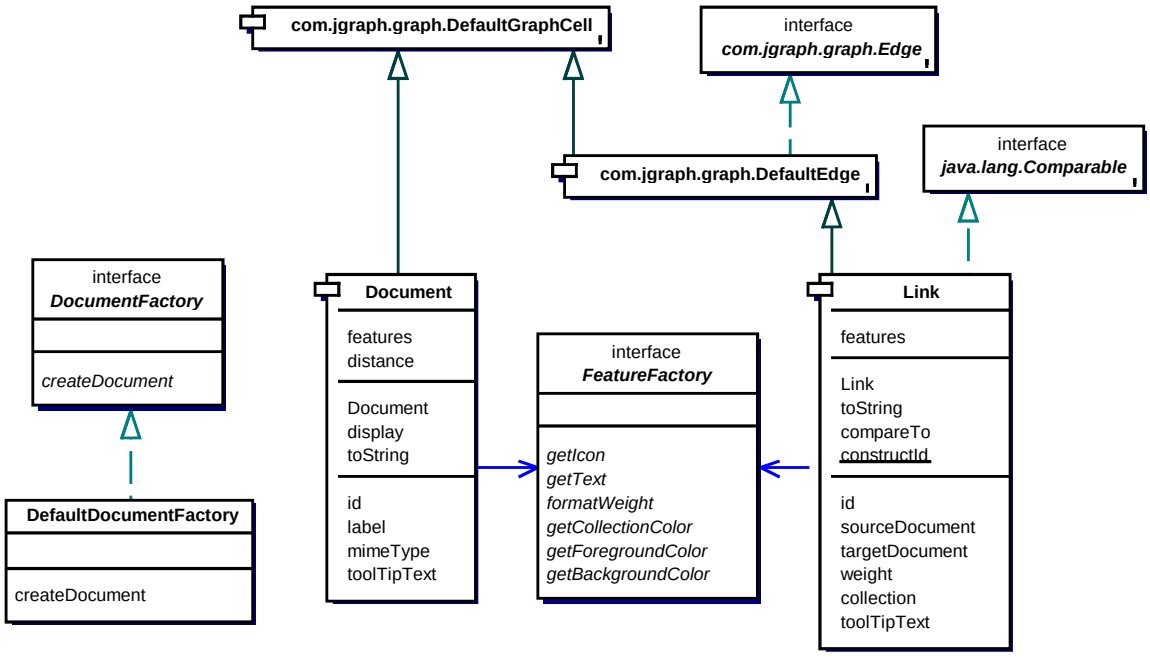


Figure 4.1: Graph Element Extensions

To create new document instances, a factory pattern is used. This allows maximum flexibility and extendibility, since new types of documents can be added to the system at any time. `DocumentFactory` defines the interface of such a document factory. A concrete implementation of this interface is `DefaultDocumentFactory`. It generates instances of `Document`, the "fallback type" for documents.

Both the `Document` and the `Link` class make use of the `FeatureFactory` interface. This interface provides labels, colors, icons and other properties or attributes relevant for the graphical presentation.

4.2.2 Graph Model

Figure 4.2 shows the class diagram for the graph model extensions. The document factories are managed by `TrailGraphModel`, an extension of the JGraph class `DefaultGraphModel`. New factories can be added to the model along with the type identifier of the documents they support. In order to create a new document object, `TrailGraphModel` looks up the corresponding factory and uses it to create the new instance. Again, this has been implemented for

web documents (see section 5.2). Attributes and properties for the graphical appearance of the elements can be retrieved via the `FeatureFactory` interface.

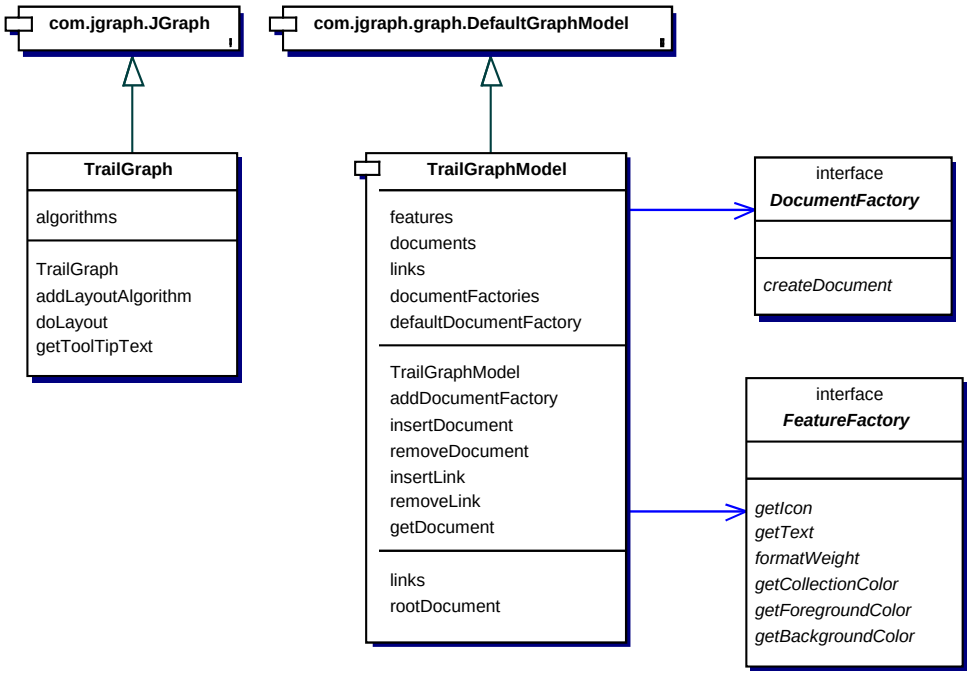


Figure 4.2: Graph Extensions

In addition to the document factoring, `TrailGraphModel` provides methods for inserting, retrieving and deleting graph elements. This is required, since in our framework, documents and links are usually accessed by their identifiers, which is not supported by JGraph.

`TrailGraph` is an extension of the `JGraph` class and provides support for layout algorithms. This is explained in the next chapter.

4.3 Layout Algorithms

Since the aim of the JGraph library is graph editing rather than automated graph visualization, it does not support any graph layout algorithms. Our implementations of these algorithms can be found in the package `ch.etc.trailguide.layout`. See figure 4.3 for the class diagram of this package.

`GraphLayoutAlgorithm` defines an generic interface for graph layout algorithms operating on an instance of `JGraph`. All the layout algorithms have to implement this interface. Currently there are two implementations available, but others can easily be added.

Layout algorithms implementing the `GraphLayoutAlgorithm` interface can be added to the `TrailGraph` object. The `TrailGraph` object overwrites the method `doLayout`, which is called by the Swing framework to layout the GUI objects. In order to lay out the graph, the registered layout algorithms are invoked in the same order as they were registered with the `TrailGraph` object.

This sequence of layout algorithms supports both *absolute* and *incremental* layout algorithms. *Absolute graph layout algorithms* set the position of all vertices of the graph, independently of

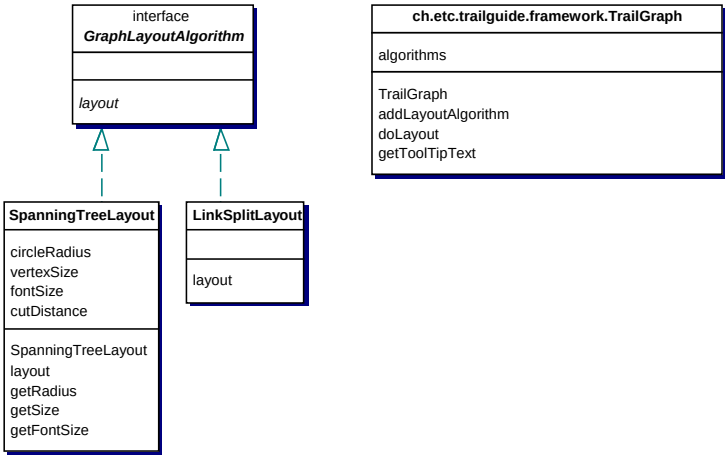


Figure 4.3: Class Diagram for Layout Algorithms

any current position, whereas *incremental graph layout algorithms* only alter the position in order to optimize the layout.

4.3.1 SpanningTreeLayout

The `SpanningTreeLayout` algorithm belongs to the first group of algorithms. The layout of the graph is determined by calculating a *spanning tree*[3] of the graph and laying out this tree using the *balloon view* algorithm described in section 2.2.1. The size of the vertices and the radius of the balloons decreases with the distance of the vertices from the root vertex. This corresponds to some kind of fish eye view. Figure 4.4 shows a sample layout produced by this algorithm. It was taken from the TrailGuide application.

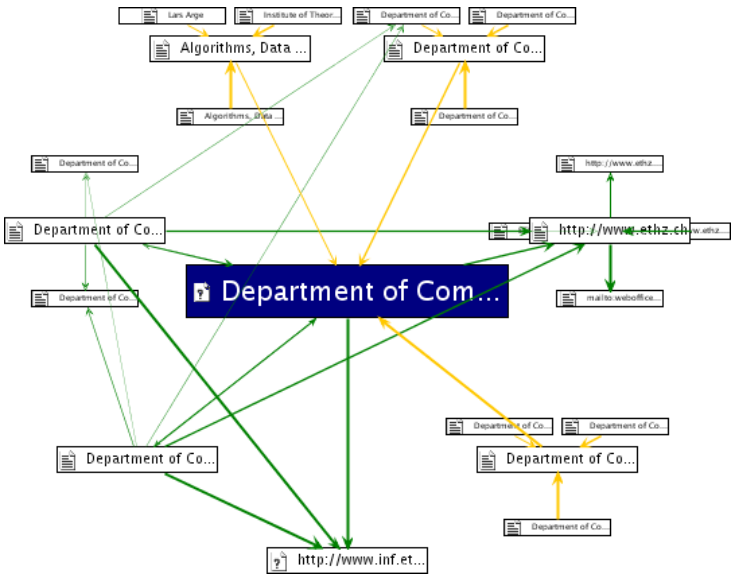


Figure 4.4: Spanning Tree Layout Example

[3]The spanning tree is a subset of the graph, containing all the vertices and some edges of the graph, such that the subset is a tree and any two vertices are connected by a sequence of edges.

The spanning tree calculation is done by traversing the graph in a *breadth-first* order and adding the edges to the tree if they do not violate the tree rule. The interesting feature of our implementation however, is that for every vertex, the edges to the neighbors are sorted by their weight in a descending order and added only if the tree rule is not violated. The reason for this sorting is that deleted edges are not respected by the balloon view layout algorithm and so might influence the visualization of the graph in a negative way. By deleting only the edges with the smallest weight, we try to keep this influence as small as possible.

4.3.2 LinkSplitLayout

A problem of the spanning tree layout algorithms is the rendering of multiple links between two documents. Since links are just straight lines, they are painted one over the other and only one link is visible in the end. This problem is not specific to the spanning tree layout algorithm, but will arise for all algorithms, where links are just straight lines.

In JGraph, edges can be straight lines, quadratic curves or even bezier curves. The `LinkSplitLayout` makes use of this feature by altering the affected links to quadratic curves. The additional point for the calculation of the quadratic curve is added on a line that is placed in the middle of the direct connection of the documents and perpendicular to it. See figure 4.5 for an example.

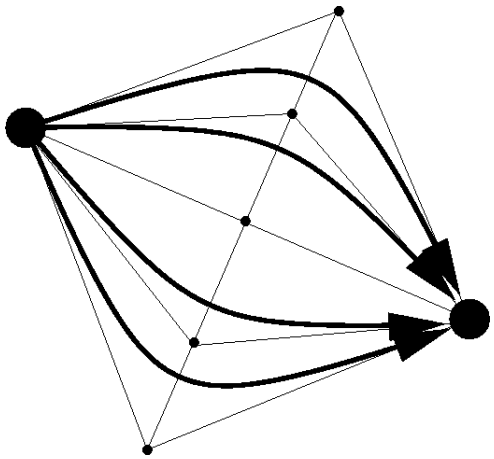


Figure 4.5: Helper Points for the Quadratic Curves

4.4 Integrating the Component

The generic visualization framework is a Swing component and can easily be integrated into any Swing-based application:

- Implement the `FeatureFactory` interface to provide labels, colors, icons and other properties for the graphical visualization.
- Create a new instance of `TrailGraphModel` and add the document factories that you desire. The fallback factory `DefaultDocumentFactory`, i.e. the document factory for unknown cases, is already installed.

- Create a new instance of `TrailGraph` using the prepared model object and configure it using the methods provided by the `JGraph` class.
- Add the desired graph layout algorithms to the graph instance.
- Add mouse listeners to the graph in order to support your specific behavior.
- Integrate the graph object into your Swing GUI.

The graph can now be manipulated by calling the methods of the `TrailGraphModel`.

4.5 Extending the Framework

There are two kinds of possible extensions to the framework. One is the addition of new document types, another the implementation of new graph layout algorithms.

4.5.1 Adding Document Types

The `Document` class provided by the framework should only be considered a base class and a fallback. It implements nothing but the minimal requirements for documents and is not able to display documents.

A new document type is added by subclassing the `Document` class. New attributes can be added if needed. The method `display` has to be overwritten in order to display the document that is represented by the object. For a web document for example, this would mean loading the web page into the web browser. Along with this new document class, the corresponding factory has to be supplied. This is done by implementing the `DocumentFactory` interface. It is not absolutely necessary to implement a factory class for every new document class. A factory can also create documents of multiple classes. As soon as the factory is registered with the `TrailGraphModel` object with the corresponding type identifier, the new documents are available to the application.

4.5.2 Adding Layout Algorithms

Another way of extending the functionality of the visualization framework is to add new graph layout algorithms. This is done by implementing the `GraphLayoutAlgorithm` interface and registering the algorithm with the `TrailGraph` object. Of course, both absolute and incremental layout algorithms can be implemented.

5 TrailGuide

5.1 Architecture

The TrailGuide architecture is based on the existing architecture of the *Intelligent Caching Proxy*. See figure 5.1 for an overview.

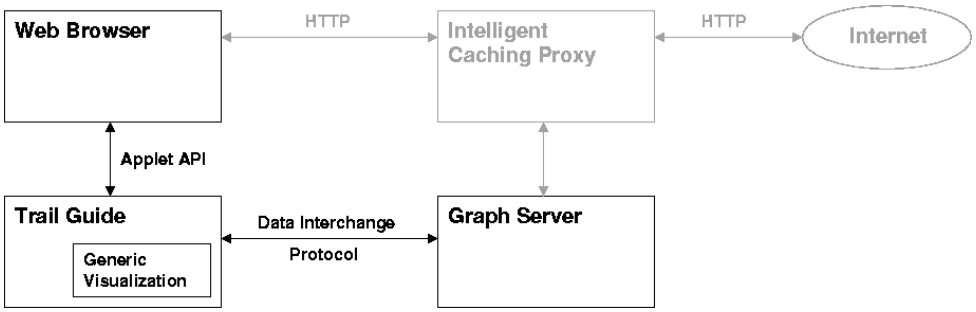


Figure 5.1: Overall Architecture

The TrailGuide is a component running on the client side. It is a plug in to the web browser, implemented as a *Java Applet*. The communication with the web browser in done using the *Applet API*. The TrailGuide is initialized with some parameters and uses the Applet API to load pages into the web browser.

To get its data, the TrailGuide communicates with a *Graph Server* using the *Data Interchange Protocol*. This protocol is described in chapter 6. The GraphServer itself was not part of this thesis, but we implemented a fake server with very limited functionality for testing and demonstrational purposes. This GraphServer is not further described in this document.

The communication protocol between the GraphServer and the Intelligent Caching Proxy is not fixed by the TrailGuide application. It is even imaginable, that the GraphServer functionality is implemented by the Intelligent Caching Proxy itself. The TrailGuide application only needs a server that implements the Data Interchange Protocol.

The inner architecture of the TrailGuide component is shown in figure 5.2. The TrailGuide applet is based on *Swing* and the *generic visualization framework*, which is described in chapter 4.

In order to support web documents, the TrailGuide provides extensions to the generic visualization framework. These extensions are described in section 5.2. Other components are the applet class itself and the classes for the *Graphical User Interface*. These classes are described in section 5.3 and 5.4.

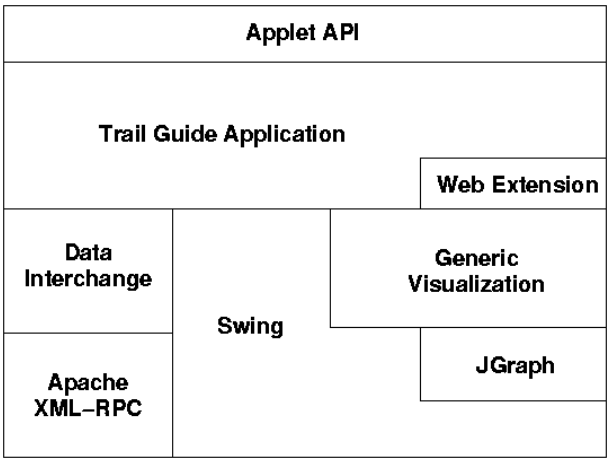


Figure 5.2: Architecture of the TrailGuide Application

5.2 Web Document Extension

The TrailGuide application provides the classes `WebDocument` and the corresponding factory `WebDocumentFactory` in the package `ch.etc.trailguide.web`. See figure 5.3 for the class diagram.

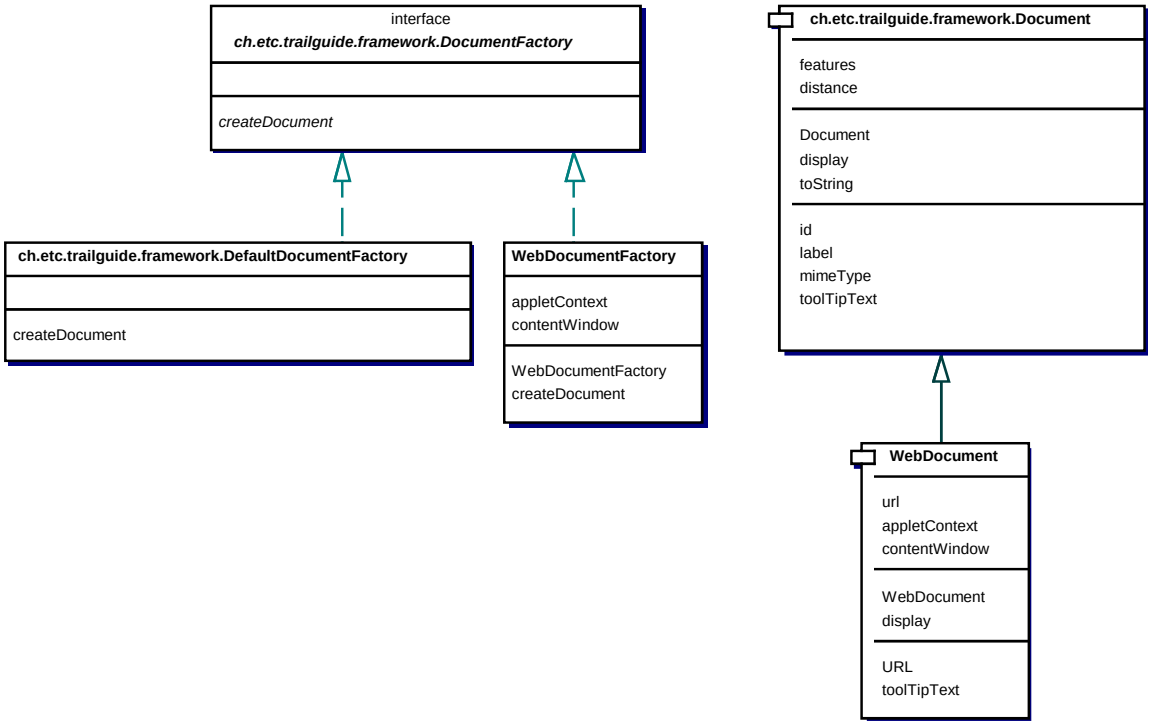


Figure 5.3: Class Diagram for Web Document Extensions

The class `WebDocument` is an extension of `Document` the base class for all documents. It provides an additional attribute `url` and overwrites the `display` method. If this `display` method is called, the web document is loaded into the browser, using its URL and the `AppletContext`. `WebDocumentFactory` is the corresponding implementation for the `Docu-`

mentFactory interface for WebDocument. It is installed at the TrailGraphModel for the web document identifier webdocument, i.e. for this identifier, the TrailGraphModel uses the WebDocumentFactory to generate documents.

5.3 Graphical User Interface

The *graphical user interface* of the TrailGuide application is based on *Swing* and the *Applet API*. Its classes can be found in the ch.etc.trailguide.gui package. See figure 5.4 for the class diagram.

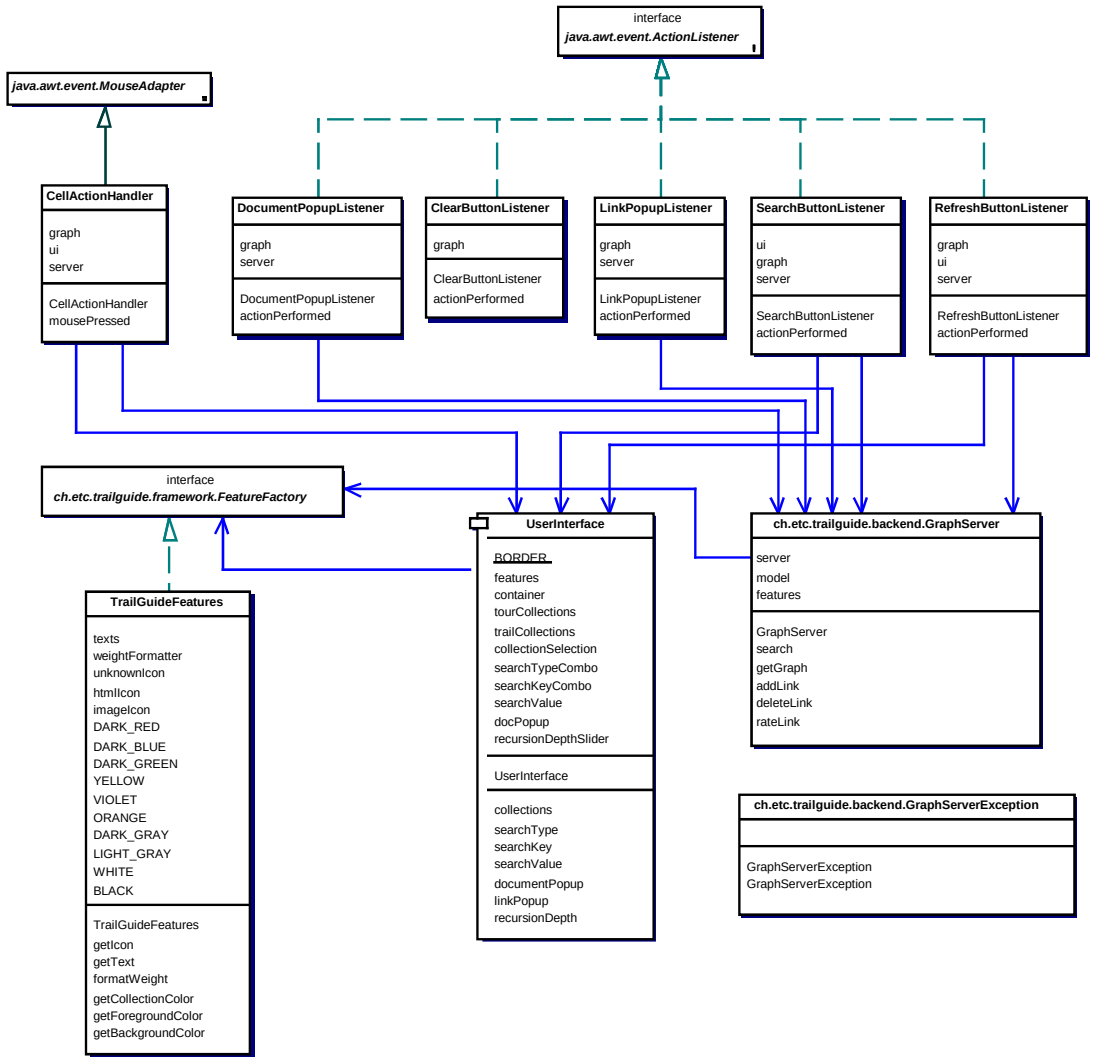


Figure 5.4: Class Diagram for the Graphical User Interface

The central class of the GUI is UserInterface. It is not derived from a Swing class, but it holds Swing components, that are rendered into a given Swing container. These components are not shown in the class diagram.

To implement the behavior of the user interface, the TrailGuide application provides various ActionListener and one MouseListener . ClearButtonListener , RefreshButtonListener and SearchButtonListener are obviously listeners for JButton instances.

`DocumentPopupListener` and `LinkPopupListener` provide popup menu functionality for documents and links. Mouse actions are supported by the `CellActionHandler` class, that listens for actions on both document and link cells.

The class `TrailGuideFeatures` implements the `FeatureFactory` interface. It is used to configure the graphical appearance of the application by providing labels, colors, icons and other properties. It is both used by the `UserInterface` class, as well as the generic visualization framework.

The user guide in chapter 7 provides additional information on the user interface.

5.4 TrailGuide Applet

As mentioned before, TrailGuide is a *Java Applet*. The class `TrailGuide` extends the class `JApplet`. It implements the `init` method and the methods `appletInfo` and `parameterInfo`. See figure 5.5 for the class diagram.

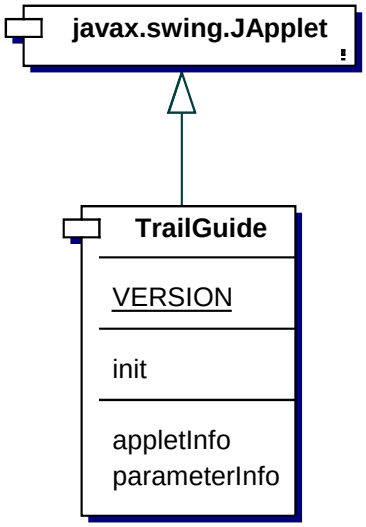


Figure 5.5: Class Diagram for the TrailGraph Applet

The applet is embedded in a HTML document. See figure 5.6 for an example. The web browser needs support for at least *Java 1.4*. The object tag is used to embed the applet. The width and height of the applet are configurable.

The applet supports two parameters `contentWindow` and `serverURL`. `contentWindow` is the name of the browser window displaying the documents. You can use `_blank` if you want a new window to be opened every time. With the parameter `serverURL`, you can set the host name, port number and path for the Graph Server. Remember that this URL has to be valid for the client machine, so `localhost` is probably not the best idea for a productive system.

```
<object classid="clsid:CAFEEFAC-0014-0000-0000-ABCDEFFEDCBA"
        width="800" height="600" align="baseline"
        codebase="http://java.sun.com/products/plugin/autodl/
                              jinstall-1_4_0-win.cab#Version=1,4,0,0">

  <!-- object parameters -->
  <param name="code" value="ch.etc.trailguide.TrailGuide"/>
  <param name="codebase" value="."/>
  <param name="archive" value="trailguide.jar"/>
  <param name="type" value="application/x-java-applet;version=1.4">

  <!-- applet parameters -->
  <param name="contentWindow" value="trailGuideContent"/>
  <param name="serverURL" value="http://localhost:80/"/>

</object>
```

Figure 5.6: HTML Code for the Embedding of the TrailGuide Applet

6 Data Interchange Protocol

The data interchange protocol is based on synchronous XML-RPC messages. XML-RPC is a protocol for remote procedure calls based on XML and HTTP. The parameters of the request and the result values are encoded in a simple XML-based language. The caller invokes a remote procedure by sending an HTTP POST request with the encoded parameters in its body. The result of the procedure call is returned encoded in XML in the body of the HTTP response. See [Win99] for details on the XML-RPC protocol.

The TrailGuide acts as an XML-RPC client and the Graph Server as the XML-RPC server, implementing the XML-RPC procedures.

6.1 Data Structures and Special Values

Some XML-RPC data structures and special values are used for multiple calls. In the following section, we will give a short overview.

6.1.1 Type Identifiers

The type identifier for documents is used to distinguish several document types (web documents, database queries, ...). In principle, type identifiers can have any value, i.e. the values are not defined by the data interchange protocol, but by the applications using the protocol. However, the values that are currently supported by the TrailGuide application are enumerated in table 6.1.

Value	Description
any	All documents
webdocument	Web documents, i.e. any document that can be identified by a URL

Table 6.1: Type Identifiers for the Data Interchange Protocol

6.1.2 Collection Identifiers

The collection identifier for links is used to classify the different sources of a link. Like type identifiers, the values for collection identifiers are not defined by the protocol, rather than the specific applications. Possible collections could be user defined links, links from the proxy cache or links from the public collection of another person. See table 6.2 for the collection identifiers currently supported by the TrailGuide application.

Value	Description
any	All collections
private	Private collection of the user
public	Public collection of the user
document	Links extracted from the document itself (e.g. HTML links)
google	Links to a document from the google search engine
cache	Links from the proxy cache
friends	Common collection of friend users
group	Common collection of group users

Table 6.2: Collection identifiers for the Data Interchange Protocol

6.1.3 Document Data Structure

Table 6.3 shows the attributes of the document data structure. The mandatory fields must be supplied in order to implement the protocol. Optional fields can be added depending on the specific types of documents or links and on the specific implementation of clients and servers. An optional field for documents is the MIME type. The applications should be tolerant regarding missing optional links.

Attribute	XML-RPC Type	Mandatory	Description
id	string	*	identifier of the document
label	string	*	label for the document
type	string	*	type identifier
url	string		url of the document (for web documents)
mimeType	string		the mime type of the document

Table 6.3: Attributes of the Document Data Structure

The document identifier `id` is supplied by the server application. The client application (e.g. TrailGuide) does not have to know anything about the internal structure of the `id` string.

6.1.4 Link Data Structure

The attributes of the link data structure are enumerated and described in table 6.4.

Attribute	XML-RPC Type	Mandatory	Description
source	string	*	identifier of the source document
target	string	*	identifier of the target document
weight	double	*	weight for the link
collection	string	*	collection identifier for the link

Table 6.4: Attributes of the Link Data Structure

The `source` and `target` strings are document identifiers. The weight of a link is a positive real number. It does not necessarily have to be a probability, i.e. $0 <= weight <= 1$.

6.2 Methods

The following methods are supported by the data interchange protocol:

6.2.1 getGraph

The method `getGraph` is used to get the graph for a specific root document. The parameters of the procedure are presented in table 6.5.

Parameter	XML-RPC Type	Description
id	string	identifier of the root document
depth	integer	maximum distance of documents from the root node
collections	array of strings	the identifiers of the collections

Table 6.5: Parameters of the `getGraph` Method

The result of the getGraph procedure is an array with two elements. The first one is an array of documents, the second one is an array of links. See section 6.1.3 and 6.1.4 for a definition of the document and link data structures. See appendix C.1 for a sample request and response.

6.2.2 search

The `search` method is used to search for documents. The parameters of the method are outlined in table 6.6.

Parameter	XML-RPC Type	Description
type	string	a type identifier
key	string	the key name of a property
value	string	the search string

Table 6.6: Parameters of the `search` Method

The response is an array of documents as defined in section 6.1.3. For a sample request and the corresponding response see appendix C.2.

6.2.3 addLink

The semantics of this method depends on the collection that contains the link. Links can only be added to user-defined link collections. The addition of a link to an automatically generated collection is always ignored. In table 6.7 we show the method's parameters.

The `addLink` method returns a boolean that tells the client whether the link has been added successfully or not. A sample method call can be found in appendix C.3.

6.2.4 deleteLink

The semantics of this method depends on the collection that contains the link. For user defined link collections, the link might be removed whereas for automatically generated collections, the link is only hidden. The parameters are shown in table 6.8.

Parameter	XML-RPC Type	Description
source	string	the identifier of the source document
target	string	the identifier of the target document
weight	double	the weight for the link
collection	string	the collection identifier

Table 6.7: Parameters of the `addLink` Method

Parameter	XML-RPC Type	Description
source	string	the identifier of the source document
target	string	the identifier of the target document
collection	string	the collection identifier

Table 6.8: Parameters of the `deleteLink` Method

The `deleteLink` method returns a boolean telling the client whether a link has really removed or not. See appendix C.4 for an example of a method call.

6.2.5 rateLink

The user can give feedback to the system about the generated links. One possibility is to simply delete the link. The rating of links however is a more detailed one. Both generated and authored links can be rated. The parameters for the `rateLink` method are shown in table 6.9. The return value of the `rateLink` method is the double value which is new weight for the rated link. See appendix C.5 for a sample request and response.

Parameter	XML-RPC Type	Description
source	string	the identifier of the source document
target	string	the identifier of the target document
collection	string	the collection identifier
rating	int	the rating

Table 6.9: Parameters of the `rateLink` Method

`rating` is an integer parameter. The semantics can be defined by the application, but currently the values in table 6.10 are supported.

Value	Semantics
0	not very important
1	quite important
2	important
3	very important
4	extremely important

Table 6.10: Values for the Rating Parameter of the `rateLink` Method

7 User Guide

In the last chapters we described the architecture, design and implementation of the TrailGuide application. In this chapter we want to present the graphical user interface of the application. We describe the use of the different GUI components and the features of the application. The compilation and installation of the system is described in appendix B.

7.1 Graphical User Interface

The TrailGuide is an *Java Applet* that runs within a web page. Figure 7.1 shows the application window with a typical graph.

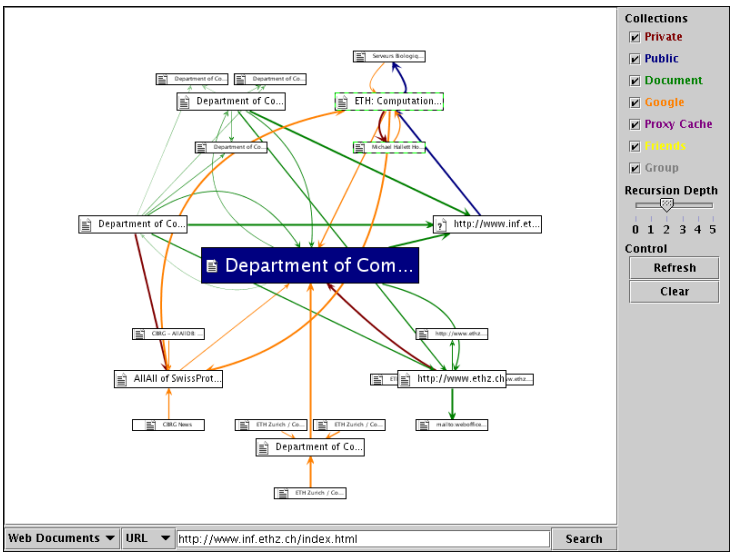
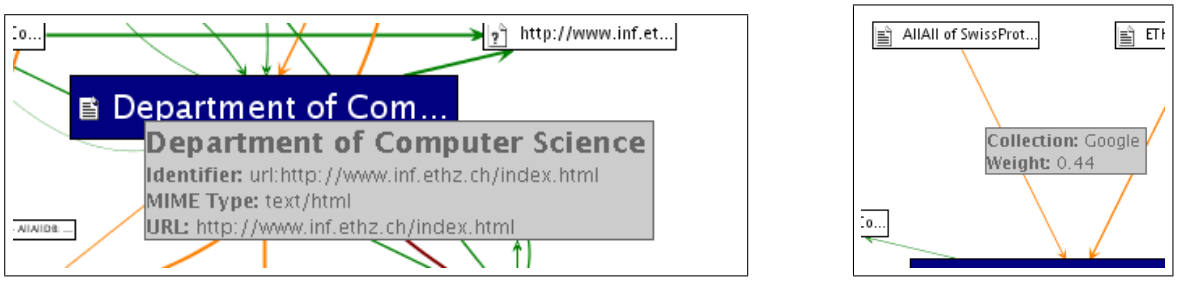


Figure 7.1: TrailGuide Window

On the right side is a *configuration panel* whereas at the bottom a *search panel* can be found. The rest of the window is reserved for the concrete graph visualization.

The visualization only shows the title and the MIME type for documents, and the collection and weight for links. A tooltip window providing additional information about a specific object pops up on mouse over. See figure 7.2 for two examples.

The graph can be configured using the controls in the configuration panel on the right side. At the top, you can select the link collections you would like to use. With the recursion slider, the level of recursion used to build the graph can be set. Any changes of the configuration settings do not immediately affect the graph, but are considered on the next update operation.



(a) Document Tooltip

(b) Link Tooltip

Figure 7.2: Tooltips

7.2 Functionality

7.2.1 Search for a Document

To search for a document, the *search panel* at the bottom of the page can be used. The settings of the *configuration panel* have a direct influence on the search process. You can select the *document type*, the *attribute* and a *value* for the selected attribute. By clicking on the search button, the first document that matches is displayed along with the corresponding graph. If more than one document is found, the right one can be chosen in a popup window.

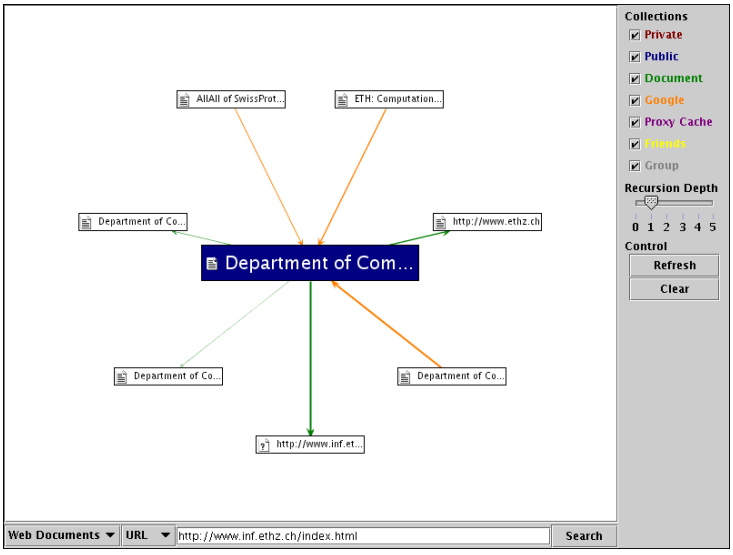


Figure 7.3: Search for a Document

Figure 7.3 shows the result of a search. The type of document was *web document* with the attribute *URL* and value *http://www.inf.ethz.ch/index.html*. The activated link collections are *Document* and *Google*, the recursion depth is *1*. The matching document is highlighted.

7.2.2 Browsing

The most characteristic functionality is *browsing* the graph. By double-clicking on a document of the graph, the document is loaded into the browser and the graph is updated.

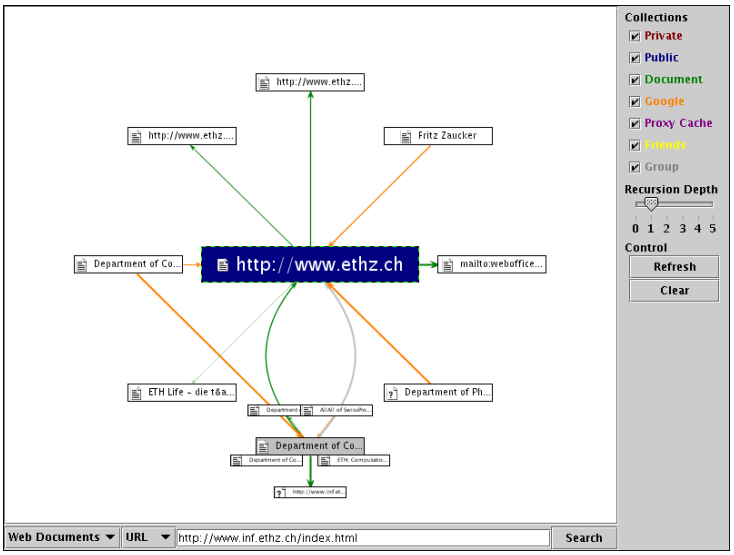


Figure 7.4: Double-Click on a Graph Document

Figure 7.4 shows the application after such a browse operation. The user clicked on the document *http://www.ethz.ch*. This document is now selected and the graph for this document was retrieved from the server. The documents from the last view are still available in the graph, but they appear smaller, since they are farther away from the current document. All the documents that have already been visited are highlighted with a light gray background. What you can not see in the screen shot is that the web document also has been loaded into the web browser.

7.2.3 Add Link

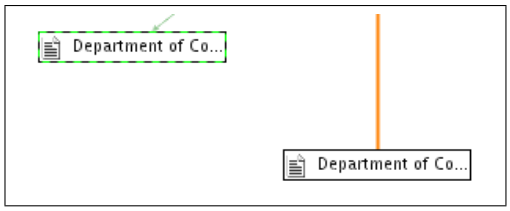
To some collections, links can also be added. Currently only the collections *Private* and *Public* allow for the addition of links.

To add a link between two documents, both documents have to be visible on the screen. First, the source document has to be selected. This is done by clicking once on the document in the graph (see figure 7.5(a)). Next, click on the target document with the right mouse button, which causes a popup menu to appear. Select *Add Link* and the collection you would like to add the link to (see 7.5(b)). The new link is now added to the visualization and also on the server. This can cause a re-layout of the graph.

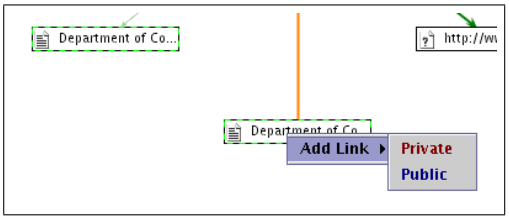
If the link that you would like to add already exists in another collection and is displayed on the screen, you can also just right-click on this link and add it with the same popup menu. After the operation, both links will appear in the visualization and on the server.

7.2.4 Move Link

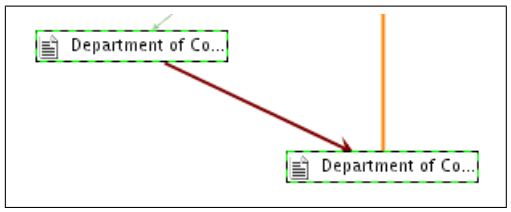
If you do not want to keep the old link, you can also just move it to the new collection. The procedure is the same, but you select *Move Link* instead of *Add Link*. See figure 7.6 for a screen shot.



(a) Select the Source Document

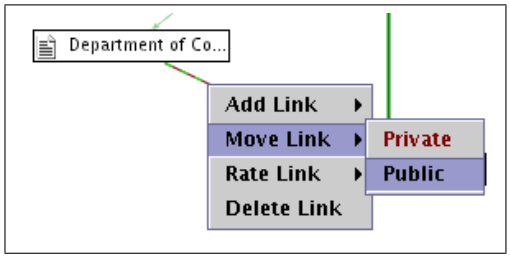


(b) Select Target Document and Collection

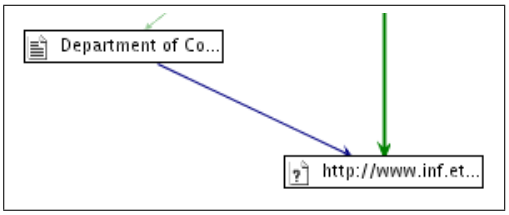


(c) Link was added

Figure 7.5: Add a Link



(a) Select new Collection



(b) Link was moved

Figure 7.6: Move Link into another Collection

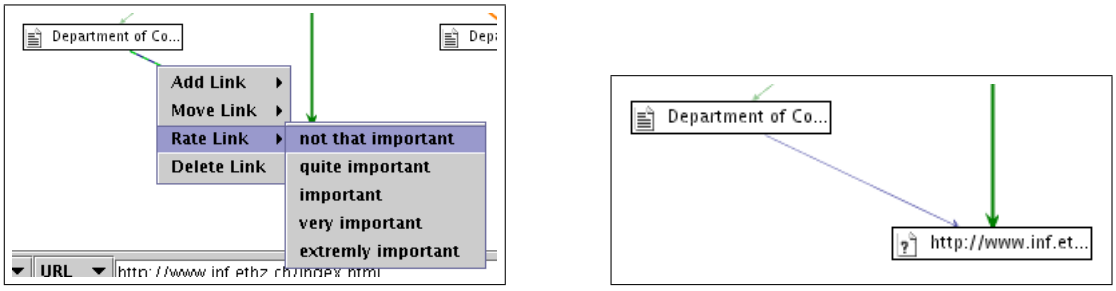
7.2.5 Rate Link

Rate Link can be used to give the system feedback about generated links or to change the weight of authored links. See the screen shots in figure 7.7.

Select the link with a right-click and choose the rating from the popup menu. The links is now updated. The width of the link corresponds to the new weight, which is determined by the GraphServer. For authored links, it might exactly reflect your rating. For generated links however, your rating might just be one of many and the effect on the link might be small.

7.2.6 Delete Link

Analogous to *Add Link* and *Move Link*, there is also a *Delete Link* menu item. Deletion of links is possible for all the collections; the semantics however may vary. For the collections *Private* and *Public*, deletion means that the link is removed from the collection. For the other collections,



(a) Select the Rating

(b) Link was rated

Figure 7.7: Rating Links

the link is not actually deleted, since this is either not possible (e.g. the *Google* collection, which is generated using the google search engine) or would affect other users (e.g. the *Friends* collection). For these collections, the link is only marked as deleted and is no longer displayed for the current user.

8 Summary and Outlook

In chapter 1 we gave a quick introduction to the topic. The aim was to present the field of research and to give a motivation for the importance of concepts like *interlinked objects*, *visualization* and *knowledge transfer*.

In the following chapters 2 and 3 we presented a survey on graph visualization techniques and on existing proposals and implementations of hypertext systems. The implications have been vary valuable for the design and implementation of our own implementation. An important conclusion we could draw, specially from the graph visualization survey, can be summarized as: Less is more! Fancy three-dimensional visualizations look really nice, but you run the risk that they confuse the user more than they help him. A rather simple, two-dimensional visualization is usually more effective.

The first part of our implementation work was dedicated to a generic visualization component for interlinked objects. This component was described in chapter 4. The aim was to design a generic framework that could be extended to support any type of documents. The most important issue here, next to the design of the framework, was the visualization of the documents and links, specially the graph layout algorithms.

The proof of concept for the extendibility and usability of the visualization component was provided with TrailGuide, a prototype application for web documents which was described in chapters 5 to 7. This documentation includes the extension to the generic visualization component for the support of web documents, the architecture and design of the TrailGuide applet and the data interchange protocol used to communicate with the GraphServer backend that provides and manages the graph data. As well as the generic visualization component, the design of both the TrailGuide application and the data interchange protocol is extensible to support other types of documents.

With the implementation of the TrailGuide prototype we have been able demonstrate the benefits of our proposed system. Even though the visualization is still improvable, we were able to see its positive effects, specially on the navigation within the information space of the internet. The effect on other application areas (e.g. object-oriented databases), specially such that do not provide links as integrated feature, is assumed to be even bigger. And although is has not really been used yet, the feature of sharing link collections in a community promises to be very interesting.

Future Works

Even though the TrailGuide application has always been intended to be a prototype only, some parts of it are probably worth investing more time. With our spanning tree algorithm and link splitting algorithm we believe to have found a pretty good graph layout. Additional research and development in this area however, would be very useful. This includes the algorithms themselves, but also the way they are applied. In the current implementation, the layout algorithms are run if necessary and then the result is displayed. This often results in a completely

new layout, i.e. the objects on the screen are moved to a completely different place. This is unavoidable, but very confusing for the users. An animated motion of the objects would probably help a lot.

Fogging is a very nice feature we discovered in our survey on hypertext systems. Since Swing support alpha values for colors, it should be pretty easy to integrate this feature into our application. The farther away a document is from the current root node, the smaller the alpha value would be.

The configuration of the TrailGuide applet is currently done with constants in the source code, a property file for texts and applet parameters. The text property file makes it very easy, to port the applet to another language. The constants in the source code however impede a lot of flexibility. The supported link collections for example do not depend on the applet, but on the implementation of the GraphServer. Thus, another possibility would be to implement an additional XML-RPC call `config` to the GraphServer, which would basically provide configuration data like this.

Another unsolved problem is the synchronization of the browser and the visualization view. If the user clicks on a document in the TrailGraph applet, the web browser is updated and displays the same page. However, if the user clicks on a link in the web browser or enters a new URL, this change is not propagated to the graph visualization. The reason for this missing feature are security restrictions in the Applet API. There is no possibility for the applet to get the URL of the current document. A work-around however, would be to get this information from the Intelligent Caching Proxy. If a new document is loaded by the web browser, the Intelligent Caching Proxy (or the GraphServer) could make a callback to the applet. The big advantage of such a mechanism is the independence from the specific browser. It would not matter if the browser was a web browser or maybe a generic database browser.

In the development of the TrailGuide application, at lot of effort was put into a clean and extensible design. The integration of the object-oriented database management system OMS would be a very remunerative goal. A lot of components for such a project have already been developed or are planned at least. The Intelligent Caching Proxy has always been intended to support caching and prefetching of generic objects, not just web documents. This would also include query responses to an OMS database. Generic browsers for such a database do also exist.

A Assignment

ETH Eidgenössische Technische Hochschule Zürich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

Institute for Information Systems:
Prof. M. C. Norrie

Visualisation of Tours and Trails

Corsin Decurtins

The Intelligent Caching Proxy (ICP) is a proxy server which monitors regular web users and tries to detect specific access patterns. Currently this information is used to improve web response times by performing active prefetching of data. In a new project, the same statistics are used to build dynamic trails (i.e. to dynamically build 'virtual links' between different pages. In the future, this linking mechanism will be extended to allow not only links between web pages, but also any physical object which has a unique identifier.

The goal of this diploma project is to develop a visualisation component which enables a user to browse his own virtual information space. The visualisation will include authored links between objects (tours) and also dynamically built links (trails). The user should be able to manually add new links, delete existing links and give feedback to the systems to improve the performance of dynamically generated links. The visualisation framework will be used to build a first prototype which will be a 'trail guide' add-on for existing web browsers (a component showing information about links between web pages based on monitored user behaviour).

The main tasks of this diploma thesis are as follows:

- Investigate existing Hypertext Systems (especially systems which dynamically generate trails) and existing visualisation technologies (regular graphs, cone trees, hyperbolic trees, etc.).
- Define a format and protocol (e.g. SOAP) for the interchange of data between the link generator and the visualisation component.
- Design of a framework for visualisation of any kind of interlinked objects (might be inspired by the former 'jgraph' project).
- Implementation of a 'trail guide' as an add-on component for existing web browsers giving visual feedback about dynamically built links between different web pages.

Optionally the existing link generator framework may be extended with the functionality to share linking information. The idea is that different users can share information about their virtual worlds (using peer-to-peer technologies such as the Gnutella protocol). This will allow users who have less or no knowledge about a specific object to build links based on information of parts of the community.

The project report should give an overview of existing trail building systems and explain available visualisation techniques. It should describe in detail the generic visualisation component and how it can be used to build specific applications. Finally, the architecture of the trail guide application should be explained (e.g. communication with the link generator) and there should be a user guide for this application as well.

Start Date: Monday 5 November 2001
Environment: Java
Supervision: Beat Signer, IFW D46.2

B Installation Guide

This appendix describes how to compile and install the TrailGuide application along with our test server *IGS* (Intelligent Graph Server).

B.1 Compiling

The compiling process is driven by the *GNU make* tool. The main `Makefile` is in the directory `$(PROJECT_ROOT)/src`. It uses separate Makefiles for the TrailGuide application, the documentation and currently also the test server.

The supported targets are:

`all`	This is the default target. It compiles all the application files. The documentation is not affected.
`check`	This target executes some checks on the sources and/or documentation, e.g. unit tests (for applications) or spell checking (for documentation).
`clean`	This cleans the source directory from temporary files.
`distclean`	This cleans *all* the generated files.
`doc`	This target is used to generate the documentation, both the report and the API documentation for the source files.
`install`	Install the application (see section B.2).

All the generated files are placed in a separate tree, rooted at `$(PROJECT_HOME)/generated`.

B.2 Installation

The installation is triggered by the `install` target of the `Makefile`. The `make` command takes a parameter `DESTDIR` that specifies where to install the application. The command might look something like this:

```
make install DESTDIR=/opt/trailguide
```

In the `$(DESTDIR)`, the subdirectories `doc`, `htdocs` and `lib` are generated. `doc` holds the documentation, i.e. several versions of this file and the HTML version of the API documentation generated by JavaDoc. `lib` holds the generated jar files for the backend applications. The directory `htdocs` can be used as root directory for a web server. It contains HTML documents for the TrailGuide applet, the `jar` file for the applet, other files and links to the documentation.

B.3 Running the Application

The TrailGuide applet does not have to be started explicitly. It can just be accessed via the web server. The test backend IGS is started up with the command

```
java -classpath igs.jar:xmlrpc.jar ch.etc.igs.IGS PORT
```

where `PORT` is the port number. Make sure that `PORT` matches the port number specified in the HTML document holding the applet (`trailguide.html`).

C Data Interchange Protocol Examples

C.1 getGraph

```
POST / HTTP/1.1
Content-Type: text/xml

<?xml version="1.0" encoding="ISO-8859-1"?>
<methodCall>
  <methodName>getGraph</methodName>
  <params>
    <param>
      <value>url:http://www.globis.ethz.ch/</value>
    </param>
    <param>
      <value><int>2</int></value>
    </param>
    <param>
      <value>
        <array>
          <data>
            <value>private</value>
            <value>public</value>
            <value>cache</value>
            <value>group</value>
          </data>
        </array>
      </value>
    </param>
  </params>
</methodCall>

HTTP/1.1 200 OK
Content-Type: text/xml

<?xml version="1.0" encoding="ISO-8859-1"?>
<methodResponse>
  <params>
    <param>
      <value>
        <array>
          <data>
            <value>
```

```
      <array>
        <data>
          <value>
            <struct>
              <member>
                <name>label</name>
                <value>ETH Zurich</value>
              </member>
              <member>
                <name>type</name>
                <value>webdocument</value>
              </member>
              <member>
                <name>id</name>
                <value>url:http://www.ethz.ch/</value>
              </member>
              <member>
                <name>mimeType</name>
                <value>text/html</value>
              </member>
            </struct>
          </value>
          <value>
            <struct>
              <member>
                <name>label</name>
                <value>Global Information Systems</value>
              </member>
              <member>
                <name>type</name>
                <value>default</value>
              </member>
              <member>
                <name>id</name>
                <value>url:http://www.globis.ethz.ch/</value>
              </member>
            </struct>
          </value>
        </data>
      </array>
    </value>
    <value>
      <array>
        <data>
          <value>
            <struct>
              <member>
                <name>collection</name>
                <value>default</value>
              </member>
```

```
                    <member>
                      <name>weight</name>
                      <value><double>0.5</double></value>
                    </member>
                    <member>
                      <name>source</name>
                      <value>url:http://www.ethz.ch/</value>
                    </member>
                    <member>
                      <name>target</name>
                      <value>url:http://www.inf.ethz.ch/</value>
                    </member>
                  </struct>
                </value>
              </data>
            </array>
          </value>
        </data>
      </array>
    </value>
  </param>
 </params>
</methodResponse>
```

C.2 search

```
POST / HTTP/1.1
Content-Type: text/xml

<?xml version="1.0" encoding="ISO-8859-1"?>
<methodCall>
  <methodName>search</methodName>
  <params>
    <param>
      <value>webdocument</value>
    </param>
    <param>
      <value>url</value>
    </param>
    <param>
      <value>http://www.inf.ethz.ch/index.html</value>
    </param>
  </params>
</methodCall>


HTTP/1.1 200 OK
Content-Type: text/xml

<?xml version="1.0" encoding="ISO-8859-1"?>
```

```
<methodResponse>
  <params>
    <param>
      <value>
        <array>
          <data>
            <value>
              <struct>
                <member>
                  <name>url</name>
                  <value>http://www.inf.ethz.ch/index.html</value>
                </member>
                <member>
                  <name>label</name>
                  <value>Department of Computer Science</value>
                </member>
                <member>
                  <name>type</name>
                  <value>webdocument</value>
                </member>
                <member>
                  <name>id</name>
                  <value>url:http://www.inf.ethz.ch/index.html</value>
                </member>
              </struct>
            </value>
          </data>
        </array>
      </value>
    </param>
  </params>
</methodResponse>
```

C.3 addLink

```
POST / HTTP/1.1
Content-Type: text/xml

<?xml version="1.0" encoding="ISO-8859-1"?>
<methodCall>
  <methodName>addLink</methodName>
  <params>
    <param>
      <value>url:http://www.inf.ethz.ch/index.html</value>
    </param>
    <param>
      <value>url:http://www.ethz.ch</value>
    </param>
    <param>
```

```
      <value><double>1.0</double></value>
    </param>
    <param>
      <value>public</value>
    </param>
  </params>
</methodCall>

HTTP/1.1 200 OK
Content-Type: text/xml

<?xml version="1.0" encoding="ISO-8859-1"?>
<methodResponse>
  <params>
    <param>
      <value><boolean>0</boolean></value>
    </param>
  </params>
</methodResponse>
```

C.4 deleteLink

```
POST / HTTP/1.1
Content-Type: text/xml

<?xml version="1.0" encoding="ISO-8859-1"?>
<methodCall>
  <methodName>deleteLink</methodName>
  <params>
    <param>
      <value>url:http://www.inf.ethz.ch/index.html</value>
    </param>
    <param>
      <value>url:http://www.ethz.ch/</value>
    </param>
    <param>
      <value>document</value>
    </param>
  </params>
</methodCall>

HTTP/1.1 200 OK
Content-Type: text/xml

<?xml version="1.0" encoding="ISO-8859-1"?>
<methodResponse>
  <params>
    <param>
      <value><boolean>1</boolean></value>
```

```
    </param>
  </params>
</methodResponse>
```

C.5 rateLink

```
POST / HTTP/1.1
Content-Type: text/xml

<?xml version="1.0" encoding="ISO-8859-1"?>
<methodCall>
  <methodName>rateLink</methodName>
  <params>
    <param>
      <value>url:http://www.inf.ethz.ch/index.html</value>
    </param>
    <param>
      <value>url:http://www.ethz.ch</value>
    </param>
    <param>
      <value>cache</value>
    </param>
    <param>
      <value><int>3</int></value>
    </param>
  </params>
</methodCall>


HTTP/1.1 200 OK
Content-Type: text/xml

<?xml version="1.0" encoding="ISO-8859-1"?>
<methodResponse>
  <params>
    <param>
      <value><double>0.8</double>
    </value>
  </param>
</params>
```

D API Documentation

This is an abridged version of the JavaDoc API documentation for the TrailGuide application and the generic visualization component. For more details, use the HTML version of the API documentation.

D.1 ch.etc.trailguide

This is the root package for the trail guide application. It contains the application classes only. All the other classes of the application can be found in the subpackages.

ch.etc.trailguide.TrailGuide

```
java.lang.Object
  java.awt.Component
    java.awt.Container
      java.awt.Panel
        java.applet.Applet
          javax.swing.JApplet
```

`public TrailGuide extends JApplet`

Trail Guide

Fields

`public static final String`	`VERSION` the version

Constructors

`TrailGuide()`

Methods

`public String`	`getAppletInfo()` get information about the applet
`public String[][]`	`getParameterInfo()` get the parameter information

`public void`	`init()` initialize the applet

D.2 ch.etc.trailguide.backend

This package contains classes for the access to the backend, i.e. the graph server.

ch.etc.trailguide.backend.GraphServer

```
java.lang.Object
```

`public GraphServer extends Object`

This class represents the graph server. It provides methods to access this graph server using the data interchange protocol and updates the graph model.

Constructors

`GraphServer(XmlRpcClient server, TrailGraphModel model, FeatureFactory features)` constructor

Methods

`public void`	`addLink(Link link)` add a link
`public void`	`deleteLink(Link link)` delete a link
`public void`	`getGraph(String root, int depth, Vector collections)` get the graph
`public void`	`rateLink(Link link, int rating)` rate a link
`public Document[]`	`search(String type, String key, String value)` search the model for a specific document

ch.etc.trailguide.backend.GraphServerException

```
java.lang.Object
  java.lang.Throwable
    java.lang.Exception
```

`public GraphServerException extends Exception`

GraphServer Exception

Constructors

`GraphServerException(String message)` constructor without cause exception
`GraphServerException(String message, Throwable cause)` constructor with cause exception

D.3 ch.etc.trailguide.framework

This package contains classes and interfaces of the generic visualization framework.

ch.etc.trailguide.framework.DefaultDocumentFactory

```
java.lang.Object
```

```
public DefaultDocumentFactory extends Object
implements DocumentFactory
```

Default Document Factory

Constructors

`DefaultDocumentFactory()`

Methods

`public Document`	`createDocument(String id, String label, Map attributes, FeatureFactory features)` create a new document

ch.etc.trailguide.framework.Document

```
java.lang.Object
  javax.swing.tree.DefaultMutableTreeNode
    com.jgraph.graph.DefaultGraphCell
```

```
public Document extends DefaultGraphCell
```

Document

Fields

`public int`	`distance` the distance from the root node
`protected FeatureFactory`	`features` the feature factory

Constructors

`Document(String id, String label, String mimeType, FeatureFactory features)` constructor

Methods

`public void`	`display()` display this document
`public final String`	`getId()` get the id of the document
`public String`	`getLabel()` get the label of the document
`public String`	`getMimeType()` get the mime type of the document
`public String`	`getToolTipText()` get the tool tip text
`public String`	`toString()` get the string representation

ch.etc.trailguide.framework.DocumentFactory

`public interface DocumentFactory`

Interface of a document factory

Methods

`public Document`	`createDocument(String id, String label, Map attributes, FeatureFactory features)` create a new document

ch.etc.trailguide.framework.FeatureFactory

`public interface FeatureFactory`

This interface defines methods for access to various GUI features, e.g. labels, colors, icons...

Methods

`public String`	`formatWeight(double weight)` format a weight
`public Color`	`getBackgroundColor(boolean current, boolean visited)` get the background color
`public Color`	`getCollectionColor(String collection)` get collection color

`public Color`	`getForegroundColor(boolean current, boolean visited)` get the foreground color
`public ImageIcon`	`getIcon(String mimeType)` get icon
`public String`	`getText(String key)` get the text for a key

ch.etc.trailguide.framework.Link

```
java.lang.Object
  javax.swing.tree.DefaultMutableTreeNode
    com.jgraph.graph.DefaultGraphCell
      com.jgraph.graph.DefaultEdge
```

```
public Link extends DefaultEdge
implements Comparable
```

Link

Fields

`protected FeatureFactory`	`features` the feature factory

Constructors

`Link(String id, double weight, String collection, FeatureFactory features)` constructor

Methods

`public int`	`compareTo(Object obj)` compare to an other object
`public static String`	`constructId(Document source, Document target, String collection)` construct a link id
`public String`	`getCollection()` get the collection identifier for this link
`public final String`	`getId()` get the id of the link
`public Document`	`getSourceDocument()` get the source document
`public Document`	`getTargetDocument()` get the target document
`public String`	`getToolTipText()` get the tool tip text

`public double`	`getWeight()` get the weight of the link
`public String`	`toString()` get the string representation of this link

ch.etc.trailguide.framework.TrailGraph

```
java.lang.Object
  java.awt.Component
    java.awt.Container
      javax.swing.JComponent
        com.jgraph.JGraph
```

`public TrailGraph extends JGraph`

This is the graph for the Trail Guide applications.

Constructors

`TrailGraph(TrailGraphModel model)` constructor

Methods

`public void`	`addLayoutAlgorithm(GraphLayoutAlgorithm algorithm)` add a layout algorithm
`public void`	`doLayout()` layout the graph
`public String`	`getToolTipText(MouseEvent event)` get the tooltip text

ch.etc.trailguide.framework.TrailGraphModel

```
java.lang.Object
  javax.swing.undo.UndoableEditSupport
    com.jgraph.graph.DefaultGraphModel
```

`public TrailGraphModel extends DefaultGraphModel`

This is the graph model for the Trail Guide applications. It extends the DefaultGraphModel of the JGraph framework and provides additional functionality.

Constructors

`TrailGraphModel(FeatureFactory features)` constructor

Methods

`public void`	`addDocumentFactory(String type, DocumentFactory factory)` add a document factory
`public Document`	`getDocument(String id)` get the document
`public Collection`	`getLinks()` get all the links
`public Document`	`getRootDocument()` get the root document
`public Document`	`insertDocument(String id, String label, String type, Map attributes)` add a new document
`public Link`	`insertLink(Document source, Document target, double weight, String collection)` add a new link
`public void`	`removeDocument(Document doc)` remove a document
`public void`	`removeLink(Link link)` remove a link
`public void`	`setRootDocument(Document rootDocument)` set the root document

D.4 ch.etc.trailguide.gui

This package contains classes and interfaces for the graphical user interface.

ch.etc.trailguide.gui.CellActionHandler

```
java.lang.Object
  java.awt.event.MouseAdapter
```

`public CellActionHandler extends MouseAdapter`

Cell Action Handler

Constructors

`CellActionHandler(JGraph graph, UserInterface ui, GraphServer server, FeatureFactory features)` constructor

Methods

`public void`	`mousePressed(MouseEvent event)` mouse button was pressed

ch.etc.trailguide.gui.TrailGuideFeatures

```
java.lang.Object
```

```
public TrailGuideFeatures extends Object
implements FeatureFactory
```

This class implements the FeatureFactory interface and so provides the specific TrailGuide features.

Constructors

`TrailGuideFeatures(Applet applet)` constructor

Methods

`public String`	`formatWeight(double weight)` format a weight
`public Color`	`getBackgroundColor(boolean current, boolean visited)` get the background color
`public Color`	`getCollectionColor(String collection)` get collection color
`public Color`	`getForegroundColor(boolean current, boolean visited)` get the foreground color
`public ImageIcon`	`getIcon(String mimeType)` get icon
`public String`	`getText(String key)` get the text for a key

ch.etc.trailguide.gui.UserInterface

```
java.lang.Object
```

```
public UserInterface extends Object
```

Graphical User Interface

Constructors

`UserInterface(Container container, JGraph graph, FeatureFactory features, GraphServer server)` default constructor

Methods

`public Vector`	`getCollections()` get the selected collections
`public Container`	`getContainer()` get the parent
`public JPopupMenu`	`getDocumentPopup()` get the popup menu for documents
`public JPopupMenu`	`getLinkPopup()` get the popup menu for links
`public int`	`getRecursionDepth()` get recursion depth
`public String`	`getSearchKey()` get the chosen search key
`public String`	`getSearchType()` get the chosen search type
`public String`	`getSearchValue()` get the search value

D.5 ch.etc.trailguide.layout

This package contains various graph layout algorithms.

ch.etc.trailguide.layout.GraphLayoutAlgorithm

```
public interface GraphLayoutAlgorithm
```

Defines the interface of a graph algorithm.

Methods

`public void`	`layout(JGraph graph)` layout the graph

ch.etc.trailguide.layout.LinkSplitLayout

```
java.lang.Object
```

```
public LinkSplitLayout extends Object
implements GraphLayoutAlgorithm
```

This class implements an layout algorithm that handles multiple links between two documents. The links are altered from straight lines to curves, such that all the links are visible.

Constructors

`LinkSplitLayout()`

Methods

`public void`	`layout(JGraph graph)` layout the graph

ch.etc.trailguide.layout.SpanningTreeLayout

```
java.lang.Object
```

```
public SpanningTreeLayout extends Object
implements GraphLayoutAlgorithm
```

This class implements a graph layout algorithm using spanning trees. The spanning tree of the graph is calculated. For conflicting links, the link with the bigger weight is chosen.

Constructors

`SpanningTreeLayout(int circleRadius, Dimension vertexSize, float fontSize, int cutDistance)` constructor

Methods

`public void`	`layout(JGraph graph)` layout the graph

D.6 ch.etc.trailguide.web

This package contains specific classes and interfaces for the trailguide graph visualization for web documents.

ch.etc.trailguide.web.WebDocument

```
java.lang.Object
  javax.swing.tree.DefaultMutableTreeNode
    com.jgraph.graph.DefaultGraphCell
      ch.etc.trailguide.framework.Document
```

```
public WebDocument extends Document
```

Web Document

Constructors

`WebDocument(String id, String label, String mimeType, URL url, AppletContext appletContext, String contentWindow, FeatureFactory features)` constructor

Methods

`public void`	`display()` display the content of this object
`public String`	`getToolTipText()` get the tool tip text
`public URL`	`getURL()` get the url

ch.etc.trailguide.web.WebDocumentFactory

```
java.lang.Object
```

```
public WebDocumentFactory extends Object
implements DocumentFactory
```

Web Document Factory

Constructors

`WebDocumentFactory(AppletContext appletContext, String contentWindow)` constructor

Methods

`public Document`	`createDocument(String id, String label, Map attributes, FeatureFactory features)` create a new document

Bibliography

[Ald02] Gaudenz Alder. Design and implementation of the JGraph Swing component. http://jgraph.sourceforge.net/paper.html, March 2002.

[BB96] Chris C. Brown and Steven D. Bendfort. Tracking WWW users: Experience from the design of HyperVIS. In *Proceedings of WebNet'96: World Conference of the Web Society*, October 1996.

[BETT94] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollics. Algorithms for drawing graphs: an annotated bibliography. *Computational Geometry: Theory and Applications*, 1994.

[Bra96] Tim Bray. Measuring the web. In *Proceedings of Fifth International World Wide Web conference*, pages 993–1005, May 1996.

[BTB+99] Steve Benford, Ian Taylor, David Brailsford, Boriana Koleva, Mike Craven, Mike Fraser, Gail Reynard, and Chris Greenhalgh. Three dimensional visualization of the World Wide Web. *ACM Computing Surveys*, 31(4es), December 1999.

[CK95] Jeromy Carriere and Rick Kazman. Interacting with huge hierarchies: Beyond cone trees, 1995.

[CRY96] Stuart K. Card, George G. Robertson, and William York. The WebBook and the Web Forager: An information workspace for the World-Wide Web. In *Proceedings of ACM SIGCHI '96*, April 1996.

[DK98] David G. Durand and Paul Kahn. Mapa: A system for inducing and visualizing hierarchy in websites. In *Proceedings of ACM Hypertext '98*, pages 66–78, June 1998.

[Doe94] Peter Doemel. Webmap — a graphical hypertext navigation tool. In *Proceedings of the Second International World-Wide Web Conference*, 1994.

[FHH90] Andrew Fountain, Wendy Hall, and Ian Heath. Microcosm: An open model for hypermedia with dynamic linking. Technical report, University of Southampton, 1990.

[FS98] Emmanuel Frécon and Gareth Smith. Webpath - a three-dimensional web history. In *Proceedings of IEEE Symposium on Information Visualization (InfoVis '98)*, pages 3–10, 1998.

[HDWB95] Robert J. Hendley, Nicholas S. Drew, Andrew Wood, and Russel Beale. Narcissus: Visualizing information. In *Proceedings of the 1995 Information Visualization Symposium*, pages 90–96, 1995.

[HMM00] Ivan Herman, Guy Melancon, and M. Scott Marshall. Graph visualization and navigation in information visualization. *IEEE Transactions on Visualization and Computer Graphics*, Vol. 6, 2000.

[LR94] John Lamping and Ramana Rao. Laying out and visualizing large trees using a hyperbolic space. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 13–14, 1994.

[LRP95] John Lamping, Ramana Rao, and Peter Pirolli. A focus+context technique based on hyperbolic geometry for visualizing large hierarchies. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems*, 1995.

[RMC91] George G. Robertson, Jock D. Mackinlay, and Stuart K. Card. Cone trees: animated 3d visualizations of hierarchical information. In *Human Factors in Computing Systems Conference Proceedings on Reaching through Technology*, pages 189 – 194, 1991.

[SB92] Manojit Sarkar and Marc H. Brown. Graphical fisheye views of graphs. In *Conference Proceedings on Human Factors in Computing Systems*, pages 83 – 91, 1992.

[SBG+97] David Snowdon, Steven D. Bendford, Chris M. Greenhalgh, Rob Ingram, Chris C. Brown, Lennhart Fahlen, and Marten Stenius. A 3d collaborative virtual environment for web browsing. In *Virtual Reality Universe '07*, April 1997.

[SEN00] Beat Signer, Antonia Erni, and Moira C. Norrie. A personal assistant for web database caching. In *Proceedings of the 12th Conference on Advanced Information Systems Engineering (CAiSE'00)*, June 2000.

[SM97] Hidekazu Shoizawa and Yutaka Matsushita. Www visualization giving meanings to interactive manipulations. In *Advances in Human Factors/Ergonomics 21B (HCI International '97)*, pages 791–794, August 1997.

[Win99] David Winer. XML-RPC specification. http://www.xmlrpc.com/spec, 1999.

List of Tables

List of Figures

Index