UNIVERSIDAD AUTONOMA DE MADRID

Departamento de Ingeniería Informática

An Approach to the Visualization of Adaptive Hypermedia Structures and other Small-World Networks based on Hierarchically Clustered Graphs

Dissertation written by **Manuel Freire Morán**
under the supervision of **Pilar Rodríguez Marín**

Madrid, 3rd July 2007

Contents

List of Figures

List of Tables

Abstract

Large graphs are difficult to represent and visualize in their fully expanded form. Those that exhibit the small-world property are amenable to abstraction via hierarchical clustering, allowing the user to select the desired degree of detail for each part of the graph. However, performing this abstraction presents several problems, ranging from the construction of the cluster hierarchy to the preservation of user orientation during navigational actions. This work analyzes each problem in turn, and discusses strategies to address them.

A domain-independent framework, the *CLuster-Oriented Visualization EnviRonment*, or CLOVER for short, has been designed and developed based on the above-mentioned strategies, and is available to any interested parties as an open-source library. The WOTED course authoring tool, built on CLOVER, demonstrates the applicability and use of the proposed visualization approach in the initial target domain, Adaptive Hypermedia.

Applications based on CLOVER for other domains where small-world networks can be found have also been developed: document repositories that support fragment reuse, knowledge representation with ontologies, social networks underlying student assignment similarities, and networked appliances in an intelligent home.

Resumen

A medida que se incrementa el tamaño y la complejidad de un grafo, aumentan también las dificultades para representarlo y visualizarlo. Si el grafo cumple la propiedad de "mundo pequeño", es posible resumirlo mediante clusterización (agregación) jerárquica. Con una interfaz apropiada, un usuario podría modificar, de forma interactiva, el nivel de detalle usado en la representación de cada zona de un grafo así resumido. Llevar a cabo esta propuesta presenta varios problemas, desde la construcción de la jerarquía de clústeres a los problemas de orientación que acarrea la navegación a través de los distintos niveles de detalle del grafo clusterizado. El presente trabajo analiza estos problemas, y compara y propone estrategias para hacerles frente.

Como realización de las estrategias anteriores se ha desarrollado CLOVER, cuyas siglas corresponden a la traducción al inglés de *Entorno de Visualización Orientado a CLústers*. Este *framework* se puede usar para visualizar grafos provenientes de diversos campos, y está a disposición de cualquier interesado como una librería de software libre. Sobre CLOVER se ha implementado WOTED, una herramienta de autor para cursos hipermedia adaptativos, demostrando la aplicabilidad y utilidad de la propuesta inicial para este dominio en particular.

Se describen también varias otras aplicaciones basadas en CLOVER, desarrolladas para campos que sólo tienen en común la existencia, en todos ellos, de redes de mundo pequeño. Las aplicaciones desarrolladas abarcan un sistema de composición de documentos mediante fragmentos reutilizables, una aplicación para la representación de conocimiento mediante ontologías, una visualización de orientada a detectar casos de copia en prácticas de programación, y una herramienta para monitorizar la interconexión de dispositivos dentro en una red domótica.

Acknowledgements

This thesis is dedicated to Asun, for her guidance and support in so many ways.

Pilar, friend and tutor, deserves a special mention for her insightful comments, constructive criticism, and sharp sense of humour. May we share many more coffees.

To José Manuel and Asun, my parents, for their patience and support, and to Laura, my sister, for all those trips to Liverpool that I owe her. And to the rest of my wonderful family, from uncles to cousins, in no particular order: Juan, Poe, Concha, José Luis, Fran, Manuel, Pepa, Mari, Gary, José Mari, Mari, Laura, Antonio, Paco, Isabel, Hugo, Aida, Jon, Xabi, David, Irene, Queco, Raqui, Quique, Javi, Lourdes, Uge, Sara, Luis, Ana, Cati, Ryan and JP (what a bunch!), and to those that are no longer with us, Ana, Francisco, Encarna and Manuel. And to my proud grandmother María.

To friends from different walks of life, far and near. In no particular order, Mar, César, Samu, Javi, Jaime, Ramón and Maca; Álvaro, Laura, Suso, Ignacio, Sebas, Rafa, Rebeca, Eva and Ignacio (luck with those exams!); Jose, Marta, Luisma, Rosi, Alex, Nuria, Miguel, Sandra, Jose, Nacho, Luis, Julio, Arkaitz, Joe, Tania, Laura, Carlos, Claudia, Juan and Carmen; Diana, María Isabel, Teodora, and María Martín; José and Cristina; Emi and Ana; Isabel. May we always keep in touch.

To the colleages of the B-207 and B-406 labs, Leila Shafti, Estefanía Martín, Pedro Paredes, Diana Pérez-Marín, Ismael Pascual, Javier Bravo, Francisco Pérez, Pablo Haya, Abraham Esquivel and Manuel Herranz; and those that have since moved to higher floors, Germán Montoro, Ruth Cobos, Álvaro Ortigosa, Rosa Carro and Miguel Mora; or far-away lands, Enrique Alfonseca, María Ruiz and Abdellatif Abu Dalhoum. To my colleagues at the the EPS, too many to list, and to Juana Calle, our department secretary, for her patience and efficiency. And to my students, who have taught me much and suffered my teaching.

To the Institut für Informatik at the Freie Universität Berlin, and particularly to Prof. Günter Rote, who first interested me in graph visualization, and Prof. Rojas and the heroic FU-Fighters. To Lehrstuhl XI at the Technische Universität München, and Prof. Johann Schlichter, who provided valuable comments during the first steps of Clover, and the great people I met there – Elena, Frank, Georg, Peter, Martin, Rosmary, and Evelyn. To the LUSSI department at the ENST-Bretagne, and to Serge Garlatti, to whom I still owe an article, and Cuong, Jean Louis, Jean Marie and all the other colleagues I met during my stay in Brest.

To my colleagues in conferences abroad, for their input and ideas; Peter, Michael, Sergei, Alexandra, Craig, Nora, Markku, Olga and Natasha. And to all those researchers that I have not yet met, but whose ideas I have read and shared, and without which a work like this would never have been possible.

And to José Antonio Macías for his valuable comments, and Günter Rote, Marcus Raitner and Terry J. Anderson for their letters of support.

This research has been partially funded by a Third-Cycle Help from the Universidad Autónoma de Madrid; an F.P.I. scholarship from the Ministry of Education and by CI-CYT project grants TEL1999-9181 (Encitec), TIC2001-0685 (Ensenada) and TIN2004-03140 (U-Cat).

Chapter 1

Introduction

This work presents an innovative approach to adaptive hypermedia authoring and small-world graph visualization.

The first section of this introduction is concerned with the motivation for the present work. The second section lists the theoretical issues that are encountered when analyzing the problem. A short description of the chosen approach is then presented. Finally, the overview section provides the reader with a bird's view of the organization and contents of the work itself.

1.1 Motivation

Graphs are a convenient way of representing information for many classical domains, such as software programs, network topologies, or co-citation maps. In many of these domains, graphs have an internal structure which a good representation should highlight. For instance, in software call graphs or citation networks, *clusters* (groups) of highly interrelated vertices that communicate only weakly with "outside" vertices can be found. Interactive graph-based interfaces are desireable to browse and edit data for these domains. However, as graphs increase in size, interactive interfaces are hampered by information overload and low responsiveness.

Information overload is partially due to the difficulties of integrating both low-level details and their overall context, a problem common to all interfaces. Graph representations also exhibit the additional problem of high sensitivity; addition or removal of a few edges may have important effects on the overall structure of a graph, and call for a very different representation. Updating a representation can easily result in low responsiveness; and if the changes from the original to the updated representation are not easy to follow, a user would be forced to "relearn" the graph.

The above issues were encountered while designing a new authoring tool for an existing hypermedia course system. The initial plan included a tree-based representation, since courses had a mostly tree-like structure, with learning "tasks" divided into subtasks, which could in turn have their own subtasks. However, a task could have more than one "parent" task – and this was difficult to expose with a tree representation. Although mostly tree-like, the system's courses were actually directed acyclic graphs, and this motivated the design of a second authoring tool that used a simple graph representation. Graphs for most courses were large and difficult to layout; this contrasted with the simplicity of trees, where the degree of detail is easy to adjust by expanding and collapsing branches. However, since courses were "almost" trees, and trees are easy to use, why not try to merge their advantages and the expressive power of graphs? How could this be applied to graphs, substituting clusters (groups of highly related vertices) for branches?

As in tree branches, which can be seen as children of the branches that contain them, graph clusters can be used as children of new higher-level clusters, leading to a hierarchical clustering. Expansion and collapse operations of a hierarchically clustered graph can then be defined, and the problem would be solved – except for certain important issues. First, while the definition of a "branch" is unambiguous (all descendants of a given tree node), that of a "cluster" is not, and good heuristics or user intervention are needed to define and maintain the cluster hierarchy. Second, graphs are much more complex to layout than trees (and indeed, the field of graph drawing has attracted considerable attention during recent years). Manual layout is not a problem for small, static graphs; but automatic layout is required if the visible portions of the graph can change over time as a result of expand/collapse operations. Additional measures are also necessary to avoid disorienting the user: after an expansion or collapse, the "same" graph will be visible, but with a different level of detail; the updated layout must not be allowed to vary too wildly, or the user would be forced to "relearn" the graph after each browsing operation, which is clearly not desireable. Finally, due to the simple correspondence between tree structure and layout, updating a partially collapsed tree representation after a change in its structure is relatively simple; updating the representation for a clustered graph (which may also involve an update to the clustering itself) is much more complex.

However, these obstacles did not seem insurmountable, since there was abundant prior art for most of the individual issues (clustering, graph drawing, disorientation, and so on). After some initial work, the application-independent part of the previously mentioned adaptive-hypermedia authoring tool was separated from the application-dependent code, giving birth to a graph visualization framework based on hierarchical clustering. Reading Watts and Strogatz's[137] article on *small-world graphs* –graphs that were highly structured (and therefore clusterable), but not entirely regular– and their prevalence in many real-world domains further motivated the development of the visualization framework, and the design and implementation of applications that could exemplify its use in very different areas.

This was the motivation for the present work: an innovative approach to graph visualization, based on hierarchical clustering and dynamic graph browsing.

1.2 Theoretical Issues

The approach that has been outlined in the motivation section involves several fields of knowledge. Part I, Preliminaries provides an introduction to each of these fields. This section provides an overview of the contents of Part I, providing short descriptions of each field and the motives that have lead to its inclusion.

Work on this approach began as an effort to provide better visualization for an existing Adaptive Hypermedia course system. The goal of *Adaptive Hypermedia* (AH) is to adapt hypermedia spaces (covering any type of linked media, and including adaptation of the links themselves) to individual users, instead of pursuing a one-size-fits-all approach. This requires a *user model* (a representation of the user's characteristics, background and goals) to be built and updated by the AH system. The user model is then used to decide what to show to each particular user, and how to tailor it for presentation. AH systems differ in their goals (for instance, education, search or reference), and several important differences exist regarding the type of user model and how it is represented, acquired and updated, the types of adaptation supported, and the techniques applied to implement adaptation.

Furthermore, Adaptive Hypermedia requires the preparation of different views of the contents of the system's domain; the adapted contents are easier to create and manage in the form of loosely-coupled modules. Given the high cost of preparing these modules, it is important to allow reuse in other domains or contexts. Module reuse requires the presence of machine-readable information on its contents and intended outcomes, the context where it makes sense (for instance, knowledge prerequisites), and how it can be adapted to fit into this context. These descriptions are usually provided as machine-processable metadata. Standardized metadata can be achieved through well-defined ontologies, a knowledge representation formalism that provides conceptualizations of entities and their relationships, and can be backed up with support for reasoning. In addition, ontologies themselves are amenable to representation as graphs.

Once created or retrieved, AH modules have to be authored into a coherent AH space, which should then be tested and maintained. Authoring adaptation can be a complex task, since these systems work by gathering information about each user in an constantly-updated user model, and then use this information to select what to present, and how it should be structured or highlighted. Authoring tools are needed to allow domain experts to create these AH spaces. But these domain experts should not be required learn computer science or become experts in the AH system itself. The initial authoring tool intended to fill this gap for one such system, using a graph visualization to represent the relevant information of the adaptive space being edited.

The use of a graph visualization for an AH authoring tool is natural, since graphs have been used extensively to represent the link structure of classical hypermedia spaces. These representations can be readily extended to visualize the structure of many AH systems. Graphs themselves, as a mathematical construct, belong to the field of Graph Theory, a part of Discrete Mathematics and Combinatorics. The related field of Graph Drawing deals with their planar and spatial representation. But since large graphs are difficult to render and interpret (specially on small areas such as screens), *clustering* can be used to abstract away unnecessary detail, by collapsing groups of tightly related vertices into single 'cluster' vertices. The section on graph clustering includes a discussion of *graph grammars*, which can be used to locate clusters based on pattern-matching rules described in terms of local graph topology.

Clustering only works if the graph contains well-defined clusters to begin with; hierarchical clustering further requires that this is also true for abstracted versions of the graph. Many natural and artificial graphs exhibit the *small-world* property: a high degree of clustering when compared to randomly generated graphs of similar size and density, but similar average path lengths. This property is related to scale invariance and self-similarity. The small-world property, scale invariance and self-similarity (in the loose sense of "properties") have been described, in different degrees, for graphs obtained from computer programs, in friend-of-a-friend networks (FOAF), scientific citations, power grids, protein interaction networks, and the world-wide web. Therefore, hierarchical clustering would seem to be an ideal solution to visualize a large family of graphs.

Finally, when representing graphs or other types of abstract information, the field of *Information Visualization* (IV) comes into play. Information Visualization is dedicated to the study of the most effective ways to present abstract information to users, and in this context, it is strongly related to *Graph Drawing.* IV can also be considered an important part of *Human-Computer Interaction* (HCI), a field that studies interface between humans and computers, but considering the whole process of representation and user feedback required to perform tasks on a computer.

1.3 Approach and Clover Framework

The heart of the proposal is contained in the design and implementation of the Clover (CLuster-Oriented Visualization EnviRonment) framework. This framework provides a domain-independent base on which to create interactive hierarchical-clustering graph visualizations. While Clover can be used to visualize any graph structure, hierarchical clustering is specially well suited to graphs with the small-world property: low average path lengths but high degree of clustering.

Clover is designed as a pipeline, illustrated in fig. 1.1. The initial data is transformed into a graph, which is then filtered and clustered, subject to selection of the "most interesting" clusters, laid out, and rendered on screen. Each step of the pipeline transforms the the previous steps' output into something required by the next one, until the resulting view is displayed to the user. However, it could also be thought of as a loop, since the user can manipulate the view itself, browse the graph via expansion and collapse of clusters (or indirectly by selecting different 'areas of interest': the black triangles in the far right of fig. 1.1), change the filtering, or even modify the underlying graph, which would be translated into a change in the original data. Any of these changes would trigger an update in the remaining "pipeline" steps, leading to an updated view of the graph. Multiple views of the same graph, differing in any of the intermediate stages, can be maintained simultaneously, supporting a multifaceted visualization of the data.

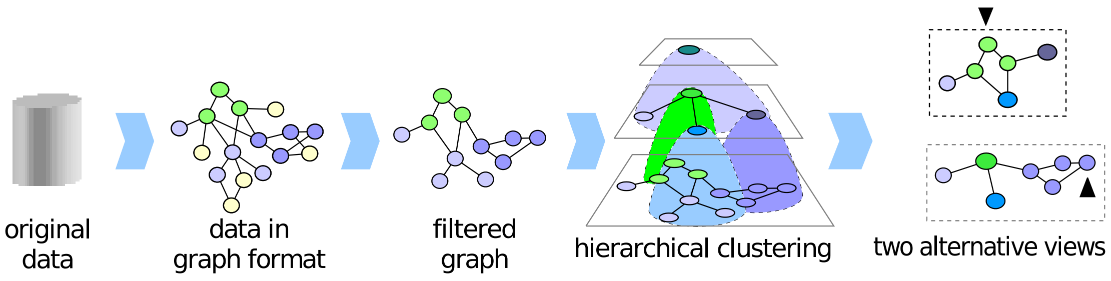


Figure 1.1: A simplified CLOVER pipeline

Several issues which are commonly found with graph visualizations have been dealt with in CLOVER. Layout is performed automatically, allowing users to start to work with graphs without the need to do a manual layout first; but manual tuning is allowed and preserved, allowing intuitive layout customization. During user browsing, automatic incremental layout is used to make extra space for new vertices resulting from cluster expansion or reabsorb space left by a collapse. The previous layout is changed as little as possible. Any changes to the graph are highlighted and smoothly animated to focus attention only on changes and keep the user oriented. And layout predictability is enhanced by means of a "history" mechanism that reuses previous layouts whenever possible.

The architecture of CLOVER is highly modular, and all steps of the default behavior can be altered or extended to suit the needs of the application that is using the framework. Only the initial graph generation step is needed to achieve a fully-functional visualization. However, best results may require tuning graph filtering, clustering strategy, layout parameters, vertex and edge representations, animations and feedback control to match the application's requirements. CLOVER's modular architecture makes it easy to customize for any domain, as the multiple CLOVER-based applications described in this work can attest.

Compared to existing graph visualization approaches, the main contributions of

Clover are the use of hierarchically clustered graphs that allow the representation of large graphs at arbitrary degrees of detail; and the fully automated nature of the visualization pipeline, where changes at any step are propagated, without need of user intervention, to all dependent views, trigerring animated incremental updates.

The main testbed for Clover is WotEd, a graph-based editor for the Wotan Adaptive Hypermedia course system. Using Clover, WotEd represents AH courses as clustered graphs, allowing large, complex courses to be displayed and modified – even if they were created or modified outside WotEd. Additionally, WotEd introduces innovative monitoring support, where student progress through a course can tracked on the same interface with which the course was originally created. Monitoring relies on the ability of Clover to automatically animate updates to a graph's structure on any view of this graph, which is one of its most distinctive features.

1.4 Overview

The work is organized into three parts:

Part I – Preliminaries Sets the groundwork for the approach presented in Part II, with an introduction to each of the fields of knowledge touched by the present work. Terms and concepts used in the approach are introduced and described.

Part II – Approach Presents the approach itself. First, a review of graph visualization strategies is performed; then, the design of Clover is presented. The next chapters introduce the application of Clover to Adaptive Hypermedia authoring and other domains.

Part III – Conclussions and Future Work Contains the conclussions, a brief discussion of important aspects, and outlines future work.

Additionally, Appendix A contains a translation of this first chapter into Spanish. Appendix B is dedicated to implementation notes and other technical information relevant to Part II.

Part I

Preliminaries

Chapter 2

Graphs and the Small-World Property

Graphs of different types are all around us. Road maps can be seen as collections of towns and roads that link them. Computer networks can be analyzed as boxes and cables. Hierarchies, flow diagrams and biological interactions can also be represented as sets of items (*vertices*) connected to each other via *edges*. Although the use of graph-like representations is much older, the formal characterization of graphs as abstract mathematical entities can be dated back to the 18th century and Euler's famous "Seven Bridges of Königsberg" problem (depicted in fig. 2.1). Since then, graphs have found a widespread use in many branches of mathematics, most importantly the field of *Graph Theory*.

This chapter starts with a set of definitions for the basic graph-theoretic terms that are used throughout the work. The second section examines the *small-world* property and related phenomena, which are shown to exist in many interesting domains, including the web, device networks, social networks and neural networks. These phenomena have a direct impact on the possibility of using clustered graphs to generate meaningful graph abstracts. The last two sections deal with graph clustering and the superposition of a hierarchy on a graph, resulting in hierarchically clustered graphs.

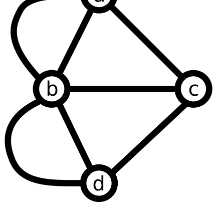


Figure 2.1: Graph of the the "seven bridges of Königsberg" problem. Is it possible to visit all vertices crossing each edge exactly once?

2.1 Graph Theory

Graph Theory is a subfield of Discrete Mathematics. It is related to the fields of Combinatorics, Group Theory, and Topology. This section presents the basic graph theory needed to understand the remainder of this work; an interested reader is pointed to [71] or [110]. It must be noted that each source uses a slightly different terminology, and that certain definitions can also vary depending on their source. However, these differences, when they arise, are of a minor nature: the underlying ideas can be easily identified.

From a mathematical standpoint, a *graph* G is a collection of *vertices* V and *edges* E, where each edge $e = (u, v)$ connects two vertices $u, v \in V$: $G = \{V, E\}$ with $E \subseteq V \times V$. In $e = (u, v)$, u is the *source* vertex of the edge, and v is the *target*. This can be written as $source(e) = u$ and $target(e) = v$, respectively.

Other authors prefer to use *nodes* and *links* instead of vertices and edges; in this work, *nodes* are only used when referring to the vertices of trees (a special type of graphs). Likewise, the term *branches* will be used to refer to tree edges. Also, in some applications, the term *network* is preferred to that of graph, perhaps attaching slightly different meanings, such as requiring edges to be labeled with weights (this is typical, for instance, in biology and electrical engineering). In this work, “graph” and “network” will be considered equivalent, and the choice of term will reflect common usage in the area under discussion.

If the order of vertices on an edge is not considered important (and E is considered a set of *unordered pairs* $\{u, v\}$), the graph is said to be *undirected*. Undirected graphs are equivalent to *directed* graphs (also called *digraphs*) where, for each undirected edge $\{u, v\}$, edges (u, v) and (v, u) can be found. A special case are *loop* edges, which include a single vertex: $\{v, v\}$ is considered equivalent to (v, v).

An undirected version of a directed graph can be built by substituting, for all pairs of vertices u and v with at least one edge between them, any directed edges (u, v) or (v, u) by a single undirected edge, $\{u, v\}$.

A graph is said to be a *multigraph* if there is more than one edge that connects the same vertices. Graphs may have labels attached to edges or vertices; this is very common when the graph is used, for instance, to represent knowledge: vertices could represent concepts or concept classes, and would be labeled accordingly, while edges typically represent relationships between concepts. Since it is frequent to have pairs of vertices that can be related in more than one way, multigraphs are a natural fit for these applications. Other cases where it may be desireable to have more than one edge between a pair of vertices may arise; for instance, the “seven bridges of Königsberg” graph of fig. 2.1 is an undirected multigraph.

If multiple edges between the same two vertices are not allowed, then there is an upper bound to the number of possible edges in an undirected graph, $|V| \cdot (|V| - 1)/2$.

A graph with n vertices and all possible edges is said to be the *complete graph* of order n, and can be written as K_n. This notion of "completeness" is linked to the *density* of a graph: the number of edges as compared to the number of vertices. Graphs with n vertices and $\sim n \cdot (n-1)/2$ edges are *dense*, while graphs with $\sim 2n$ edges or fewer are considered *sparse*.

A graph vertex v is said to have *outgoing degree* (or *out-degree*) n when there are exactly n edges with v as source. In a similar way, the *in-degree* of v is defined as the number of edges that have v as a target. In undirected graphs, the *degree* of a vertex v is simply the number of (undirected) edges that contain v.

Subgraphs

If $G = (V, E)$ is a graph, a graph $G_1 = (V_1, E_1)$ is a *subgraph* of G iff V_1 is non-empty and $V_1 \subseteq V$. Since G_1 is also a graph, it follows that E_1 can only contain edges that were present in E, and only those whose endpoints are still present in V_1. It may, however, include less edges than it could. If E_1 includes all edges present in G for the chosen subset of vertices V_1, then G_1 is said to be the *induced subgraph* of G. A special notation exists for this case. If U is a subset of the vertices in V, then $\langle U \rangle$ represents the subgraph of G induced by U.

Other important subgraphs result when a single vertex, or a single edge, is removed from the original graph G. In the first case, $G - v$ represents the subgraph of that results when vertex $v \in V$ is removed (together with all edges $e \in E$ such that $source(e) = v$ or $target(e) = v$). Similarly, $G - e$ is the subgraph of G that results after removing e from the the set of edges, E.

When a subgraph retains all vertices found in the original graph G, it is called a *spanning subgraph*. For instance, the subgraph $G - e$ with $e \in E$ is always a spanning subgraph. Note that this does not imply connectedness of the graph, which will be introduced in the following subsections.

Walks, paths and cycles

If x and y are two vertices (not necessarily distinct) in V, an *x, y walk* is a finite sequence of vertices and edges such that each edge in the sequence takes off where the last one left. Both edges and intermediate vertices may be repeated along the sequence:

$$x = x_0, e_1, x_1, e_2, x_2, \ldots, e_{n-1}, x_{n-1}, e_n, x_n = y$$

where $\forall i \in [1, n]$, $e_i = (x_{i-1}, x_i)$. The *length* of the walk is n, the number of edges.

When a walk has no duplicate intermediate vertices, it is said to be a *path*. A closed path is a *cycle*. Similar terms (*trail* and *circuit*) are used when there are no duplicate edges. A *directed acyclic graph*, that is, a directed graph without any cycles, is often termed *DAG*.

The set of vertices that are connected through a path of length 1 ("a short walk away") to a given vertex form the *neighborhood* of that vertex, and may be written as k_v. That is, if $G = (V, E)$, and $v \in V$, then $k_v = \{u | \ \exists \ e = (v, u), e \in E\}$.

Connectivity

An undirected graph is *connected* if there is a path between any two distinct vertices. For a directed graph, connectivity is defined in relation to an "associated undirected graph", obtained by discarding the directions of every edge, and considering multiple edges between two vertices as a single edge. The original, directed graph is considered connected if and only if the associated undirected graph is also connected.

A maximal set of connected vertices is said to be a *connected component*. The number of connected components of a graph is denoted by $\kappa(G)$. For example, since all the vertices of a connected graph form a single connected component, if G is connected then $\kappa(G) = 1$.

Let $G = (E, V)$ and $v \in V$. If G was connected, but $G - v$ (the graph that results after removing v) is not, then v is an *articulation point* of G. More generally, v is an articulation point iff $\kappa(G - v) > \kappa(G)$. A similar concept can be applied to edges: $e \in E$ is termed a *bridge* of $G = (E, V)$ iff $\kappa(G - e) > \kappa(G)$. It can be seen that if e is a bridge then both of its endpoints are bound to be articulation points. An intuitive characterization of an articulation point or bridge is that of a "narrow pass" between regions of a graph, such that any path that crosses from one region to the other must include this vertex or edge.

The *distance* between any pair of vertices can be defined as the length of the shortest path from one to another, if such a path exists, or infinity if there is no path between them. In a connected graph, the graph's *diameter* is the greatest distance between all pairs of vertices, and the *characteristic path length* is the average distance between all pairs of vertices. Note that, if the graph is not connected, diameter and characteristic path length are not defined.

Trees

A special case of connected graph is the *tree*: a connected graph without cycles. In a tree, there is a single path from each vertex to every other – that is, every edge is a bridge, and all "inner" vertices are articulation points. Trees are interesting because they have the minimum number of edges required to keep the graph connected (which can be seen to be $|E| = |V| - 1$), and because they arise in many practical problems and graph-related algorithms. It is customary to use T instead of G to represent a tree. In cases where the graph has multiple components, but each of these components is a tree, the graph is termed a *forest*.

A directed tree $T = (V, E)$ is said to be *rooted* if a vertex $r \in V$ exists (the *root* of the tree) such that r has no incoming edges and it is possible to reach all other vertices through a path that starts in r. Additional terminology is used when dealing with the edges of a rooted tree: in $e = (u, v)$, $e \in E$, u is said to be the *parent* of v, and conversely, v is termed the *child* of u. If two vertices u_1 and u_2 share the same parent, they are called *siblings*. All vertices reachable through directed edges from a vertex v are *descendants* of v, and conversely, the set of all vertices that can reach v through directed edges form the *ancestors* of v (note that $v = descendants(v) \bigcap ancestors(v)$). Rooted trees are frequently used to represent containment hierarchies, such as computer filesystems or the table of contents used in this work.

Graphs and trees are very often used together. For instance, it is typical to use *spanning trees*, a tree that is also a subgraph, and includes all of the original graph's vertices, as skeletons of a full graph.

Since during most of this work trees will be used alongside with graphs, and both will have a degree of overlap in their vertices and possibly even edges, a different terminology will be used to distinguish between both. Directed tree edges that are not in the graph will be called *branches*, and tree vertices will sometimes be referred to as *nodes*.

2.2 Small-World Networks and Real-World Graphs

Many graphs that arise in Nature, such as the connections in an animal's nervous system, or the interactions between proteins, are neither entirely random nor do they adhere to a strict structure. It turns out that a large number of human-made graphs, such as friend-of-a-friend networks, power grid layouts, computer code dependency graphs and author citation networks exhibit remarkably similar properties. This section introduces several concepts and observations that characterize such real-world graphs, and which should be taken into account when preparing to visualize them. An in-depth review of the characteristics and generating models of complex real-world graphs can be found in [28].

Small-world networks were first described by Watts and Strogatz in 1998 [137], and were named after Milgram's "small world problem" [96], who performed a well-known experiment on the existence of surprisingly short chains of acquaintances between random pairs of persons. They are characterized as connected graphs where the typical degree k is $1 \ll k \ll |V|$, and that are neither completely random nor completely regular, but instead exhibit phenomena from both extremes: low graph average path length and, at the same time, a high degree of clustering. Low average path length is typical in random connected graphs – it can be intuitively explained by the low relative probability of any vertex to become isolated from the rest through a random rewiring procedure [137].

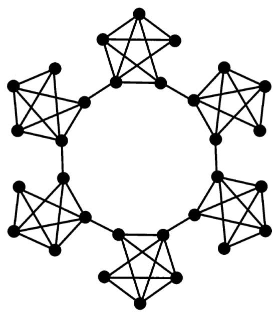


Figure 2.2: Connected cavemen graph, from [136]. This graph has a very high clustering coefficient (see eq. 2.1), but does not satisfy requirements for small-world, since the average path length increases linearly with the number of "caves".

Watts and Strogatz define the *clustering coefficient*[137] of an undirected graph $G = (V, E)$ as

$$C(G) = \frac{1}{|V|} \sum_{v \in G} \frac{|edges(k_v)|}{|k_v| \cdot (|k_v| - 1)/2} \tag{2.1}$$

where k_v is the set of neighbors of vertex v, and $edges(k_v)$ is the set of edges that exist between those neighbors, that is

$$edges(k_v) = \{e \in E : source(e) \in k_v \wedge target(e) \in k_v\} \tag{2.2}$$

The interpretation is that $|V| \cdot (|V| - 1)/2$ is the maximal number of edges that can exist between a set of vertices V; the degree of clustering is a measure of the ratio of the "allowable" edges that actually exist between the neighbors of each vertex. A high degree means that, for most vertices, their neighbors are also direct neighbors of each other (an intuitive description of "cluster"). In graphs with low clustering coefficient, however, vertices a and b with a common neighbor c are, generally speaking, not direct neighbors themselves. Applied to social networks, the concept of "clustering coefficient" has a natural mapping: your group of friends is tightly clustered if your friends are also friends of each other. Figure 2.2 provides an example of a graph with a high clustering coefficient.

Table 2.1 includes results on small-world characteristics of real-world graphs from several sources. Average path lengths of small-world graphs are similar to those of random graphs of the same size and density (edge-to-vertex ratio), but their clustering coefficient is orders of magnitude higher than that of "equivalent" random graphs. The surprise is that graphs from such different origins (including biological networks which were clearly not of human design) all share the same property.

From the point of view of graph visualization, two interesting results arise from the small-world property. First, the high clustering coefficient of many "natural" graphs makes clustering an interesting approach. And secondly, the fact that average distances

Row	Source	L_{actual}	L_{random}	C_{actual}	C_{random}
1	Film actors	3.65	2.99	0.79	0.00027
2	Power grid	18.7	12.4	0.080	0.005
3	C. Elegans	2.65	2.25	0.28	0.05
4	IMDB	3.2043	2.6694	0.9666	0.0243
5	Resyn assistant	3.2847	3.2847	0.9518	0.1942
6	Mac OS 9	2.8608	2.8608	0.3875	0.0179
7	.edu sites	4.062	4.062	0.156	0.0012

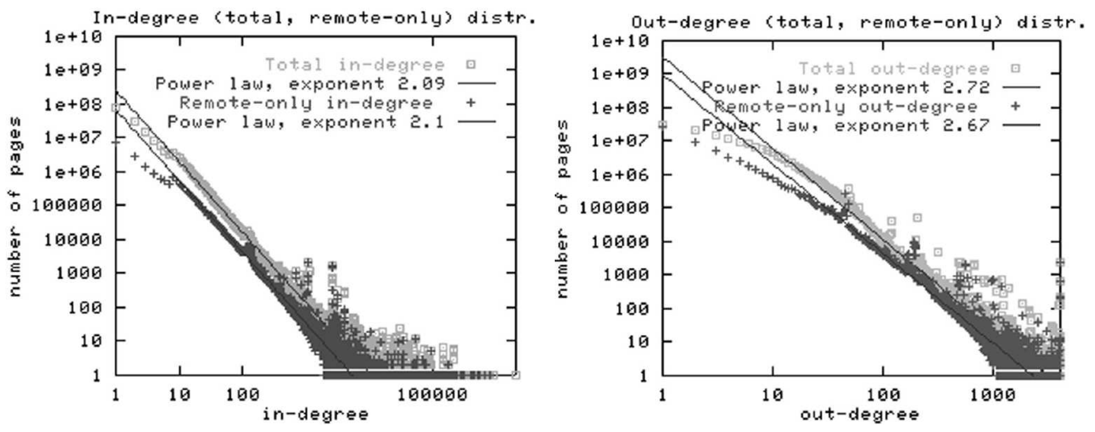
Table 2.1: Examples of small world networks, from [137] (rows 1–3) and [17] (rows 4–7). Rows 1 and 4 of this table represent graphs extracted from the Internet Movie Database (IMDB). In both cases, film actors were used as vertices, and the differences are due to how edges were created. Row 2 represents the layout of a power grid from western states of the USA. Row 3 is the map of the neural network of *Caernohabditis Elegans*, a nematode that has been extensively studied as a model organism. Rows 5 and 6 represent the dependencies within software applications, and the graph of row 7 was generated from a snapshot of the contents of the world-wide web's ".edu" top-level domain.

are low ($O(ln(|V|))$) makes direct navigation, as opposed to search, a viable option for browsing small-world graphs – if the complexity due to the large graph size can somehow be managed.

Scale Free Networks

In their discussion of small-world networks, Watts and Strogatz experimented using graphs where the distribution of the number of neighbors of each vertex was narrowly centered around a constant k (since the graphs were generated through random rewiring of a lattice where each vertex was connected to its k nearest neighbors). However, in most real-world graphs, including those presented in Table 2.1, this is not the case. Indeed, it is very common to find an exponential decay of the probability of having k neighbors. Networks with this property are said to be *scale free networks.* A particularly large graph that has been shown to be scale-free is the subset of the WWW analyzed in [33], numbering around 200 million vertices. The power-law distribution of vertex degrees in this subset is illustrated in fig. 2.3.

In a scale free network, the probability of a vertex having degree k follows a power law defined by $P(k) = k^{-\gamma}$, where γ is a constant that must be determined for each graph [23]. The power law may not hold through the whole range of vertex degrees, because in many cases there exist upper bounds to the maximum degree of a vertex (for instance, there are physical constraints to the number of movies that an actor can participate in); therefore, small-world graphs can be classified as scale-free, broad-scale, or single-scale [15], depending on the range of values of k where the power law is seen to hold.



(a) In-degree distribution (b) Out-degree distribution

Figure 2.3: Power-law distribution of in-degree (a) and out-degree (b) within the vertices of a sample of the WWW, from [33].

In [15], Amaral also notes that scale-free networks are many times also small-world (although this need not be true; in particular regarding the "low average path length" condition [28]), and that the example networks of [137] (rows 1-3 of Table 2.1) belong to both categories. The prevalence of this property in networks from vastly differing origins begs the question of a common pattern of creation.

Generation of Scale Free and Small World Networks

The following paragraph from Barabási and Albert [23] is illustrative in explaining the prevalence of small-world and free-scale graphs in cases where the graph has evolved through constant aggregation of additional nodes:

> [...] the random network models assume that the probability that two vertices are connected is random and uniform. In contrast, most real networks exhibit preferential connectivity. For example, a new actor is cast most likely in a supporting role, with more established, well known actors. Consequently, the probability that a new actor is cast with an established one is much higher than casting with other less known actors. Similarly, a newly created webpage will more likely include links to well known, popular documents with already high connectivity, or a new manuscript is more likely to cite a well known and thus much cited paper than its less cited and consequently less known peer. These examples indicate that the probability with which a new vertex connects to the existing vertices is not uniform, but there is a higher probability to be linked to a vertex that already has a large number of connections.

The model of graph generation proposed in [23], the extensions suggested by [15], and many alternatives that also seek to emulate observed natural networks properties in artificially generated graphs, are reviewed in [28].

Scale free networks and hierarchical clustering

The relevance of the small world property to graph representations is that, in those graphs where this property arises, it should be possible to exploit the high clustering coefficient to substitute "clusters" of vertices for abstracted versions, yielding a meaningful overview of the network. Local edges (edges that connect local cluster neighbors together) can be abstracted with low information loss, while longer-range edges (spanning distant clusters) that determine the macro-structure of the graph are preserved and uncluttered, enabling a better overview.

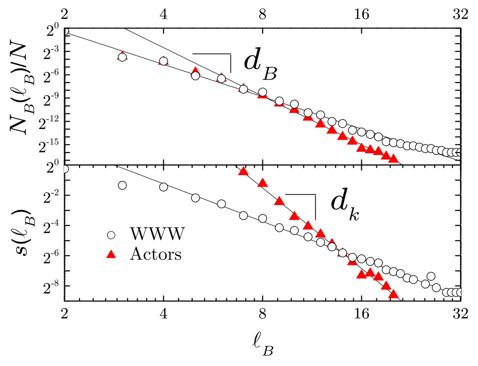
The idea of abstracting away the local neighborhoods of "clusters" to reduce the complexity is certainly tempting. Another question arises: will the abstracted graph have, in itself, a small-world topology? Intuitively, if the graph of friendships from EU residents is considered, and an abridgment is made, it is likely to coincide with geographical locations – and when "friendship" between the resulting groups is reconsidered, it is again likely to match larger geographical demarcations.

This idea has been explored by Song et al. in [116], where a remarkable self-similarity was found in the successive coarsenings of certain graphs. Coarsenings were generated by replacing groups of strongly connected "boxes" with individual vertices with equivalent connections. Their experiments were performed on a very wide range of networks, including classical examples of social networks (movie actors), natural networks (protein interdependence), and the word-wide web. Auber and Chiricota [17] also point to this self-similarity as a basis for their hierarchical clustering approach, which will be covered in the next section.

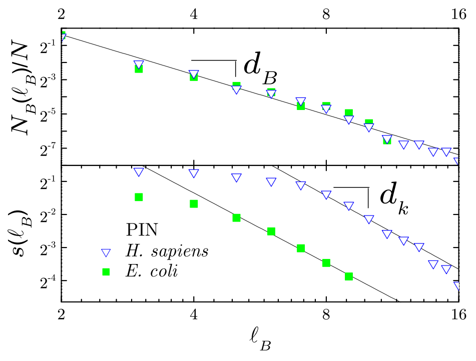
Figure 2.4 illustrates the results of the coarsening procedure in [116], by tiling the network into boxes of radii of at most l_B. The problem of doing this in a minimal number of boxes has been found to be NP-complete [20]; in this experiment, vertices were assigned to boxes with a simple pseudo-random heuristic, and the results suggest that self-similarity does not depend on an optimally efficient packing.

2.3 Graph Clustering

Clusters have been informally introduced earlier; however, there is an important difference between the "clustering coefficient", a metric, and the idea of *clustering* as it is used throughout the rest of this work. The existence of a high clustering coefficient indicates that, for "natural" complex graphs, clustering should be effective, since there are clusters to be found. This section deals with the task of finding them, and the



(a) World Wide Web and co-starring actors



(b) Protein interaction networks (PIN) from E. Coli and H. Sapiens

Figure 2.4: Self-similar natural networks, from[116]; Two power laws are seen to be in effect; in the top graphs, $N_B(l_B)/N \sim {l_B}^{d_B}$, indicating a self-similar network with fractal or "box" dimension d_B. In the lower graphs, the power law is $s(l_B) \sim {l_B}^{d_k}$, where $s < 1$ is the scaling of the edge degrees at each level.

formal definition of a clustering hierarchy.

Clustering seeks to find groupings of elements into subsets based on similarity between elements. Cluster analysis is also referred to as grouping, clumping, classification, and unsupervised pattern recognition. The goal is to find disjoint subsets, called clusters, such that two criteria are satisfied: homogeneity (two vertices in the same cluster should be closely related to each other) and separation (vertices in different clusters should present low similarity) [74].

Formally, given a graph $G = (V, E)$, a *clustering* is a set S of vertex subsets $\{S_1, \ldots, S_k\}$ which together cover the complete set of vertices, V. In the common case where no overlap is allowed between the clusters (empty pairwise intersections of S_i), the clustering is also a *partition*. That is, $S = \{S_1, \ldots, S_k : \bigcup_{i<k}^{i} S_i = V \ \wedge \ \bigcap_{i<k}^{i} S_i = \emptyset\}$.

In the optimization problem of graph partitioning, the goal is to minimize the number of inter-partition links in the graph (this is related to the minimal-cut problem). Multiple variations exist, such as finding the best partitioning with k partitions, or seeking answers where the size of partitions is kept constant. The problem is in general NP-hard [131], but of great interest in such fields as design packaging (where the number of connections between the components of a board must be minimized), or distributed systems (where the minimization of the number of messages crossing machine boundaries is critical). In these applications, clustering is frequently applied as a preprocessing step to reduce the number of nodes to be partitioned.

When presented with a graph, there are two main possibilities: to use only the graph's connectivity to generate the clustering (structure-based clustering), or to exploit the semantic relations present in the graph's content to improve the clustering (content-based clustering). The former strategy can be applied to all graphs, while the latter can provide better results in exchange for application-specific knowledge. For instance, [30] describes a specific algorithm for clustering documents within a hypertext space. Typical approaches are to refine a general-purpose algorithm with application-specific heuristics, or to include application-specific heuristics into a general-purpose algorithm.

Examples of general-purpose graph-clustering approaches include classical pattern recognition algorithms such as K-Means and Single-linkage, and more involved methods such as HCS (Highly Connected Subgraphs) [74] and MCL (Markov Cluster aLgorithm) [131].

Clustering to generate graph abstracts

When generating a clustering that is intended to be used as an abstract of the whole graph, the minimization of a cost function is only a means, and should not be mistaken with the actual goal. In many cases, a partitioning that is good from a graph-theoretical point of view (minimizing extra-cluster connections and maximizing intra-cluster ones) will also present a valuable insight into the graph structure and will make a good abstract. But this need not be so, specially if the semantics of the graph do not closely match its structure. For instance, not all vertices or edges may be equally important, or important edges may have been omitted from the graph.

If a graph has a set of "natural" (from a semantic point of view) clusters that should be located, several undesirable scenarios are possible:

- A cluster may be split up into several smaller clusters, losing the significance of the larger, "natural" cluster.
- A cluster may be joined together with several other unrelated clusters, forming a cluster that makes little semantical sense, that is, that fails to capture a "real" common characteristic.
- The number of clusters may be too high or two low – and in this case, one or both of the above problems is bound to occur.

Semantically motivated clusterings need not be unique; that is, there may be more than one semantic criteria according to which the graph can be clustered. Imagine a graph of biological species with edges whenever there is a "feeds-from" relationship; this graph could have natural clusters corresponding to the biological families to which each species belongs, or to the habitats in which they live, and depending on which is chosen, a "dolphin" would be classified either as a mammal (which would be largely land-based) or in the same cluster as other sea-based animals. General-purpose clustering algorithms, by themselves, clearly do not suffice for “correct” human-oriented abstract generation.

However, if the goal of clustering is only to ensure small-world and self-similarity properties of the graph (in order to guarantee that further clusterings would be possible), very simple approaches, such as the boxing algorithm described in [116], would suffice.

Interactive clustering

Some algorithms have tunable parameters, allowing the user to set this parameter at the point that produces the “best” results for the particular application. For instance, [17] presents an algorithm where k, the number of partitions, is a free parameter. By default, it is set to a point where it is near to the maximal value of the partition cost function, MQ, defined in eq. 2.3.

$$MQ(C;G) = \underbrace{\frac{1}{p}\sum_{i=1}^{n}(s(C_i,C_i))}_{intra-cluster} - \underbrace{\frac{1}{p(p-1)/2}\sum_{i<j} s(C_i,C_j)}_{inter-cluster} \tag{2.3}$$

The cost function MQ (eq. 2.3) reflects the desireability of a high intra-cluster similarity with simultaneous minimal inter-cluster similarity. In this function, the ratio $s(A,B)$ is used to quantify the clustering coefficient between two sets of vertices, A and B, as defined in eq. 2.4. In other words, $s(A,B)$ quantifies the ratio of all possible edges between vertices of sets A and B that are present in G. Note that this ratio is a generalization of the expression $edges(k_v)$ used in the calculation of the clustering coefficient (see section 2.2). In equation 2.1, $edges(k_v)$ could have been written as $s(k_v, k_v)$.

$$s(A,B) = \frac{|\{(u,v) \in E : (u \in A \wedge v \in B) \vee (u \in B \wedge v \in A)\}|}{|A| \cdot |B|} \tag{2.4}$$

However, there may be several points where k maximizes the value of MQ. By letting the user adjust this parameter, cluster size can be adjusted, providing a better fit to the application (in the case of [17], that of providing insight into computer code dependency graphs).

Graph grammars

A *graph grammar* is similar to a traditional string grammar. Productions match a right-hand side subgraph to a left-hand side subgraph (similar to type-0 unrestricted Chomsky grammars); simpler variants only allow single edges or vertices to be matched and substituted, but clustering via graph grammars requires entire subgraphs to be matched and mapped into single clusters. A review of graph grammar theory and applications can be found in [111].

Graph grammars can be useful to generate clusterings, and present an interesting framework in which to specify application-specific “clustering rules”. Additionally, graph grammars are simple to understand, since the right-hand side and the left-hand side can both be represented as graphs, and the matching process can be illustrated directly on the graph. Multiple graph querying languages rely on representing both queries and results as graphs [13, 44, 72, 73, 80, 87].

There are multiple formal specifications of graph grammars, with a corresponding variety of pattern-matching constructions. Typically, vertices and edges can be queried for their properties (each is assumed to contain a key-value map, and keys can be queried for), and both can be queried for their local graph structure (with operators such as `degree-of(v)` or `target-of(e)` resulting in a number and a vertex, respectively). Key-value maps for edges and vertices can also be used to include information on the “types” of edges and vertices. For instance, a `type` attribute may be defined for all graph elements, so that the statement `attribute-value(`$v_{dolphin}$`, ‘type’, ‘mammal’)` would return “true”.

The use of rules that are to be matched to a graph introduces the question of parsing alternatives: which rule should be executed if more than one can be applied at a given moment? Between two rules, r_1 and r_2 that are both applicable, several types of dependency may exist. It is possible that the outcome of applying any one of them still allows the other to be applied, and the end outcome after applying both is idempotent. It is also possible that each order of application, while possible, results in a different graph, so that r_1 and r_2 are non conmutative at a given point (note that r_1 and r_2 may be conmutative for one graph but not for another, since the parts of the graph that match may be different in each case). Finally, the application of one rule may bar the application of the other: the result of applying r_1 could render a graph that would not match r_2’s left-hand side.

Predictability of rule outcomes may therefore be desireable to yield consistent clusterings. The order of application can be enforced by assigning a priority to each rule, guaranteeing that if rules r_1 and r_2 can both be applied, then the match will be resolved in favor of the highest-priority one.

2.4 Hierarchical Clustering

In *hierarchical clustering*, the clustering process is repeated on the graph resulting from the initial clustering (the *quotient graph*) until a single cluster-vertex is left, the *root*. Since each vertex from the original graph will be contained in exactly one cluster-vertex of each successive quotient graph, recursive clustering results in an inclusion hierarchy whose leaves are the vertices in the original graph. This process is depicted in fig. 2.5.

The definition of a hierarchy on top of a tree can be formalized in several ways. A

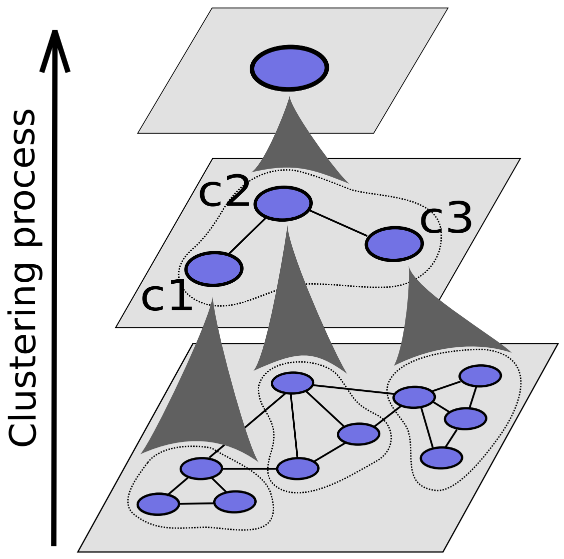


Figure 2.5: Hierarchical clustering

review of tree+graph structures can be found in [109]. In this work, we choose a simple interpretation found in [17, 38, 52, 82], where a graph $G = (E, V)$ is extended with a rooted tree $H = (N, B)$ such that $V \subset N$; namely, $V = \{n \in N : desc(n) = \emptyset\}$ – each vertex of G is represented by a leaf vertex in the hierarchy H. Non-leaf vertices of H represent *cluster vertices*, and at higher levels of the hierarchy, these cluster vertices will contain other cluster vertices (and possibly zero or more “leaf vertices”)..

A *clustered graph* is a graph with an associated clustering hierarchy, of which only a portion is currently visible. Formally, $G' = (G, H, S)$, where $G = (V, E)$ is the original graph, $H = (N, B)$ is a hierarchy defined over this graph, and S is the *slice* – a subset of the vertices of N that covers all the vertices in V. Therefore, $S = \{v_1, \ldots, v_k : v_i \in N \wedge \bigcup_{i<k}^{i} desc(v_i) = V\}$. Unless a slice S of a hierarchy H is entirely composed of the leaves of H, then it will also form a cut-set of H: the removal of the vertices in the slice would render H disconnected. The slice is used to determine the visible portion of the graph: abridgments of G can be generated from G' by varying the set of hierarchy vertices that are included in S. For a given slice S, the set of edges is defined as $E' = \{(u, v), u \neq v : \exists(x \in desc(u), y \in desc(v)) \in E\}$. Edges of E' that are not present in E are termed *induced edges*, and a single induced edge may represent more than one “normal” edge.

A stronger requirement on the composition of a slice S, not enforced by [52] or [82], is that there be no two vertices $u, v \in S : u \in desc(v)$. This condition is equivalent to that of forcing the cluster-vertices that form S to be non-overlapping, that is, $\bigcap_{i<k}^{i} desc(v_i) = \emptyset$. That is, the leaves of the trees rooted in each of the vertices of S form a partition of the entire graph G.

Creating a clustered graph $G' = (G, H, S)$ from a graph $G = (V, E)$ relies first on the creation of a hierarchy, $H = (N, B)$. The hierarchy can be initialized to $N = V$, and built incrementally by repeated application of the following operations:

`create(v)` v is added to N, with no children.

`add(v, u)` $u \in N$ is added to $children(v)$, and B is updated accordingly.

Since H is a hierarchy, there should be a single vertex $r \in N$, the root of the hierarchy, such that all other vertices of N are descendants of r. The slice can be initialized to any subset of N that fulfills the conditions for a clustered graph, but it is simplest to initialize it to r.

Operations on a Clustered Graph

The concept of a clustered graph is more powerful than simple graph partitioning, because the cluster hierarchy guides the operations of "zooming in" or "zooming out" in the degree of abstraction of parts of the graph. Intuitively, a currently visible vertex can be *collapsed* by subsuming it, together with all its siblings (as defined by the hierarchy), in a single cluster-vertex. The reciprocal operation is an *expansion*, where a the cluster-vertex is substituted for its descendants (again, as defined by the hierarchy). Formally, these operations can be defined as:

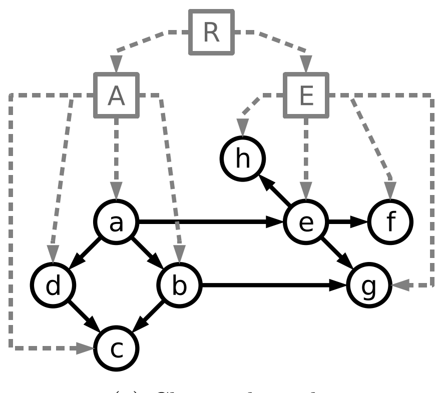
`collapse(v)` The slice is updated as $S' = S - desc(v) + v$, visible edges are recalculated accordingly. In other words, the descendants of v in the current slice S are substituted for v itself, and all edges with a vertex $u \in desc(v)$ as an endpoint will be subsumed into induced edges with v as the new endpoint. Note that edges with both endpoints within $desc(v)$ do not induce new edges, and are removed in the process.

`expand(v)` Vertex v, a vertex in the current slice S, is substituted for its immediate children: $S' = S + children(v) - v$. Visible edges are also updated, with induced edges that had v as an endpoint rebuilt to use one (or more) of the children of v instead; edges $(t, u) : t, u \in children(v)$ must be re-introduced.

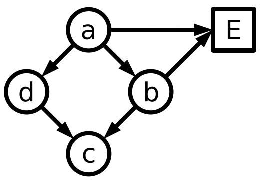
Refining Clusters

Graph abstracts hide multiple base vertices under higher-level cluster vertices. Therefore, finding a good label for this representative that can preserve the "information scent" [66] is an important goal; users browsing the graph should be able to predict, from this label and any other clues present in the interface, what can be found beneath the cluster vertex, and whether it will advance their current goal.

If the cluster hierarchy has been automatically generated, cluster-vertex labels must also be generated automatically, for instance selecting a single "most important vertex"



(a) Clustered graph



(b) A slice

Figure 2.6: On the left, the graph itself (solid lines) and the superimposed clustering tree (dashed lines). On the right, the clustered graph, with the "slice" S set to $\{a, b, c, d, E\}$.

from each cluster's children and using its label as the cluster's label. This is a crude approach, and much better results can be obtained if the user is willing to label each cluster, or even better, to adjust its contents. This requires additional operations:

`detach(v, u)` Cluster-vertex u, with $v = \text{parent}(u)$, is detached from its parent. It can now be `add`ed to any other cluster, retaining its own children.

`label(v, s)` Labels cluster-vertex v with a string s, substituting the automatically-assigned label with a better match to the semantics of the cluster vertex.

The `detach` operation can be used to delete a cluster u; once its children have been detached and added to parent(u), u can be safely removed, as it will no longer be referenced. `label` can be made more complex than a string, by referncing an icon or any other visual hint that could somehow symbolize the cluster's contents to a user.

The importance of making interaction with clustered graphs easy or intuitive is not a concern of graph theory or graph clustering themselves. Instead, it belongs to the field of human-computer interaction, which is introduced in the next chapter.

Chapter 3

Information Visualization and Human-Computer Interaction

The field of Computer Science that seeks to provide an effective interface between mind and computer is termed *Human Computer Interaction*, or HCI. The specific area of HCI that deals with the presentation of information is *Information Visualization* [70], or IV for short. Another definition is that of Chen [42]: "a computer aided process that aims to reveal insights into an abstract phenomenon by transforming abstract data into visual-spatial forms". The field of Information Visualization has expanded greatly with the advent of better graphic displays and computing capacity. An interested reader is pointed to [135] or the readings in [39].

Information Visualization is heavily related to Scientific Visualization. The major difference is that the latter focuses on data that reflects the physical world, such as weather forecast or medical image diagnosis, and therefore has a more intuitive mapping into a visual format. Information Visualization deals with abstract data, with non-obvious mappings to the real world. As such, it is also related to Information Retrieval, which is concerned with "the representation, storage, organization and access to information items" [22].

Certain presentations of information can be processed much more effectively than others. Tufte's series of books on the visual presentation of information [127, 128, 129] provides compelling examples. As a species, evolution has prepared us to respond to certain presentations of information much more efficiently than to others; we are more proficient dealing with visual representations than with abstract concepts [99]. Quoting R. W. Hamming, "the purpose of computing is insight – not numbers". We have an innate capability to filter out information according to "classes", and to fall for "outliers" that introduce noise, to react to colors and to movement[117], and to "see" fast sequences of still images as continuous motion. We build mental models of the world around us, but suffer from a limited buffer size.

The problem is not so much lack of information as an overabundance of data of which to make sense of. When the amount of information surpasses our capacity to interpret it, *information overload* is said to occur. By presenting information in a manner that is more suited to its comprehension, Information Visualization seeks to provide a higher "mental bandwidth" and hopefully avoid the problem of information overload.

Information representations need not be static. Indeed, one of the main characteristics of IV as compared to traditional "visual presentation" is animation and interactivity: a whole new axis involving time and user feedback. This does not directly increment the quantity of information that can be presented; instead, it allows information that is more *relevant* (to the user and task at hand) to be shown, at the expense of other details.

As an example, in the field of educational hypermedia, a representation likely to produce information overload could be a complete, unformatted listing of a course's structure, direct from the database. A better visualization of this data would hide irrelevant detail, while allowing a user to explore the course at various levels of abstraction, gain insight into its structure and that of the underlying set of concepts, and support the recognition of the pedagogical ideas that the course's original author sought to implement.

According to Chittaro[43], the goals of visualization can be briefly stated as follows:

- Allow users to explore data at different levels of abstraction
- Give a greater degree of understanding of the data
- Encourage the discovery of details and relations which would be difficult to notice otherwise
- Support the recognition of relevant patterns by exploiting the visual recognition capabilities of users

Shneiderman [115] presents the following mantra to achieve these goals:

Overview first, zoom and filter, and details on demand

The *overview* should present the larger picture and provide orientation for further browsing. It should help the user choose those areas that may be of interest, but at the same time present little or no detail. The user can then *zoom* in on those areas of interest, to examine them more closely. *Filtering* implies abridging or not presenting unnecessary details, so as to not incur in information overload. However, when required, expanded, full information should be available. This is what Schneiderman refers to as *details on demand*.

The remainder of this chapter concerns itself with Information Visualization and HCI as applied to graphs and graph-based interfaces. The next section provides an overview of the field of Graph Drawing (which would correspond to the "representation" step of fig. 3.1), and the last section deals with the visualization of graph representations (final step and feedback lines of fig. 3.1). Note that the distinction is somewhat artificial, as the application can perform changes in the representation (which may involve

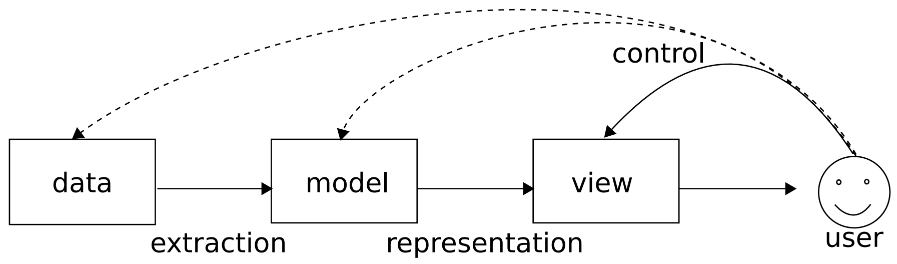


Figure 3.1: Information Visualization and Interaction. Data is first selected and transformed into a suitable abstract model (for instance, a graph). Then, it is totally or partially (filtering) represented (graph drawing). Finally, the user interacts with the representation (solid feedback line), or uses it to indirectly alter any part of the process, which may include the original data (dashed feedback lines).

graph drawing) in response to user interaction, and the extent and nature of these changes will necessarily depend on the graph drawing techniques used to generate the representations.

3.1 Graph Drawing

When visualizing and interacting with graphs, the starting point is to display the traditional $G = (E, V)$ mathematical construct on a surface or space. Performing this task automatically is the subject matter of *Graph Drawing*. The goal of this discipline can be stated in a deceivingly simple manner: given a graph, calculate the position of the vertices and the curves to be drawn for each edge. The resulting representation is frequently referred to as a *layout*.

Good graph layouts are powerful sources of information; humans are inherently good at pattern-matching, and a layout that emphasizes patterns within a graph will result much more informative than one that does not.

However simple the problem statement, the definition of a “good layout” is by no means straightforward [25]. Common criteria are preventing node overlap, avoiding edge crossings, keeping edge lengths constant, and preserving symmetry. Other criteria may involve drawing only orthogonal edges, or making all edges point in the same direction, or providing fast drawings for interactive use. In many cases, compromises must be found. For instance, many graphs cannot be represented without edge crossing. In this case, the number of crossing should be minimized. In other cases, conflicting goals may be found, such as a desire for few crossings and constant-length edges: in many cases, lowering the number of crossings involves complex edge routing. A classification of graph drawing methods can be found in [79].

Tree-drawing methods, as their name implies, can only be used to draw trees. Their main advantage is their speed (mostly linear with the number of nodes) and simplicity. A list of commonly used tree layouts, from [79], follows:

Reingold-Tilford classical top-down layout, preserving symmetry.

H-Tree uses orthogonal lines to create a less root-centered layout.

Radial the root is located at the center, and branches fan out in a radial fashion.

Balloon radial layout is performed recursively on each of the branches

Cone adding a third dimension to a balloon layout produces a “cone tree”.

Tree-maps uses nested boxes to represent branches. Extra information can be encoded in the area of the boxes, but the structure is harder to follow.

Planar representations are used when tree layouts are insufficient. The first question to ask is whether the graph can be drawn without edge crossings (a graph that passes this test is called *planar*) or not. If the graph is directed, it may also be desirable to draw all edges in the same direction (“upwards planarity”). When the graph does not directly fall into the intended categories, it is common to subdivide it into sub-graphs that do, layout these “layers” independently (there will then be no crossings within each layer), and finally join the layers together to form the drawing. The core of the complexity in this process is that of edge-crossing minimization between layers. In fact, it has been proven to be NP-hard [68]. Complex heuristics are used when subdividing the graph into suitable layers. One of the best-known layouts in this category is the Sugiyama algorithm, described in [123].

3.1.1 Force-Directed Layouts

A completely different approach to the layering strategy is that of *force-directed layouts*, or spring layouts. These layouts make use of a physical model, such as modeling vertices as ideal points and edges as springs that obey Hooke’s law (the original spring layout, proposed by Eades[51]). Starting from an initially random layout, attractive and repulsive forces in the model are repeatedly calculated, and gradient descent is used to search for an energy minimum. Different physical models result in layouts with different characteristics, and new aesthetical requirements can be reflected by carefully tuning the model.

The approach of Kamada and Kawai [85] is similar in that, it too, considers graph layout as an energy optimization problem. However, the only forces considered are those of the “springs” corresponding to each edge (there is no universal repulsion), and instead of calculating forces for every vertex on each iteration, only a single vertex is allowed to move. The optimal displacement can be calculated in $O(|V|)$.

In Eades' original model, the following attractive and repulsive forces were defined:

$$f_a(d) = k_a log(d)$$
$$f_r(d) = \frac{k}{d^2}$$

Where k is the "natural spring length" (the desired length of each edge), k_a is a constant used for attraction, and attractive forces were only applied between vertices directly connected by an edge. The use of $k_a \cdot log(d)$ instead of an ideal spring $(k_a \cdot d)$in the calculation of the attractive force is designed to prevent very long edges from exerting too great a force.

Reingold and Fruchterman [64] discuss the use of additional refinements, such as walls to constrain the rendering size, and propose another set of forces, which are less computationally intensive:

$$f_a(d) = \frac{d^2}{k}$$
$$f_r(d) = \frac{-k}{d}$$

Reingold's approach uses *simulated annealing* to control the layout process, constraining the maximum displacement of vertices in each iteration by a temperature value t, which is decreased with time. This results in large displacements during the initial iterations, and small refinements during the later stages. Different policies can be applied to the "cooling process" in an effort to obtain a better layout in less iterations.

Force-directed layouts share a set of common characteristics:

- Force-directed layouts are easy to implement and extend, and yield good results for a large family of graphs. They tend to display symmetries in the original graph and keep consistent edge lengths.

- A new layout is highly non-deterministic, since it is initialized with random positions and is the result of a high number of iterations where all vertices have influenced all others.

- Existing vertex positions can be easily reused, rendering incremental layouts, which can be substantially faster and of better quality than if a random layout was initially chosen. However, unless convergence is very fast, it is difficult to predict where any individual vertex will end.

- Complexity is $O(|V|^2)$ per iteration, since repulsive forces must be calculated for each pair of vertices, of which there are $(|V| \cdot (|V| - 1))/2$. If the graph is dense and/or the initial layout is bad, a large number of iterations (in the order of $|V|$) may be needed before a good drawing is generated.

- Layouts can be readily extended to 3 (or more) dimensions, by extending the set of dimensions that vertices are allowed to move in. Forces acting on vertices do not need to be modified.

Laying out large graphs

When the graph to layout is truly large, it is probably too expensive to generate a conventional layout that minimizes all edge crossings, and a common technique is to first lay out a spanning tree of the graph with a (fast) tree layout algorithm, and then add the remaining edges. The key here is to choose a good spanning tree so as to minimize crossing in the complete graph, and many algorithms have been developed to do just that, ranging from a depth-first search to a weight-based incremental construction starting from a carefully chosen “root” node [79].

A similar method is to select a “coarsening” of the original graph, perform layout with the coarsening, and then introduce the remaining vertices until the full graph is completely laid out. This approach, followed by Walshaw [134] has achieved dramatic speedups for force-directed methods in very large graphs. Again, the key is to use good coarsenings. Walshaw uses a recursive coarsening with two properties: the algorithm must be fast and efficient (the goal is to represent the whole graph, there is no requirement for intermediate graphs to make any sense) and clusters should be distributed uniformly. The chosen algorithm is to perform edge-matchings, that is, in each coarsening step, a maximal set of independent edges (sharing no endpoints among them) is sought and collapsed. Figure 3.2 illustrates the progress of the algorithm. First, a very coarse version is laid out; progressive iterations add more vertices, until the full graph has been drawn (fig. 3.2 *(e)*).

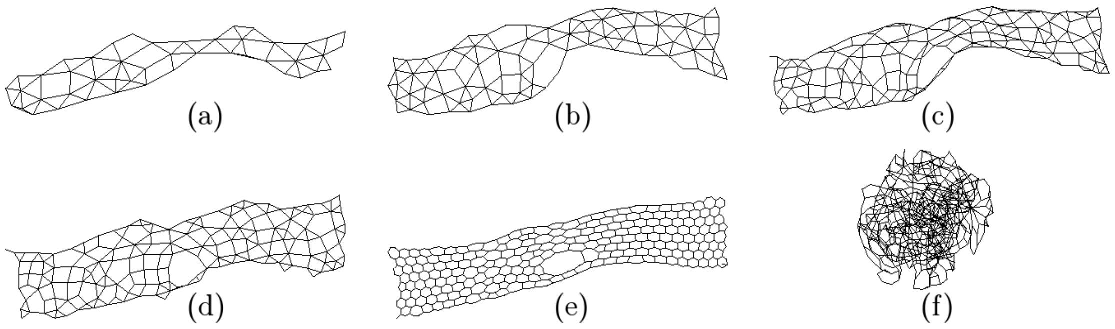


Figure 3.2: Use of graph coarsening to reduce the time to perform force-directed layout, from [134]. Layouts (a), (b), (c) and (d) correspond to coarsened versions of the same graph, which is shown fully laid out in (e). The last layout, (f), was generated in the same amount of time, but without coarsening.

3.2 Graph Visualization and Human-Computer Interaction

This section deals with the visualization of graphs and the interaction with such visualizations. Once layout has been performed, a representation (or *view*) of the graph is shown. The user will now want to perform tasks on top of this display, and this will require interaction.

Lee *et al.* [90] propose the following taxonomy of tasks for graph-related information visualization. Of course, not all tasks are equally important for all applications:

Topology-based tasks

Adjacency: Find the set of vertices connected to a given vertex, or the vertex with the most neighbors.

Accessibility: Find the set of vertices within a given radius (in edge hops) of another vertex.

Common connection: Given a set of vertices, locate all their common neighbors.

Connectivity: Locate the shortest path between two vertices, or identify connected components, articulation points or bridges; locate clusters.

Attribute-based tasks

On vertices: Find vertices with specific attributes; gain an overview of the contents of the list of vertices.

On links: Perform topology-based tasks, restricting the search to those edges that fulfill a certain conditions on their attributes.

Browsing tasks

Follow path: Follow a given path through the graph.

Revisit: Return to a previously visited vertex.

General tasks

Overview: Gain an overview of the general layout and connectivity of the graph.

In order to perform these tasks, a user must first be provided with a view that presents the necessary information. This requires the information to be visible and distinguishable. Problems may arise due to occlusion (parts of the graph's drawing hiding other parts), ambiguous renderings (more than one possible interpretation; for instance, edge intersection with very small inner angles) or not enough display space to represent the whole graph at once while retaining necessary details.

3-D Representations

While most representations are planar, many of them can be easily carried over into the third dimension: trees can be represented as conic trees, as a landscapes as in the Silicon Graphics Inc. (SGI) File System Navigator (included with their operating system until version 5), or as points on the surface of a sphere [79]. Many graph layout algorithms (such as force-directed layouts) can be readily extended to a third dimension, with the advantage that it is always possible to route edges to avoid undesired crossings (although that does not prevent the crossing to reappear when the view is rendered in 2D).

The third dimension gives, literally, more “space”, but at the expense of requiring the user to navigate the spatial representation to locate a suitable view. Due occlusion, certain vertices or edges will be visible only from certain viewpoints – and if a chosen viewpoint is inside the convex hull defined by the graph’s outermost vertices, parts of the graph are bound to fall “behind” the user’s point of view. The user is expected to navigate to discover the occluded portions, although it is frequent to use other mechanisms such as transparency or perspective transformations to simplify this task.

In addition, true 3D interfaces also present a wealth of navigational options, making them more complex (and unfamiliar) to navigate in than conventional 2D. Some measures can be taken to address this issue, such as providing an “ground” plane from which “up” and “down” can be defined. Additionally, traditional 2D pointing devices such as mice, touchscreens or trackpads are ill-suited for interaction in 3D spaces.

To avoid occlusion and difficulties associated with 3D interaction, many systems use a “2.5D” approach instead (such as the one depicted in fig. 3.3) by performing a 2D layout and superimposing additional information on the third dimension, before rendering the results in 3D.

Another option within 3D graph visualization interfaces is to force particular parts of the graph into 2D planes, and render these as a stack (particularly useful to display different interactions between similar systems). Graph clusters can also be allowed to have a 3D internal structure, while the graph composed of the clusters themselves is rendered in a 2D plane that bissects all clusters. These approaches have been explored in the WilmaScope graph visualization environment, described in [50].

3.2.1 Detail and Context

It does not make sense to try to represent a graph with thousands or even hundreds of nodes on a small area, such as a computer screen: the nodes will be so tiny and there will be so much edge clutter that it is naive to expect the user to make much of the drawing, except perhaps to understand the general outline and estimate the graph’s size.

Graphs share this problem with any other visual representations that are larger

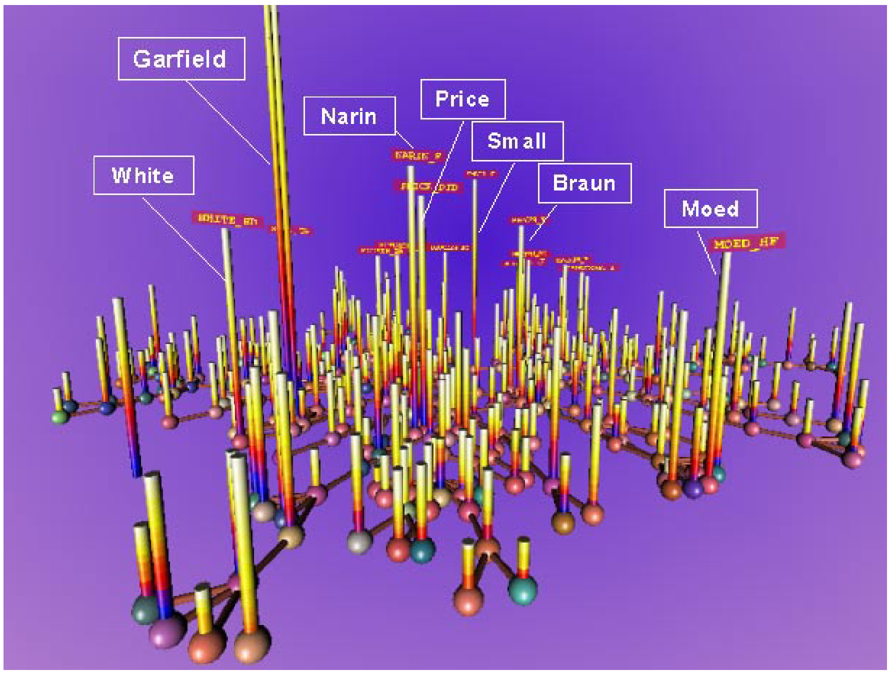


Figure 3.3: A 2.5D visualization of a scientific co-citation graph, from [29]. Vertical bars indicate the number and importance of the citations.

than the available space. Parts of the data must be hidden, or made smaller, to allow other parts to be examined in detail. When the user is seeking relations among this data, it will usually not do to simply hide some parts: some measure of context may be necessary to keep orientation and prevent loss of important relations.

In the case of graphs, context is more delicate than in other scenarios. A single edge can make an important difference, connecting several otherwise distant areas – or it may be almost irrelevant, if these areas were already well connected. Individual vertices can fulfill similar roles (as in the case of articulation points). Context, understood as "knowledge of important surroundings" is not only present in the connections between major graph areas - some parts of a graph will generally be more relevant than others; and the degree of relevance may vary depending on the current task. It is therefore difficult to determine what context should be supported for a given detail operation.

Graph detail can be lowered by several techniques. The first is of course to shrink a part of the layout; but this will likely result in a mess of edges and vertices and their labels. Edges can be omitted from low detail areas, although some (possibly a spanning tree of the area) may be preserved to show the connectedness. Labels from low-detail vertices can be removed, vertex shapes can be substituted by points, and multiple low-detail vertices can be merged into clusters, as is performed by [132].

Leung and Apperley [91] refer to techniques to accommodate both detail and context as *distortion-oriented*, and distinguish between distortion applied to the graphical representation (*graphical data*) or to the original, non-graphical data. In each case, the classification distinguishes whether a mixture of high and low detail content is present ("distortion") or not. An abridged version of their classification follows:

- *Graphical data* – a result of either inherently graphical data or a graphical abstraction of non-graphical data

  Distorted view: detail in context, via spatial transformation. This can be achieved via polyfocal projections and graphical fisheye lens.

  Non-distorted view: detail with little or no context, as in zooming and windowing (ie.: overview+detail).

- *Non-graphical data* – "distortion" in an information-density sense.

  Distorted view: abstracting and thresholding

  Non-distorted: paging and clipping

Note that it is perfectly possible to mix several techniques; for instance, graphical techniques such as zooming and windowing are often combined with non-graphical ones such as abstracting and thresholding.

Overview+Detail

A typical case of non-distorted view is the *overview+detail* setup, where a small overview window is used to represent the entire context and a large detail window to examine the fine detail (generally with a variable degree of zoom). It is typical to place a marker in the overview window signaling the portion that is currently being displayed in the detailed view. This approach is most useful when distances and angles found in the representation are meaningful, and is therefore typically found in image manipulation programs, CAD tools, or geographical information systems. These interfaces are also known as *pan & zoom*, in reference to the typical modes of interaction used to shift the area of detail and vary the degree of zooming.

Many user interface toolkits provide multiple-document support with panning, and even zooming, "for free". This has made overview+detail interfaces very widespread. Recently, fast hardware and application frameworks such as SHriMP [120] (1999) have enabled zoom-based interfaces to speed up dramatically, giving birth to *Zoomable User Interfaces*, or ZUIs [105]. The concept of ZUIs, however, also incorporates elements of thresholding and abstracting.

Focus+Context

Focus+Context interfaces incorporate context surrounding a detailed "focus" (distortion may be necessary to make the necessary space), and update the focus to follow the user's actions, in such a way that when part of the context is operated on, it becomes focused, while the old focus reverts back to a low degree of detail and merges into the surrounding context.

Fisheye views are a technique of distorting the data, not the representation proposed by Furnas in [65]. Information elements present in the display are assigned a *degree of interest* (DoI), and those with importance below a certain cutoff value are then abstracted. The DOI assigned to a given information element x given a current *point of interest* (PoI) y is illustrated in eq. 3.1: it is the result of considering the *a priori importance* (API) of x and its distance to the current point of interest, $\odot$. Refinements on this technique assign multiple types of DOI to each element, so that different aspects of the elements can be (un)highlighted independently.

$$\mathrm{DOI}_{\mathrm{fisheye}}(x|\odot = y) = \mathrm{API}(x) - D(x, y) \tag{3.1}$$

Graphical distortions reprocess an image to enlarge or reduce certain areas. Multiple areas can be smoothly enlarged, as in polyfocal projections, or only two levels of detail used, as in the bifocal display. Another option is the perspective wall, that follows the model of a flat wall with the context reduced as if by the effect of perspective. Polar coordinates can be used to create fisheye views [112], and finally hyperbolic spaces allow huge representations to be fit in a limited space by reducing detail exponentially fast at the borders (see fig.3.4).

The term *fisheye view* itself stems from the wide-angle lenses used in photography, and is also used in the context of visualization to refer to similar graphical distortions involving smooth changes in zoom level, such as hyperbolic trees. An example of such a graphical "fisheye" effect is presented in fig. 3.4 (from [89]). The figure depicts the process of focus-change in a hyperbolic tree. In the first snapshot, the original point-of-interest, labeled as A, is located at the center of the image. In the last snapshot of fig. 3.4, the new point-of-interest, labeled as B, has moved smoothly towards the center, while A and its children nodes have been shifted towards the low-detail periphery.

3.2.2 Preserving the mental map

While interacting with a graph, a user will gather and maintain an idea of the general structure of this graph and the general location and relations between parts of the whole: a "mental map" [97]. When the interface relies heavily on changing the viewpoint (as do all focus+context approaches), special care must be taken to prevent breaking the user's current mental map [97]. The degree of change depends heavily on whether the graph layout is computed only once, at the start of the visualization, or incrementally, after each navigational or editing step. This decision can be roughly assimilated to geometric vs. informational distortion.

The "fixed layout" approach is followed by [113], [132] and others. For instance,

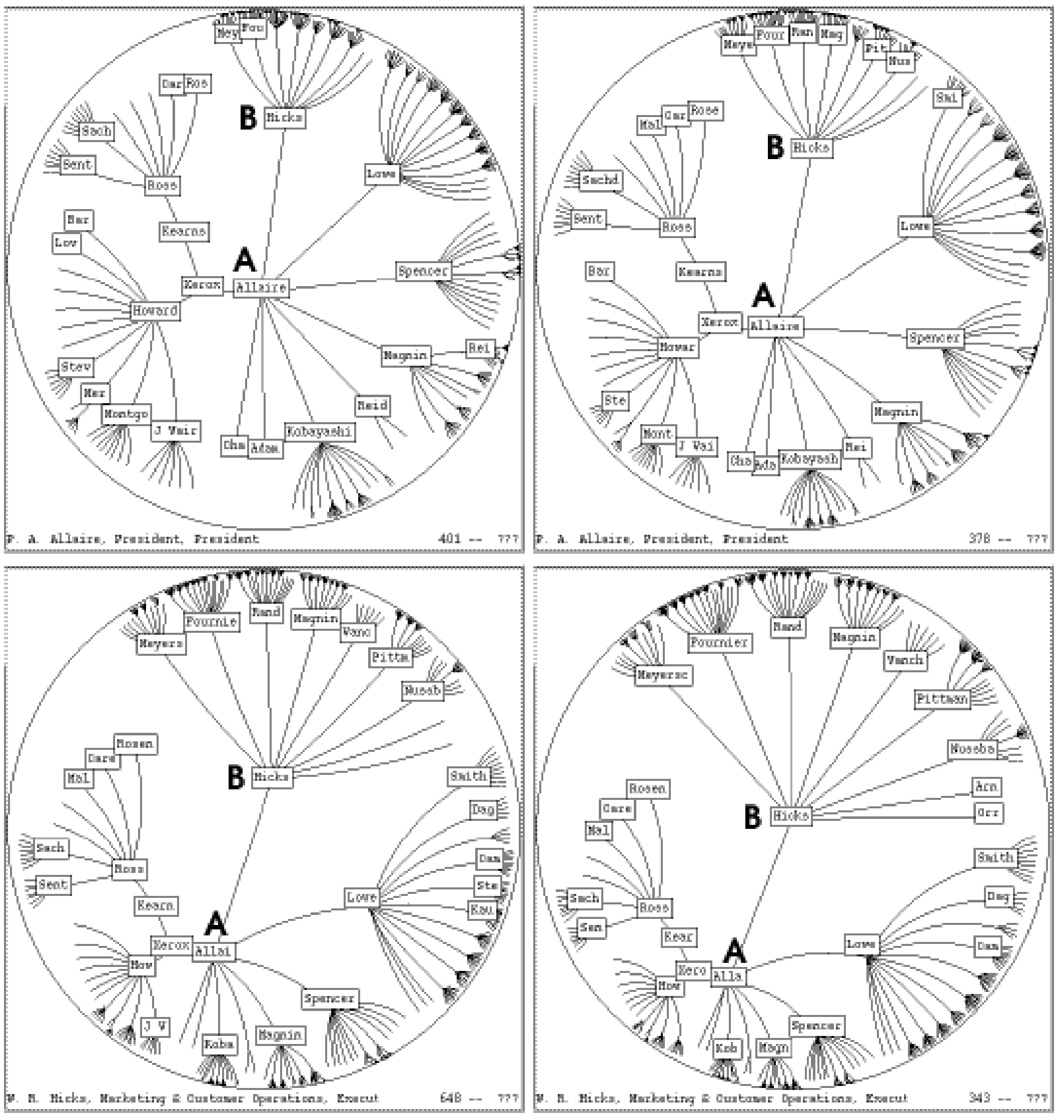

A
B
Allaire
Hicks
P. A. Allaire, President, President
W. R. Hicks, Marketing & Customer Operations, Execut

Figure 3.4: Change of focus in a hyperbolic tree, adapted from [89]. Images are ordered left to right, top to bottom. The initial focus is *A*, and the desired focus is *B*.

in [132], the whole layout is precomputed in advance, using a layout algorithm where proximity is closely related to cluster-belonging. When navigating, three areas are defined around the cursor, which behaves as a "magnifying lens": in the outermost area, clusters are represented as large vertices, at a fixed maximum level of abstraction; in the intermediate area, the degree of abstraction varies between this maximum level and full detail; finally, in the central area there is no abstraction, and variable geometric zoom is performed.

Fixed-graph approaches such as the above avoid recomputing layouts, because when substituting high-level clusters for single cluster vertices, these are simply located at the average position of the "leaf" vertices they represent. This minimizes change, and is an effective means for preserving user orientation. However, a single full layout cannot be used if the graph is being edited instead of simply navigated (adding or deleting edges from graphs can potentially alter the whole graph structure, and require a new clustering and a full relayout). A second problem is that generating a good layout for a large graph can require a long time, specially if the algorithm should output a result that is similar to the currently presented layout.

On the other hand, incremental layout approaches (mainly [52] and [63]) do not need to perform full layouts, make a better use of display space, and can accommodate interactive graph editing. However, because layout is slow, the number of displayed nodes must be relatively small to preserve interactive behavior. This favors semantical zooming over geometrical distortion: the former lowers displayed node count, and the latter is most useful when large amounts of nodes are visible.

A series of general "factors" that contribute to mental map preservation are identified in [60]. Although they are general enough to be relevant to any focus+context navigation schemes, this discussion will deal only with hierarchical graphs that undergo relayout after an edit or navigational action.

Predictability

A first factor is to make the jump from one view to the next predictable. The user should be aware of what changes are to be expected in the view before triggering them. This goal can be furthered by providing visual feedback of actions before they take place, educating the user as to the internal workings of navigational actions, and keeping these simple to understand and use. As a side effect, any strategy that increases predictability of navigational actions will also offer valuable insights about the graph structure. This can lead to a better mental model. However, excessive navigational aids can also lead to information overload.

Degree of change

A second factor is to minimize the jump itself. The new view will hide some nodes by collapsing them into higher-level clusters, and will expand new clusters into their components. Other sources of change include altering the underlying graph, or if the graph is being filtered before presentation, changes to the filtering.

Keeping a low degree of change between views depends heavily on the chosen incremental layout algorithm. Many factors determine the "goodness" of a layout. There is agreement on some criteria, but no officially accepted "goodness" metric - which would, in any case, depend heavily on the application. As discussed in section 3.1.1, force-directed layout algorithms produce good layouts on a large range of graphs, and are particularly well suited to incremental layout.

When minimizing change, a balance must be struck between keeping changes low and finding a "better" layout (which will probably differ from the current one because of the stochastic nature of force-directed layouts). This can be achieved during relayout or by transforming the resulting layout "a posteriori" into a version that is more similar to the original one. A posteriori transformations such as affine transforms (rotation, shear, symmetry) do not modify the "energy" of the layout, but can result in a better match to the previous one. However, weakly-connected subgraphs can undergo such transforms independent of each other, and detecting and reverting each of them is a difficult problem.

Traceability

The last factor is to evidence the changes as they take place, so that they can be tracked and integrated into the user's mental map. This is usually achieved via animation. A simple approach is to interpolate a series of frames between the old and the new view, and present them in quick succession. [62] presents a detailed study on the desireable characteristics of graph animations. Better animation results if the intermediate frames are individually correct graph drawings, avoiding such things as overlapping nodes and overlapping or easily confused edges, which may result in display of non-existing structures. Another important property is the "structure" of movements – a rotation of several nodes at once is much easier to follow than several independent movements. However, all this can be expensive to calculate, and animation should be perceived as continuous and smooth in order to be effective. If incremental layouts are performed and changes are kept small, the animation structuring techniques used in *Marey* [62] are probably overkill.

3.2.3 Creativity Support

As a part of better human-computer interaction guidelines, graph editing environments for knowledge engineering tasks should actively support and encourage creativity. Long development lifecycles from idea to testable prototype are counterproductive, since by the time an approach is proved wrong, too much effort has been invested to start all over again. To avoid this, according to [114], editors should provide

- Sketch capability – Partial efforts should yield partial results. If it is very expensive to test out an idea, fewer ideas will be tested.
- Support for learners and experts – The tool should be shallow to learners, allowing easy tasks to be performed easily; but with this should not preclude a deep end for advanced swimmers.
- Wide walls – Support should be implemented for very general primitives that can be combined in different contexts, yielding expectable results. "Artificial" constraints should be avoided wherever possible. For instance, the Undo/Redo

and Cut/Copy/Paste primitives can be reused in many different contexts. In graphs, this list could include ”New/Delete/Connect” to achieve completeness.

Some of these issues will be encountered in the following chapter, when analyzing the problems faced by adaptive hypermedia content authors.

Chapter 4

Adaptive Hypermedia

Hypermedia refers to the combination of contents of any type (any combination of video, audio, static images and text, generically called *media*) and links to other contents. It is a logical extension of the original hypertext concept, with textual documents linked to each other enabling a non-linear reading. Hypermedia spaces are natural graphs, where individual pieces of media can be considered vertices and the links between them constitute the edges.

The field of *Adaptive Hypermedia* (AH) strives to develop hypermedia tailored to each user's needs and preferences. In a traditional hypermedia environment all users are presented with the same media fragments and offered the same links to navigate them. However, not all users share the same goals and preferences, or start out with the same knowledge. Adaptive hypermedia changes the contents, their presentation or their navigational organization based on information about the user as collected by the system. The information that allows the tailoring to take place is called *user model* (UM), and the process of tailoring is termed *adaptation*.

Adaptivity, Adaptability and Transparency

Opperman *et al* [102] distinguish between *adaptivity* and *adaptability* - the first is system-initiated, while the second is requested explicitly by the user. For example, a screen where the user could alter preferences in a straightforward manner would fall under *adaptability*. If the same actions where taken by the system after inferring the preferences from the user's behavior, that would be *adaptivity*. Since many adaptive behaviors do not fall cleanly into any of these categories (for instance, a questionnaire where the answers are indirectly translated into different presentations, or a system where the user has a measure of control over the adaptations suggested by the system), this distinction will not be used in the remainder of the work. Adaptation is taken to mean a mixture of both, excluding only the most simple forms of adaptability (such as a menu that allows the user to change the application's background picture).

Before losing sight of this distinction, it must be noted that adaptability without adaptivity is likely to be frustrating for users. Systems that perform adaptation should be capable of explaining the decisions that they take on the user's behalf. In an ideal scenario, where the "correct adaptation" was always chosen over any alternatives, users probably would not mind. But since performing the correct adaptations based on incomplete data is bound to result in occasional mistakes, adaptive systems should always provide some breadcrumb trail that allows users to check and/or correct wrong steps of the process. This important property has been termed *scrutability* [86] or *transparency* [81].

4.1 Classification of AH Systems

This overview of Adaptive Hypermedia starts out pointing the types of adaptation that can be performed. Then, the main types of AH systems are briefly enumerated. Finally, considerations on the nature of documents in a hypermedia space are offered.

4.1.1 Types of Adaptation

What types of adaptation can be performed in adaptive hypermedia systems? A well-known taxonomy is that by Brusilovsky [34], revisited in [35]. According to this taxonomy, two main types of adaptivity can be distinguished. Adaptive presentation covers changes in the way a given piece of information is presented, while adaptive navigation support focuses on the hypermedia links that provide the structure. The term *structure* is used frequently throughout the present work to refer to the connections that are available for navigation.

Adaptive Presentation Techniques

- *Adaptive text presentation* covers text adaptation, and can be further subdivided into
  - *natural language processing* of textual content.
  - *canned text*, such as selection of alternative texts, changes in text detail (*stretchtext*), conditional text inclusion or "dimming", and changes in text block order.
- *Adaptive multimedia presentation*, where non-text media such as images, sounds or animations are modified to suit the user.
- *Adaptation of modality*, choosing the best media type (or combination thereof) for a given concept.

Adaptive Navigation Support Techniques

- *Direct guidance*, where the system recommends the next step to take based on the user's profile
- *Adaptive link sorting*, where lists of links are sorted according to their predicted degree of interest for the current user.
- *Link hiding*, which covers elimination, disabling and hiding of links not considered relevant at a given moment.
- *Link annotation*, providing a short text to illustrate the link's relevance to current context. Color-coding is also included in this category.
- *Link generation* to potentially interesting resources, on-the-fly. For instance, references to encyclopedia entries or web searches would fit into this category.
- In *Map adaptation*, an overview map is provided, with the map's structure reflecting the adaptations performed to the system's link structure based on the user's profile.

As with many taxonomies, some categories overlap, and adaptation techniques that fall into several categories are frequent. For instance, the [46] system could display "uninteresting" links in a font that closely matched surrounding non-link text; this could be considered both annotation (if the user does notice that it is, in fact, a non-recommended link) and hiding (because the link has been disguised).

4.1.2 Adaptation methods

Adaptation techniques are only means to a goal, the goal being to provide a better fit to the user's needs than would be possible with a non-adaptive version. In Brusilovsky's classification, adaptation techniques are only the practical implementation of higher-level *methods* that respond to the real adaptation needs. Methods can again be grouped as either presentation- or navigation-related.

Presentation-related Adaptation Methods

- *Additional explanations* for certain topics, for instance to expand a topic for which the user lacks background, or to provide extended information to advanced users.
- *Prerequisite explanations and comparative explanations* relate the current topic with prerequisite or otherwise related ones. Prerequisite explanations can fill in gaps arising from multiple possible paths through the hypermedia space.

- *Explanation variants* can be employed when it is not possible to use a simple modification (such as prepending a small 'glue' fragment) to achieve the desired adaptation, and a totally different approach is deemed more adequate.

Navigation-related Adaptation Methods

- *Global guidance* recommends, at each point, a series of steps that will lead users along the path that "best" matches their knowledge and profile. A common implementation makes use of a dynamic "next" button. Pressing this button brings a user to the next step in the path.
- *Local guidance* is less ambitious, using only the local neighborhood to decide the "next" step. If global guidance is available, this is not necessary.
- *Local orientation* helps the user make an informed decision in each point, by annotating the available links, hiding irrelevant ones, etc.
- *Global orientation support* informs users of their absolute position within the course, lessening the "lost in hyperspace" syndrome and helping them to make better long-range decisions on the path to follow.

Presentational and navigational adaptivity are complementary; for example, if Susan and Angela are both using the same system, and one is recommended a different path from another (navigation-related adaptation), it would only make sense to present individual contents (presentation-related adaptation) according to the paths that have been followed, and the data acquired by the system during past interaction.

Although both presentation- and navigation-related adaptivity present interesting challenges, this work deals mostly with the latter. Without navigation- related adaptation, there is a single link structure that is presented to all users – and complex graph visualization, though useful, is not so necessary. The use of global navigational adaptation, in contrast, requires multiple link structures to be created and maintained. Graph visualization is very useful for these tasks. Since possible navigation paths will now depend on link structure that the user is following, concepts can now change presentation order, requiring content authors to ensure that such a viewing order is indeed supported by their present adaptation. Global navigation support therefore significantly raises the cognitive burden, not only for those in charge of creating and maintaining the link structure, but also for content authors.

4.1.3 Adaptive Hypermedia Systems

This section provides an overview of the main types of Adaptive Hypermedia systems that can be found in the real world. The classification is not clear-cut, but it is roughly based on Brusilovsky [35], so it is more or less standard.

Much of the research on Adaptive Hypermedia has centered on its use in education: the possibility of bringing adaptation to course material is certainly tempting, and many existing textbooks already seek to adapt themselves to their users by providing a "reading guide" as a part of the preface (eg.: *if you are interested in such-and-such, skip chapter 8 but make sure to read chapter 9*). Also, since a large part of the AH research community is composed of faculty members, it is natural to focus on the domain they are most experienced with: education itself. Education also has certain advantages over other domains for research purposes: it is easy to evaluate outcomes with exams and questionnaires. The following is a non-exhaustive list, in alphabetical order: AHA [46], DCG+GTE [133], ELM-ART [37], Hyperbook [78], Interbook [53], Tangow[41]. Many of these systems have gone through multiple revisions; only a single version is listed for each.

Other important domains, according to [35] are *online information systems*, *online help systems*, *information retrieval* systems, *institutional hypermedia* and *personalized views*. In these systems, the goal is not so much to "educate" the user as to assist the information gathering process. The differences between these categories can be found in the domains where they are expected to be deployed. For instance, technical documents intended for general browsing could be structured into an AH online information system, whereas application-specific help would constitute an online help system, and documents describing an institution would conform an institutional hypermedia system. In the last example, faculty could be offered different views and links than enrolled students or prospective postdoc researchers.

In very large hypermedia spaces such as the world-wide web, the task of structuring information for efficient retrieval via hyperlinks is prohibitive. In an information retrieval system, there is no fixed link structure that encompasses the hypermedia space; instead, each content fragment contains metadata (or can be mined for this metadata on-demand) which allows it to be classified as "relevant" for a given set of queries. Mixed approaches with both explicit and implicit ("results of a query") links are also possible, and have been demonstrated in systems such as Welkin[14] and SCARCE[69].

A distinction can also be made between *open-corpus* and *closed-corpus* systems, where the former are designed to deal with very large result sets, and the latter are only required to perform on a restricted set of chosen documents. Breadth and extensibility favor open corpus, at the expense of predictability of results, since there is no guarantee that the system will prove capable of choosing the most relevant contents in response to a query, or indeed that any such contents will be available in the open corpus. In a closed-corpus system contents are usually carefully classified and revised to avoid these failure modes. As examples, Welkin uses an open corpus (google searches) as a complementary information source, while SCARCE is entirely driven by closed-corpus queries (structural "links" are actually queries over a well-annotated corpus).

It is also possible to perform adaptation in *information retrieval* tasks; compared to non-adaptive information retrieval, the difference is that the system fills in certain query parameters for the user instead of having the user enter these parameters manually (if such an option where available). For instance, the popular web search engine "Google" offers *Personalized Search*[3] , which weighs the results for a query taking into account past queries and the user's selection of results from those queries.

The last category, *personalized views*, refers to views that retain only a subsets of the total amount of available information. A non-adaptive offline example would be researchers that keep themselves informed by means of periodic bibliographical queries on certain keywords. An online example, which is also not adaptive because the "user model" (the set of feeds) must be explicitly maintained by the user, is the use of RSS[9], ATOM[32] or other syndication mechanisms to gather updated information on current topics of interest. Existing AH systems in this category include PowerBookmarks[92] and other adaptive bookmarking systems.

4.2 Reuse, Metadata and Domain Representation

Adaptive content provides added flexibility, since it is designed to adapt to different users. Adaptation is generally designed only for the system for which each content fragment was designed; however, once created, the potential for reuse in other systems is much greater than for non-adaptive contents. And reuse is of course very interesting, since the cost in time and labor of developing adaptive contents is greater than for non-adaptive ones (see section 4.4 on authoring) – and in both cases, it is certainly an important factor when developing any system.

An important question is what granularity of reuse makes sense. The computer-assisted learning field has coined the term *learning object* (LO) to describe a reusable e-learning course fragment (not necessarily adaptive). Multiple initiatives seek to standardize and promote LO interoperability; some efforts include the Learning Technologies Standarts Committee (LTSC) at the IEEE[4], the European-sponsored ARIADNE project[2], the Instructional Management System (IMS)[5] initiative, or the MERLOT learning object repository [7].

Different incompatible definitions of the idea of "Learning Object" can be found in each of these sources. Some include sequencing, others don't; some allow their LOs to have complex internal organization (even referring to or including additional LOs in a nested structure) or associated logic (which would be needed for AH), while others have no support for anything other than a simple presentation. Choosing the lowest common denominator leads to a definition of learning object that can include almost anything: "learning resources which can be reused". To avoid further confusion, reusable, adaptive content fragments will be referred to as *modules* in the remainder of this section.

It must be noted that reusing modules requires the importing system to be able to display the reused module. If a system A wants to reuse a module developed for B, there is an alternative approach to displaying it locally: request B to perform this service, through a well-defined protocol, and return the results. This is the approach followed by the Advanced Distributed Learning (ADL) initiative with its Sharable Content Object Reference Model (SCORM) [10].

Metadata and Retrieval

Whatever the mechanism followed to reuse an existing module, the first step is to find it. Retrieving a module from a repository requires some kind of description of the module being sought; this description is then matched against the data on all available modules in an effort to locate the best matches. Results from the field of *information retrieval* (IR) [22] are applicable to this problem.

Matching a textual query against a document's contents is only possible if the document being sought is textual in nature, or if a plain-text description has been created for this purpose. Furthermore, synonyms and other nuisances of natural language make full-text matching imprecise unless the exact same wording is used in the query and all relevant documents. A careful classification using a well-defined, constrained vocabulary allows much higher recall and precision. The classification data is then added to each document as *metadata* – data about data. While the use of such vocabularies within a single repository is certainly beneficial, users wishing to query the repository must be familiar with its classification scheme; this is a burden for local users, and makes interoperation with external repositories using a different classification scheme very difficult. This motivates the need to standardize metadata and its representation; for instance, each of the Learning Object initiatives enumerated above defines its own set of metadata.

Standardized metadata such as the well-known Dublin Core [138] has proven useful to allow interoperation of general content repositories. It provides support for many common fields, such as `title`, `subject`, `author`, `rights`, `format` and so on, with standardized representations for each. However, it is a lowest-common-denominator approach; higher-level descriptions are necessary, but they are bound to be domain-specific. A decentralized approach requires interoperable, machine processable metadata vocabularies with well-defined terms and relationships between terms. These 'shared conceptualizations' have been called *ontologies*, borrowing the term from philosophy. At the lower levels, technologies such as XML and RDF provide a common, machine-readable format in which to express ontologies. At a higher level, languages such as OWL, DAML and F-Logic allow designers to express complex relationships defined on classes and instances, and provide support for limited reasoning.

These technologies can be pursued to varying extents. Interoperation of metadata vocabularies can be very simple, as with *folksonomies* (flexible, informal ontologies built ad-hoc). On the other extreme, there is the vision of the *semantic web* [27], where software 'agents' can retrieve data from the web and interoperate with each other, opening internet to a host of exciting applications.

However, ontologies are complex to work with: the network of classes, relationships and instances can grow to a considerable size, and is difficult to visualize without adequate tool support. Because of their network-like nature, ontologies are commonly represented as graphs, and most authoring tools include a graph-based visualization. Perhaps the best-known one is JAMBALAYA [119], a visualization plugin for the PROTÉGÉ [100] ontology editor. Indeed, ontology diagrams are proving so useful that the Object Management Group, known for the popular UML specification, is working on a specification to standardize ontology representation [126].

A good visualization of the metadata vocabulary is not important only for those actively involved in the vocabulary's development; it is important for any user, since users attempting to label or query a document's labels also require good working knowledge of the vocabulary to use. We are currently unaware of any graph-based annotation/querying support.

Ontologies and Domain Models

Knowledge representation is not only useful in the context of metadata and reuse. Formalisms such as *Concept Maps* [16] or *Topic Maps* [48] have been used to define the conceptual structure of AH spaces. These conceptual structures are often referred to as *domain models*. Beyond their use to relate each content with the concepts it belongs to, these conceptualizations can convey prerequisites, examples, and a wealth of other relationships, which can be given additional semantics by the AH system.

4.3 User modelling

Adaptive Hypermedia is based on the possibility of adapting a system to meet the user's preferences and needs. However, these needs and preferences must first be obtained and represented. This section deals briefly with the contents, representation and update of *user models* in AH.

User models (UMs) are usually organized into several different categories; User preferences can be divided into *dynamic* and *static*, where dynamic ones change frequently over time, such as current goal and accumulated domain knowledge, and static ones are more stable, such as age, gender, geographical location, preferred language or perceived personality. Exactly how the model is represented and what parts are used to alter hypermedia presentation is highly system-dependant.

User models can start out blank and be filled during interaction. Alternatively, the user can be requested to select among a series of stereotypes, or fill in an initial questionnaire to prime the model. Of course, care must be taken not to make the adaptive system more cumbersome than a non-adaptive one, since the goal of adaptive hypermedia is to provide a better user experience, and this is more likely to be the case in an unobtrusive system than in one that requires heavy user intervention.

Representing User Models

This organization is reflected in the representation of the UM. A key-value pair can be used for most straightforward pieces of information; hierarchies may be applied to more complex types. For instance, when using the Felder-Silverman [55] model to represent learning styles, values for five dimensions are characterized. It would make sense to use a "learning-style" key with subkeys labeled "sensory-intuitive", "visual-verbal", "inductive-deductive" and so on, instead of five independent keys.

An important part of the UM will be dedicated to data gathered about the user's accumulated knowledge of the domain model. Since the domain itself will be structured as a network, the part of the UM that describes its coverage can be seen as an overlay on top of this network, specifying for each vertex the degree to which it is known, or is available to be presented, or is otherwise relevant to this user. Many AH systems use overlays as an integral part of their user models.

Systems that allow users to examine their UMs represent them as hierarchically organized key-value pairs, which are easy to generate. However, given that a large part of a user model is comprised of an overlay on a network, it may be much more interesting to represent the network instead. This approach would have to overcome the same limitations as representing a conceptual network or an AH system's navigational structure – with added technical difficulties, since the delivery mechanism of choice is the Web, and representing rich graphs on standard web browsers is currently not simple. Online, interactive graph applications are only starting to be developed.

Updating the User Model

User models can be updated in a variety of ways, and the choice of update scheme depends on the needs of each particular system. In an educational environment, for instance, a test may be required to mark a given concept as "well-known". In an adaptive help system, the user's indication that such a concept was known would be enough for the system to update its model accordingly.

All UM updates are subject to a degree of noise and inaccuracy. These can have multiple causes; if the system is web-based, there is usually no way to distinguish between session discontinuation and a network problem. Users will probably not grant their full attention to the adaptive system, multitasking instead among several open applications; this makes it difficult to estimate such simple parameters as the amount of time a user has spent on a page. In an educational setting, users may communicate via instant messaging or other technologies, which may taint test results. All these pitfalls must be thought of in advance by the system designers, which should also safeguard the user's privacy in their effort to deliver adaptive contents

Updates to the UM can be triggered in a variety of ways: by simply visiting a certain document, by demonstrating a degree of understanding, or by explicitly requesting the change. However, updates are not usually of a local nature: knowledge of a subconcept should somehow increase knowledge of higher-level ones, increased preference for one option in a mutually exclusive set should decrease the value of the other options, and so on. These cascaded updates can be constructed in different ways:

- Explicit rules – rules and associated triggers are defined. Triggered rules then update the user model. This allows fine-grained control over the update process, but may be cumbersome to author and maintain. This is the approach followed in the first versions of AHA [46].

- Implicit domain-model rules – the domain model may be used as a template to guide the update process. For instance, if the domain model defines a prerequisite relationship, and one such prerequisite is fulfilled, previously-locked concepts could be marked as "available". Fine-tuning could be performed via explicit overrides of this process, with the same benefits and pitfalls of using rules, but a more restricted firing condition: the customized overrides would depend on the same condition required to trigger the default behavior. This is the approach followed in WOTAN[57].

- Belief propagation over the domain model – this provides a more powerful approach than simple if-then rules, by leveraging knowledge engineering concepts such as Bayesian networks. *Bayesian Networks* (BNs) use conditional probabilities defined over the concepts of the domain model that describe, for instance, how familiarity with "quicksort" (a sorting algorithm) is believed to be influenced by familiarity with "sorting algorithms" in general. The KBS [78] system uses BNs to update its user model.

- Adaptation grammar approaches – systems that use high-level adaptation grammars, such as MOT with the LAG adaptation grammar [118], can define overall strategies that can be later compiled into specific, local rules. This approach is comparable to defining multiple transformations to be applied on the initial concept network.

Regardless of how updates are performed, it is important for adaptive hypermedia authors to be able to examine how the system is working, and if possible, provide a measure of transparency to the system's users. A good representation of the system's inner workings can provide a more accessible model, and allow system maintainers to monitor user's models during interaction in order to detect and fix possible undesired behavior.

Bear in mind that the user model determines, for each user, the output that should be generated to each request. The heart of an AH system is located in the update of the user model; adapting the presentation from the resulting UM can be seen as just another part of this update.

4.4 Authoring Adaptive Hypermedia

Authoring adaptive hypermedia is one of the main hurdles in AH adoption. Despite its many advantages over traditional hypermedia, few AH systems have been widely deployed. One of the reasons, perhaps the main one, is the fact that authoring adaptivity is currently much more complex than doing without. This section is written with Adaptive Educational Hypermedia in mind, since this is the field that the author is most familiar with; however, results should be readily extendable to any other type of AH.

When authoring traditional hypermedia, only one structure must be created and maintained: the navigational structure, defined by hyperlinks among the contents. Even this structure is non-trivial to create, and authoring tools and advanced *content management systems* (CMSs) allow authors to abstract from the low-level tasks and concentrate on the more knowledge-intensive aspects. Whatever the system, the following steps will have to be taken (adapted from [36]):

- Design and structure the hyperspace – Decide the rough contents of the hypermedia space.
- Create page content – Add content to each page.
- Define links between pages – Include hyperlinks to create the navigational structure defined during the first step.

In a sense, adaptive hypermedia systems can be considered CMSs, since they provide simplified frontends for content creation and access (while tracking users and performing updates to their UMs). However, where the simple databases of content management systems suffice, AH systems have to deal with a much larger array of models, which must first be authored. Again according to [36], authoring AH requires the following additional tasks:

- Design of the domain model – Decide how the domain will be modelled, and create a structure of that model that is both general enough to describe the initial contents that will be added and concrete enough to begin adding content. The domain structure is more general and coarse-grained than the "hyperspace structure" that the user will see; it does not contain contents, it describes what role a content can fulfill.

- Design of the user model – A description of the user-model characteristics that will be considered, and at least a preliminary idea of how they will affect what parts of the hypermedia space are made available and how they will be presented.
- Categorize contents – Added contents must be linked up to their role as defined in the domain structure, and care must be taken to ensure that they meet the goals set forth in the user model and adaptation logic.
- Adaptation and knowledge update – Define how the user model will affect the hypermedia space, and how data gathered from the user's activities in the hypermedia space should be interpreted as updates to the user model.

Not only does authoring AH require a greater number of more complex steps than developing traditional hypermedia; there is also a very strong coupling between tasks. For example, changes to a section of a traditional website are generally localized to that section, perhaps requiring minor updates to the main navigation tree. In an adaptive hypermedia scenario, modifications to the domain model will inevitably have effects on user model overlays (which are, by definition, reflections of the domain model), and therefore adaptation and knowledge update will also require updates.

An incremental, iterative process would be best suited to AH authoring, since there is a fair amount of trial and error involved in creating complex hypermedia spaces that can be tailored to different users – but this requires authoring tools which can span several tasks, and assist in dependency tracking between each of them. In the absense of such authoring tools, adaptive hypermedia authors are restricted to a cascade-like development process, where each step must be carefully considered before proceeding with the next. This contrasts with the creativity support guidelines provided in section 3.2.3.

Graph-Based Authoring Tools

An important decision when designing an interface for a knowledge-intensive tool is the type of visualization to use. Although almost any interface would be sufficient to simply enter model data, this would require authors to have advance knowledge of each model before entering it, placing too heavy a burden on authors. Adaptive hypermedia authoring is a creative task, and requires visualization support to assist authors in the analysis and understanding of each of the models and their relationships. Without good visualization support, it is naïve to expect non-expert authors to create rich adaptive hypermedia courses. However, most systems provide only textual lists or hierarchies of the main models, with field-based editing of individual aspects such as adaptation for a given domain concept.

Each of the component models that make up an AH space can be adequately represented as a graph. Concept maps, ontologies and other network-like knowledge representation formalisms used to develop domain models are already graphs; most user models dedicate a large space to an overlay of the domain model, and the link structures which will be presented to end users will, as in all hypermedia, reflect a graph. Indees, in a review of hypermedia modelling frameworks by Dolog and Bieliková [124], all of the proposed models could be depicted as graphs, both in the application domain axis (which described contents and their relationships) and in the navigation domain (which described how they were to be navigated). The most popular descriptions were UML-like Class Diagrams, Entity-Relationship Descriptions, or variants thereof. This must be taken with a grain of salt, because most AH systems are not built on top of explicit modelling frameworks.

The need for graph visualization arises in different aspects. First, adaptive hypermedia authors should be aware of the possible paths that may be followed through the system; authors should make sure that the content "makes sense" from that particular point of view, or be ready to prepare extra "glue" contents that will smoothen transitions. Another issue is that not only is representation in the authoring phase poor, visualization during testing is arguably in the same state: maintainers must often confront large user log files in an effort to understand what is right or wrong.

To quote a chapter on AH autoring written by Brusilovsky in 2002 [36]:

> The first graphic authoring tool that we will see really soon is a graphic editor of concept networks. With this editor the designer will be able to design and author connections between concepts just by placing concepts on a working window and connecting with a drag-and-drop interface. This tool will support both the design and authoring stages. A number of similar network editors exist currently in the field of concept-mapping and business presentation design, but all these tools are useful only with relatively small maps (under 50 nodes) and the main result of the work is the visual presentation of the network. As a result, these tools now can only be used during the design stage and for developing small networks. Future concept network editing tools will be very similar, but their main product will be an internal representation of the network in a database or XML markup form. Gradually, they will also handle networks of 100 and more concepts. After concept editors we may expect the appearance of similar graphic tools for structuring the hyperspace and for indexing pages or fragments with domain model concepts.

The Atlas tool [93] was an early realization of a graphical authoring tool for Tangow [41] AH courses. Additionally, Atlas could simulate the effect of a given user model in a course, and show only the parts of the course that would be presented to that user. A sophisticated filtering system was available to determine user model characteristics, allowing the "user model template" to be easily established; this type of filtering only used static properties, and did not take into account attributes that could vary during course performance. In spite of these features, Atlas proved unwieldy as courses grew in size, since only panning was available: there was no automatic layout, and zooming was not supported. Even if these features had been present, authoring large courses would still be difficult, due to the particular detail and context issues presented by large graphs (see section 3.2.1.

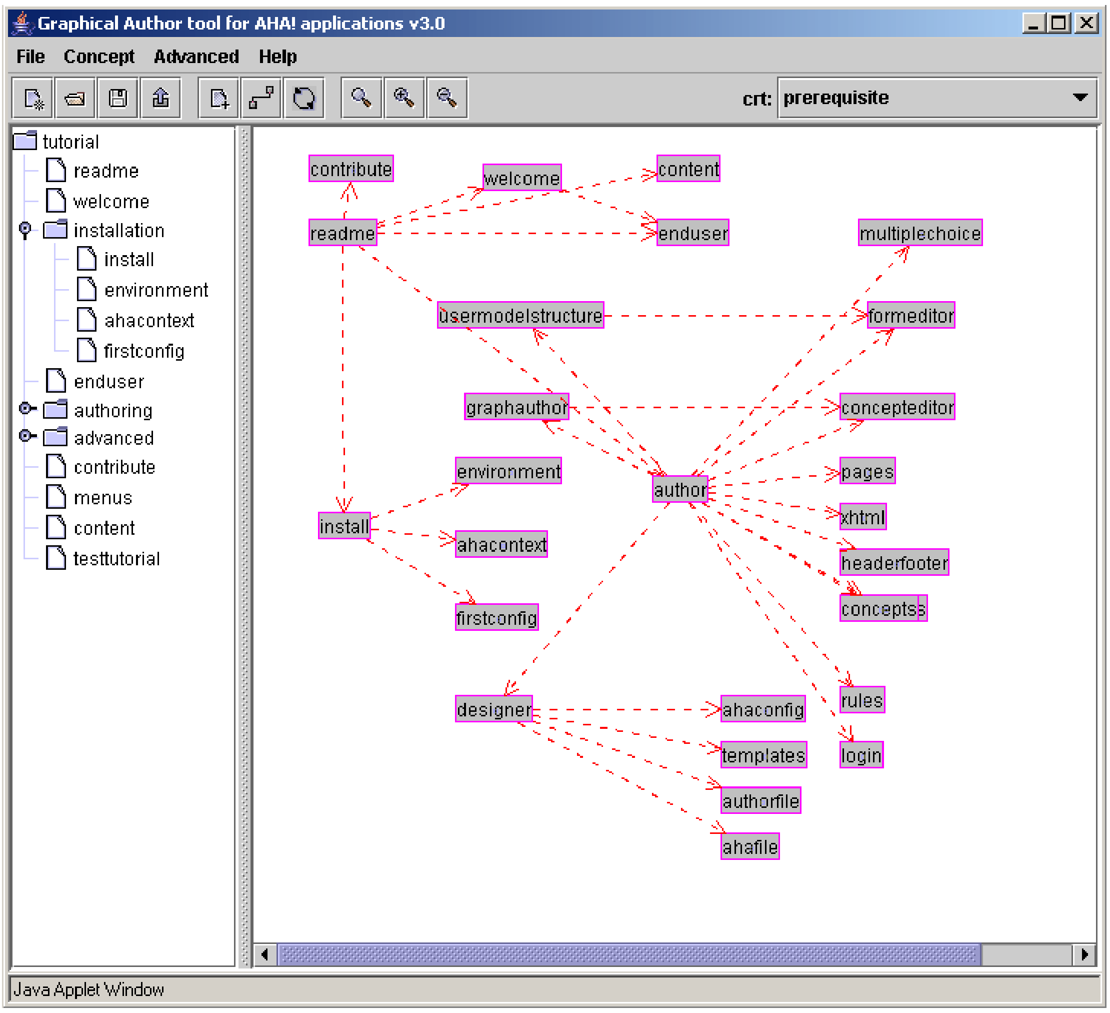


Figure 4.1: AHA's domain model graph authoring tool, from [45].

Another graph-based authoring tool can be found in AHA's latest versions [45]; a screenshot can be found in fig. 4.1. Since its presentation in AHA version 2, it has become the recommended tool for AHA course creation. While the graph authoring tool does support zooming and panning (it is based on the JGraph library), there is again no support for automatic layout. Unlike ATLAS, support for filtering is limited to showing or hiding certain relationship types.

Both AHA's graph authoring tool and ATLAS lack support for round-trip editing: graphs edited with these tools can be saved to an internal format and then exported into their respective systems, but neither tool can perform the reverse operation. That is, courses created (or simply modified) outside the authoring tools can not be opened again from these tools. This is obviously a problem when dealing with legacy contents.

Since certain valid course structures for AHA courses cannot be represented in the authoring tool, and the same can be said for TANGOWcourses and ATLAS, authors using either of these systems are expected to avoid such unsupported structures, or to postpone their introduction until all graph-based authoring is finished. This leads to a "cascade" content development model, where revisiting early design decisions made on the graph requires an author to forfeit all subsequent modifications. In complying with the guidelines set forth in section 3.2.3, authoring should be incremental, avoiding monolithic designs that may prove too large for initial requirements or, later on, fail to scale as expected.

While graph-based interfaces for adaptive hypermedia course authoring do exist, the examples that have been presented so far do not address the full complexity of the problem. Large graphs cannot be efficiently dealt with, since the location of each vertex has to be explicitly set by the user. Introducing a new edge may can therefore make a layout require substantial manual effort by the author to make it reflect the graph's actual structure. Although AHA's graph authoring tool includes support for zooming and panning, these are unwieldy for large graphs. Support for user model filtering in ATLAS is interesting, but it is limited in scope. For instance, there is no way to filter out an entire portion of the course deemed "not interesting". A similar problem exists within AHA's tool: filtering is limited to showing or hiding edges, but entire areas of the graph cannot be selectively shown or hidden. Lack of round-trip support in both systems is also a drawback.

Adaptive hypermedia authoring (not only that of courses) is hard to do. The requirements involve multiple related models, and a creative setting where authors should be allowed to experiment and retrace their steps if necessary. This is a knowledge-intensive process, and authors can be expected to use tools to support reasoning and exploration as much as to perform actual data entry. Graphs are ideally suited to represent and visualize the complex structures involved in AH authoring and other domains; but there are many design choices that can lead to better interfaces.

Part II

A Hierarchical Clustering Approach

Introduction

Previous chapters have motivated the importance of visualization in Adaptive Hypermedia and other areas where small-world networks can be found. Related knowledge fields, such as Graph Theory, Clustering, Information Visualization and Human-Computer Interaction, Graph Layout, Adaptive Hypermedia, content and structure reuse, ontologies and hypermedia authoring have been introduced. Part II brings these fields together, and presents the approach proposed in this work, together with several applications built on top of it.

Chapter 5 describes existing approaches to the general problem of graph visualization. CLOVER, a cluster-oriented visualization environment and the main contribution of this work, is presented next. The last two chapters describe applications built on top of CLOVER for Adaptive Hypermedia editing and visualization and several other fields, demonstrating the flexibility of the chosen approach.

Outline of Part II

- Chapter 5 analyzes different approaches to graph visualization, illustrating them with examples.

- Chapter 6 presents the main design decisions behind the approach of the CLuster-Oriented Visualization EnviRonment (CLOVER) framework. The first pages contain an overview of the framework's architecture. Later subsections provide greater detail regarding the *model*, *view* and *control* issues involved.

- Chapter 7 introduces a CLOVER-based authoring tool for the WOTAN adaptive hypermedia course system, WOTED. WOTED allows course monitoring in addition to standard authoring. A final section compares WOTED's clustered graph interface to the traditional tree representation.

- Chapter 8 demonstrates the flexibility of the framework, describing a series of CLOVER-based applications developed for series of different domains. Document reuse, ontology and metadata representation, academic plagiarism detection, and dynamic device networks are used as examples of the generality of the approach.

Chapter 5

Approaches to graph visualization

There are several approaches to interactive graph visualization. This section analyzes existing tools and technologies and relates them with the theory found in previous chapters.

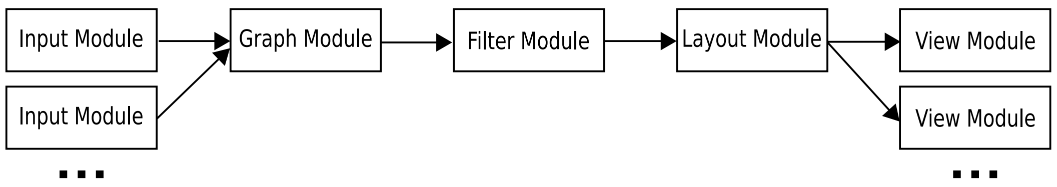
All Information Visualization systems contain stages similar to those exposed in fig. 3.1: data is transformed into an internal model suitable for the chosen representation, selected parts of this model are chosen for actual display, a graphical mapping is generated (for instance, in the case of graph visualization, layout is performed), the results are displayed, and the user interacts with this display, altering one or more of the previous steps. In the discussion of possible approaches, each of these steps will be covered in turn. A first section provides insight into the choice of general architecture.

5.1 General Architecture

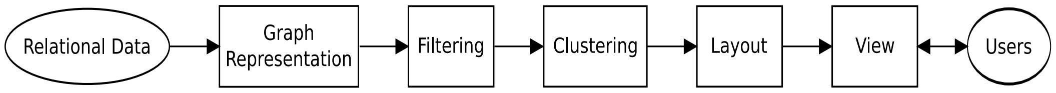
The internal data model from which representations are derived will be referred to simply as *model*; and the graphical representations themselves as *views*. This distinction is traditional in the user interface community, and reflects the importance of keeping both layers clearly separated. A single model may be represented in many different ways, and mixing the model with the view greatly complicates the use of several viewpoints. For instance, when visualizing a graph, it may be illuminating to generate multiple views with different degrees of detail. This would be impossible if the coordinates for each vertex were to be stored in the graph model itself.

The approach of layering processing steps, so each step is as independent as possible from later ones, has been successfully applied in a variety of scenarios. Network protocol stacks, for instance, rely heavily on the independence and interchangeability of each layer to simplify design and design changes.

There is, however, a great degree of freedom in the internal details of the pipeline of a graph visualization application. The internal complexity of the constituent modules can vary significantly, and it is generally possible to connect or configure modules in many



(a) Pipeline from Marshall, Herman and Melançon [95].



(b) Pipeline from Huang, Eades and Lai [82].

Figure 5.1: Pipelined architectures of two graph visualization frameworks

different ways. This is one of the advantages of a layered design. Figure 5.1 provides two examples of the *visualization pipelines* of two graph visualization frameworks as represented by their authors. The only real difference between both pipelines is the inclusion of a clustering module between the filtering and layout steps of Huang, Eades and Lai's approach (subfigure *(b)*). Similar pipelines, for other information visualization tasks, have been described in [39].

Both of the pipelines of fig. 5.1 omit the fact that the user can provide feedback to each module (presumably mediated through the interface); this feedback would correspond to the top lines of fig. 3.1.

However, such "pipeline" illustrations do not capture fundamental details on what is communicated between the different layers, nor the sequencing of the operations. It may seem apparent that, after a change in the graph representation, filtering will be updated, then the clustering, and then the changes will be propagated to the layout and finally reach the views. However, this propagation does not need to be immediate or automatic; a step that is notified of changes upstream may decide that these changes are not relevant downstream. For instance, a change in the base graph that does not alter the results of filtering would not pass to the next step. Conversely, a single upstream change may make a step generate multiple downstream notifications, covering different aspects of the single change. The pipeline illustrates only the broad flow of communication between steps, but does provide any information about the particular dialogues that can take place.

Event-driven interfaces

A common paradigm when connecting modules to each other, and many possible configurations must be supported, is to minimize the amount of information that modules should share. This can be achieved via *events*, implementing the *subscription* pattern described in [67].

Whenever a change occurs to the data in one portion of the visualization pipeline, the change may require notifications to be sent to all downstream processing steps. If the pipeline branches, for instace as a result of multiple views defined on the same model, all branches need to be informed. Hard-coding within each module the list of further modules to notified upon a change requires advance knowledge of the pipeline structure; this is impractical if users are allowed to create new views and alter their processing steps dynamically. In the subscription design pattern, the burden is shifted from the source module to its "subscribers", which must register themselves at the event source during their intialization. The module acting as event source only needs to maitain a list of generic subscribers.

These systems are termed "event driven", since all changes can be traced back to events acting on a given component and propagating throughout the rest. Most graphical user interfaces use this paradigm. Event-orientation has two important advantages:

Event sources are isolated from sinks – there is a list of sinks to deliver the event to. Who they are or what they do with the message is none of the caller's business. This simplifies the design and development of modules which can launch events; they only need to support a simple subscription mechanism, and it is up to the subscribers to register themselves and process the message.

Event sinks are isolated from sources – the message can contain all the data needed to act upon it; as long as the event is self-contained, the implementation details of the source are not important. If events can be "undone", implementing undo/redo support is simply a matter of keeping a historical stack of received events, which can then be unwound to return to previous states. Furthermore, having multiple event types interpreted as the same logical event is a common behavior in many GUIs; for instance, many graphical text processing actions can be launched via the menu bar, using a button, or pressing a certain key shortcut.

The subscription mechanism allows data flow within an event-oriented application to be a matter of plugging event sources to event sinks, none of which needs to agree on anything other than the subscription mechanism to use and the contents of events. In clustered graph terminology, the dependency graphs of event-driven programs have a much higher clustering degree than would be attainable if modules were required to have explicit knowledge of each other.

Libraries, Frameworks and APIs

Many of the systems referenced in this chapter are designed to provide a *framework* on which to build user applications, sometimes spanning small parts of the entire graph visualization process. This is achieved by providing a *library* that is designed not only to be accessed (all libraries are), but actually built upon, extending its capabilities for a particular application.

While frameworks can be built on any programming language, the concept of inheritance built into object-oriented languages such as JAVA or C++ makes them prime candidates for framework development. Frameworks themselves may, in turn, rely on yet other frameworks. For instance, the SHRIMP [122] is a graph drawing framework built on top of the PICCOLO [26] information visualization framework.

The particular interface that a library exposes to "clients" (applications or libraries that use it) is termed an *Application Programming Interface*, or API.

5.2 Internal Representation

The creation of a graph data structure is the first step of the pipeline. Graph vertices within the internal representation should contain or reference the particular piece of data from which they originate. Graph APIs take one of two possible routes: either to leave the definition of a vertex entirely in the hands of the application, or to require the use of predefined vertices where properties can easily be added and queried by label (possibly including semantics for copy, clone and delete operations). A simple approach is followed in the JGRAPHT library; the second route is undertaken, for instance, in the JUNG library [101].

Two main models are used to maintain graph data-structures. One option is to use an *adjacency matrix*, a square matrix with one row per vertex, where a value of 1 for cell A_{ij} would represent an edge from vertex i to vertex j. The second model is to maintain, for each graph vertex, a list of edges that are connected to it; this is termed an *adjacency list*.

Adjacency matrices are only appropriate for small, dense graphs, since they require $O(|V|^2)$ space, and this is very wasteful in the case of larger graphs. Furthermore, many interesting graphs are sparse, and the matrix would contain mostly zeroes. Insertion and deletion of vertices is also a problem, since removing a vertex requires entire rows and columns to be copied and updated if the matrix is to remain compact (which could be important, due to the quadratic size requirements).

Adjacency lists, on the other hand, are slow when searching for particular edges in high-degree vertices. In a simple list, linear lookup in the adjacency list of vertex v would be required to find all edges between v and w, an operation that would have required a single matrix lookup using an adjacency matrix. A common solution is to perform a space-memory tradeoff, by means of a data structure that supports fast lookups. For instance, JUNG [101] uses hash tables to speed up adjacency lookup to near $O(1)$ time complexity. The same can be said for other APIs such as JGRAPHT [98].

5.2.1 Generating Clusterings

As discussed in section 2.3, there are many possible ways to generate clusterings. Although the quality of the clustering is vital to generate meaningful graph abstracts, the process of generation itself can be isolated into a separate module, with a clear-cut interface: *generate* a clustering for a given graph, and *update* a clustering given a series of changes to a base graph. This subsection deals with the process of clustering generation.

Huang, Eades and Lai [82] propose the use of a definition of vertex-to-vertex distance on the base graph to generate a distance matrix, and the application of a general K-means clustering algorithm on this matrix to yield the clusters themselves.

Raitner [109] does not expand on how the clusterings are generated, and it is understood that they are are completely external to the framework; the framework can apply them to the current graph, and can perform suitable updates on the clustered graph if it is informed of the exact list of changes to the clustering, but does not generate or update the clusterings themselves.

Van Ham and Van Wijk [132] use the result of a full-graph layout to identify groups of nearby vertices, which are then treated as clusters.

Auber and Jourdan [19] propose the use of the metric described in equation 2.3 within section 2.3 to find suitable cutoff values at which to generate clusters according to the graph's connectivity (this interactive step could also be merged into the approach of [82]).

Updating the Cluster Hierarchy

When changes are performed to the base graph, they should generally be reflected in the hierarchy. A recently added edge, for instance, may force a series of clusters to be recalculated. In a rule-based system, rules that were triggered before may no longer be applicable; if a clustering metric is applied, threshold values may have been affected, yielding a different clustering from the previous one. If clusters have been calculated from the layout positions of vertices, vertices may have shifted due to a new layout.

The problem of updating the clustering is not that of re-running the original clustering algorithm to generate a new hierarchy, but of informing the remaining steps of the pipeline of exactly what changes have been performed. Not every change to the cluster hierarchy need be reflected; for instance, a fully-expanded graph view would never require updates due to the hierarchy change itself (structural changes to the base graph would, however, require display). But, if a given view was displaying one of the cluster-vertices that have been removed from the hierarchy, changes will be inevitable.

Existing hierarchically clustered graph visualization systems do not appear to have support for automatically rebuilding their hierarchies after changes to the base graph. An alternative is to require users to perform all such changes manually; but this is cumbersome, and does not work well with changes to the base graph that were not directly triggered by user actions (imagine a collaborative editor). Once a hierarchy change has been decided on, it is important to animate its effects, specially if the user is not aware of its location and extent.

In locating changes from one automatically generated cluster hierarchy to another, two scenarios are possible:

- If the clustering algorithm has been designed with support for “incremental” clustering, it may be able to supply a set of changes directly. Building such support requires careful study of all possible incremental changes to the graph, and how they can affect the final clustering, and is only possible for certain algorithms.

- Do not attempt to track clustering changes as they occur during the clustering process; instead, compare the initial and final hierarchy and report back a minimal set of changes from one to the other. An advantage of this approach is that it is totally independent of the clustering algorithm.

The second approach requires the computation of the set of differences between two trees. Algorithms to locate differences between rooted, labeled trees have been described in the literature, most notably [141]. The run-time complexity of a general difference-locating mechanism can be made proportional to the extent of the differences and the size of the tree: $O(kn)$ (where k is the number of changes).

5.2.2 Representing a Clustered Hierarchy

Few systems deal with clustered hierarchies in graphs. A good discussion of data structure efficiency in clustered hierarchies can be found in Raitner’s work [109], which compares his own data structure to those of Buchsbaum and Westbrook [38]. In the comparison, the following basic clustered graph operations are examined:

`edgeQuery(e)` Determines whether the induced edge e exists.

`edgeReport(e)` Determines the components of edge e, that is, if e is an induced edge, the set of original graph edges that induce e.

`expand(v)` Expands cluster vertex v.

`collapse(v)` Collapses cluster vertex v.

`newEdge(e)` Add a new edge to the graph. Only edges between leaf vertices (those that were present in the original graph) are allowed. This is different to [109], where edges can be added between any two tree vertices.

`deleteEdge(e)` Delete a leaf edge.

`newLeaf(v)` Add a new leaf vertex.

`deleteLeaf(v)` Delete a leaf vertex.

Table 5.1 contains a comparison of different internal representation approaches. The **Simple** approach only requires additional storage for the current set of visible vertices. Visibility has to be recomputed after every operation. It is therefore optimal in space, but expensive in time. The **Naive** approach, described in [38], achieves much higher efficiency. For every vertex w, $N(w)$, the set of induced edges such that $p(u) = p(v) = w$ is precomputed. Additionally, each induced edge $(u, v) \in E_I$ stores the list of all the edges that it subsumes in $L(u, v) = \{(u, w) \in E_I : w \in \mathit{children}(u)\}$ and $R(u, v) = \{(u, w) \in E_I : w \in \mathit{children}(v)\}$ (L and R refer to depth-first numbering and a top-down, left-to-right tree drawing). Edges to be shown during an expansion of vertex u are found using equation 5.1; note that $\mathit{inc}(u)$ refers to the set of all edges that are incident to u in the currently-visible graph.

$$\bigcup_{(u,v)\in \mathit{inc}(u)} L(u,v) \quad \cup \quad \bigcup_{(w,u)\in \mathit{inc}(u)} L(w,u) \quad \cup \quad N(u) \tag{5.1}$$

Once u is expanded, both u and all edges previously in $\mathit{inc}(u)$ should also be removed. Since for each of these incident edges there will be at least one edge added to a descendant of u, the complexity is $O(\sum_{z\in \mathit{children}(u)} |\mathit{inc}(z)|)$, which is optimal. This optimal complexity will be referred to as $\mathrm{Opt}_{G_C}(u)$ and is applicable both in expansion and collapse (which is symmetrical to expansion). However, in an expansion, $\mathrm{Opt}_{G_C}(u)$ refers to the clustered graph existing prior to the expansion. In a collapse, complexity would be expressed as $\mathrm{Opt}_{G'_C}(u)$, referring instead to the clustered graph after the collapse.

The space requirement for the **Naive** approach results from adding the requirements for $N(\cdot)$, $L(\cdot)$ and $R(\cdot)$ for each edge. This is influenced by the number of edges and the depth of the hierarchy tree, D, yielding $O(|E|D^2)$.

Buchsbaum and Westbrook also define more space-constrained methods, which will not be explained in depth. They are included in table 5.1, together with Raitner's results using tree cross products, labeled under **Raitner**.

	Simple	Buchsbaum-naive	Buchsbaum-uncompressed	Raitner
Additional space	$O(1)$	$O(\|E\|\ D\ log^2\|V\|)$	$O(\|E\|\ D\ log^2\|V\| + k)$	$O(\|E\|\ D)$
`edgeQuery`(u,v)	$O(\|E\|\ D)$	$O(log^2\|V\|)$	$O(\|E\|\ D\ log^2\|V\|)$	$O(log\|V\| + k)$
`edgeReport`(u,v)	$O(\|E\|\ D)$	$O(log^2\|V\|)$	$O(\|E\|\ D\ log^2\|V\|)$	$O(log\|V\| + k)$
`expand`(v)	$O(\|V\| + \|E\|)$	$O(Opt_{G_C}(v))$	$O(Opt_{G_C}(v)\ log^2\|V\|)$	$O(Opt_{G_C}(v)\ D\ log\|V\|)$
`collapse`(v)	$O(\|V\| + \|E\|)$	$O(Opt_{G'_C}(v))$	$O(Opt_{G'_C}(v))$	$O(Opt_{G'_C}(v))$
`newEdge`(u, v)	$O(1)$	$O(s^2 log\|V\|)$	$O(D\ log^2\|V\|)$	$O(D\ log\|V\|)$
`deleteEdge`(u, v)	$O(1)$	$O(s^2\ log\|V\|)$	$O(D\ log^2\|V\|)$	$O(D)$
`newLeaf`(u)	$O(1)$	n/a	n/a	$O(D)$
`deleteLeaf`(u)	$O(1)$	n/a	n/a	$O(D)$

Table 5.1: Comparison of time and space complexities of different approaches to clustered graph updates, from [109]. D is the depth of the hierarchy tree, $s = min\{D, log|V|\}$, and k is the size of the output for the `edgeReport` operation. The **Simple** approach corresponds to using no auxiliary structures. The **Buchsbaum-naive** and **Buchsbaum-uncompressed** approaches are described in [38] (compressed trees provide slightly better space efficiency, but are not included in the comparison). **Raitner** is described in [109].

5.2.3 Filtering

The goal of filtering is to avoid presentation of certain parts of the data. Filtering can be applied in different parts of the pipeline. Implicit filtering is performed when the data is first transformed into a graph: parts of the data that are not mapped to vertices or edges can be considered to have been “filtered out”. It is also common to apply filtering once the initial graph has been built; in this case, the filter is queried with each vertex and edge in the original graph - only those vertices and edges that pass the filter will proceed to the next pipeline stage. Filtering can also be applied at later stages.

In the case of clustered graphs, filtering performed before clustering and layout is the most natural approach, since from the point of view of generated representations, these will be identical to representations which could have been generated from data that did not include the “filtered out” portions. Filtering after clustering introduces additional problems: should empty or nearly-empty clusters be represented? how?. It may, however, prove useful in certain scenarios, most importantly when representations generated with different filters are to be compared; in this case, performing filtering as a last step allows easy mapping from one representation to another, by retaining a common layout: same layout position, same vertices. This comparison is more difficult in the case of early filtering: the clusters will probably not be the same, and nor will the positions. Forcing a graph into a layout that is not well-suited to this graph can result in aesthetical problems, which relate directly to how well it can be understood.

Filter chains

Filter chains are also a natural idea: combine several filtering operations into a larger filter, so that only vertices and edges that manage to pass all filters are considered to pass the overall filter. This allows simple filters to be reused and composed into more complex entities. This type of advanced filtering is a hallmark of systems designed to work with large datasets, such as JUNG [101]. It is also possible to combine simple filters in more complex configurations, such as boolean filter trees; but their use is considerably more rare.

JUNG further defines the term *unassembled graph* (a graph where some edges (u, v) may reference endpoints that have been filtered out) for use with filters. The idea is to allow simple vertex filters to be applied without worrying about possible disconnected edges, achieving a greater performance. At the end of a chain of filters that work on unassembled intermediate graphs, the graph would be rebuilt and processing would continue as normal.

5.3 Layout

Many graph visualization systems include a set of layout algorithms that can be applied to the current view. The set of layout tools spans tree layouts, force-directed layouts, and more complex layouts such as Kamada-Kawai or Sugiyama. Complete applications usually provide the user with a menu to choose the layout to apply; libraries provide the layouts as a pipeline step to be applied where appropriate.

Graph layout algorithms accept a variety of parameters; these are provided when the algorithm is run. Graph visualization systems such as JGraph [1] or yWorks [140] include layout packages with associated configuration dialogs, which can be displayed when the user decides to launch a particular algorithm. As with filters, layout algorithms can be stacked to build more powerful layouts. For instance, many algorithms expect to layout only single graph components. When multiple components must be laid out, a simple approach is to layout each component individually, and then use another layout to place the components together (avoiding overlaps). Other common layout steps include vertex overlap removal, beautification of edge crossings, or edge routing.

Manual placement is also frequent in graph visualization, except for very large datasets where it becomes impractical. All tools provide a user-interaction mechanism for users to visually displace graph vertices, and most also make provisions for manual edge routing. Both manual layout edits and automated layouts usually count as "actions" that can later be undone through an undo/redo mechanism. It is also typical for automated layouts to be able to "start" from the previous layout position (force-directed layouts greatly benefit from good starting positions).

5.3.1 Incremental Layout in Clustered Graphs

Graph layouts provided by most visualization systems are "one-shot" affairs: at a given moment, the user specifies the layout algorighm to be performed, the algorithm runs, finishes, and vertices are moved to their new positions.

Clustered graphs, however, require frequent relayouts to accommodate changes in displayed edges and vertices, either as a result of different degrees of abstraction due to user navigation, or as a result of changes to the base graph. A layout will be termed *dynamic* if it can successfully deal with these changes without breaking the user's mental map. The term *dynamic graph* is applied to graph visualizations that incorporate time into their changes, creating animations; animation is necessary to provide a transition from a previous to the updated version. This subsection deals only with the generation of an incremental layout, and not with the animation mechanism itself.

Strictly speaking, it is possible to allow navigational changes in degree of detail without changing the layout at all. This is the approach followed in Van Ham and Van Wijk's [132] system, illustrated in figure 5.2. A single, large layout of the whole

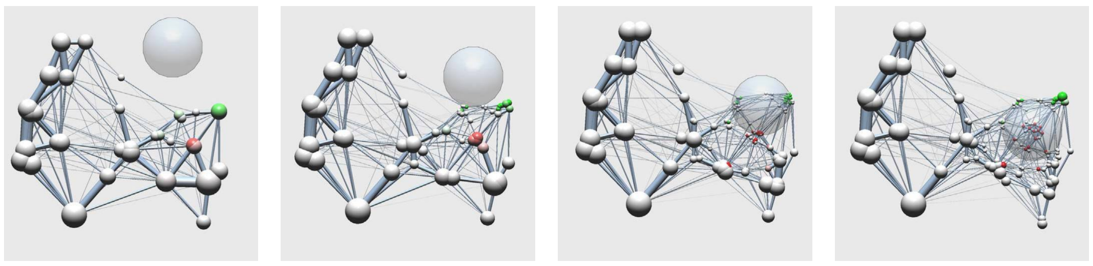


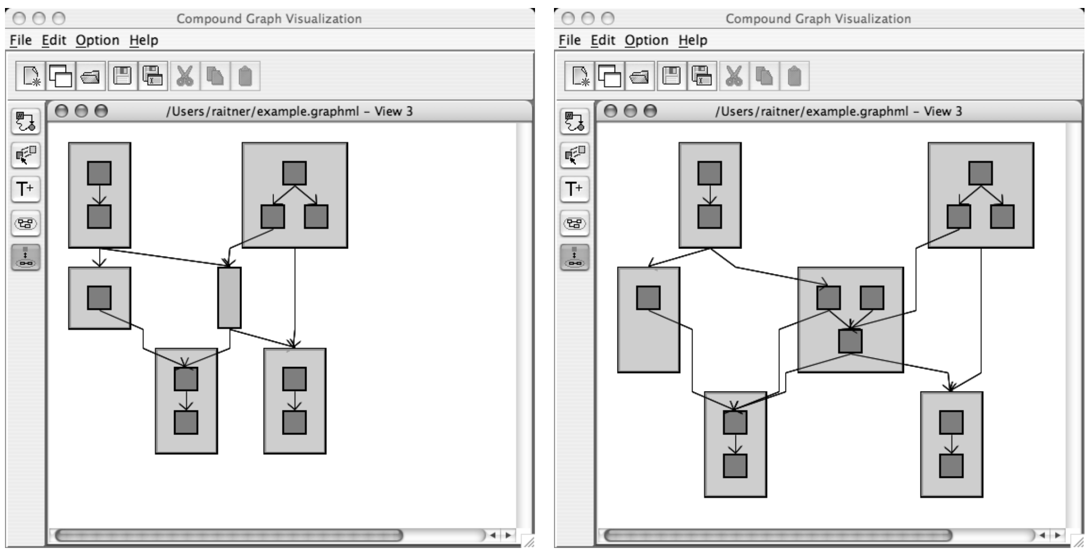
Figure 5.2: Navigation of a clustered graph without relayouts, from [132]. The transparent ball represents the focus; vertices and edges within the focus are shown in their real positions, those outside are represented by large cluster vertices or cluster edges.

graph is performed, using an algorithm where node proximity is closely related to cluster belonging. Clusters are deduced from placement, and individual vertices that belong to the same cluster are rendered as a single cluster vertex. Navigation is performed by avoiding such substitutions in the neighborhood of a "focus". This is useful when the base graph does not change. Changes to the base graph would require an incremental layout to be performed; and, with all probability, this would result in many layout-derived clusters being lost.

The two groups that have published results on incremental layouts applied to hierarchically clustered graphs are that of Peter Eades ([49, 52, 82]) and Marcus Raitner ([21, 103, 106, 107, 108, 109]). Layouts for clustered graphs must be incremental in order to maintain the user's mental map. Eades' group proposes the use of force-directed layouts, while Raitner's proposes modifications to the Sugiyama-Misue algorithm to account for incremental layouts. It must be noted that Eades has not described any complete clustered graph visualization tool, and their articles only cover partial aspects. Raitner's work, in this respect, is more complete, with successive implementations of HGV [107], GRAVISTO [21] and VISNACOM [103, 106]. A screenshot of VISNACOM in action is presented in fig. 5.3.

An advantage of Raitner's use of an approach based on the Sugiyama-Misue algorithm is that it avoids some problems present in force-directed layouts: the high sensitivity of the layouts to minute changes in starting positions, and the tendency of layouts to "drift". Since in force-directed algorithms all vertices affect all others (by means of repulsive forces), chaotic effects similar to those of the n-bodies problem can be observed. Raitner's approach is depicted in fig. 5.3; more information is available at the VISNACOM website [104].

Both these systems perform relayout as an independent step from animation. That is, after a change is performed, layout is recomputed, and after the computation has ended, layout changes are animated within the user interface.


Compound Graph Visualization
File Edit Option Help
/Users/raitner/example.graphml – View 3
Compound Graph Visualization
File Edit Option Help
/Users/raitner/example.graphml – View 3

(a) Before expansion of central cluster (b) After expansion of central cluster

Figure 5.3: Incremental layout based on the Sugiyama-Misue algorithm developed by researchers in Raitner's group [103, 106].

Interactive Update in TouchGraph

Another system that performs interactive, incremental layouts is TouchGraph (commercial version available at [11], open version available at [12]). However, TouchGraph does not operate on a true clustered graph, in the sense that all vertices more than k hops away from the currently central vertex are elided, and their edges are not represented. Therefore, a snapshot of a visualization generated with Touchgraph is not an abridgment of the graph, since any route that falls beyond the current display radius is not represented. In a true clustered graph, this would result in induced edges.

TouchGraph is also unusual in performing layout automatically after any user intervention; disturbing the layout results in an incremental, interactive force-directed relayout being performed, until convergence is again reached. Interactive, in this context, means that it is possible to perturb the layout even while relayout is under way. No separation is performed between relayout and animation, unlike in the systems proposed by Eades and Raitner.

Touchgraph is being actively used for adaptive hypermedia authoring by Darina Dicheva's group ([16, 47, 48]). A screenshot of Dicheva's implementation is provided in fig. 5.4.

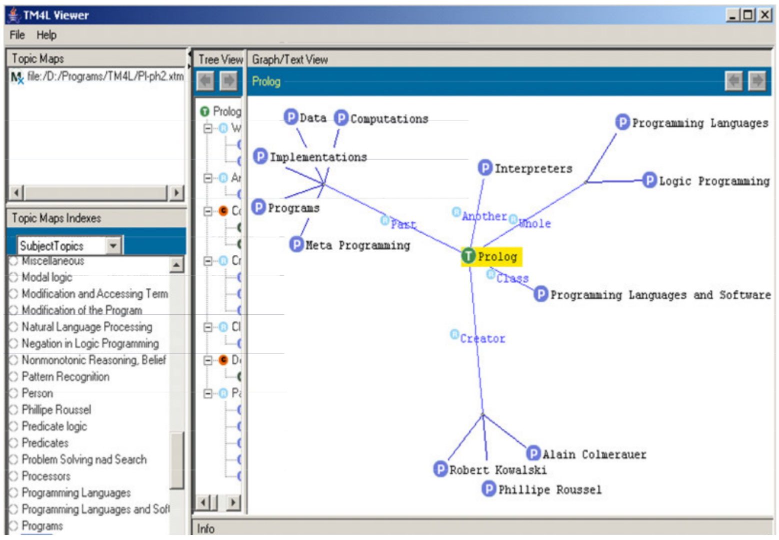


Figure 5.4: A screenshot of T4ML, a TouchGraph-based topic map authoring tool, from [47].

5.4 Presentation and Interaction

After layout has been performed, a rendering of the graph's vertices and edges using the generated layout is presented to the user. If the graph is large, it is unlikely that the full layout will fit in a single window with full detail; therefore, the user will have to navigate the display in order to access hidden portions of the graph with any degree of detail. This issue was presented in section 3.2.1, together with techniques to overcome. In this section, the application of these techniques to current applications and frameworks is examined.

Many presentation and interaction features are provided directly by the window toolkit (such as MFC, QT, GTK, Carbon or Java's Swing); where not available, they can generally be provided by the graph visualization framework. Many of the available graph visualization frameworks ([11, 140, 130, 107, 80, 94, 21, 122, 104, 101, 98]) are programmed in Java which offers an easily extensible component model, runs in several platforms, and supports feature-rich toolkits such as Swing or SWT. Notable exceptions are Tulip [17, 18, 19], which is programmed in C++ and uses the QT windowing system, and Pajek [24], which is also a C++ application.

All frameworks allow the generation of graphs that are larger than the available space, and must deal with the detail and context issues described in section 3.2.1. This is supported by rendering visual components (for instance, vertices and edges) into "virtual canvases" which need only be partially displayed; again, these can be provided either by the window toolkit or by the visualization framework. Client applications are allowed to paint outside of the visible bounds, and by selecting which offsets within the virtual canvas should be displayed, a *viewport* can be determined. Shifting the viewport, or *panning*, is performed through interaction with scrollbars, or by a "grabbing" mouse gesture where, having clicked on a given point of the graph, the point (and the whole viewport) is shifted together with the mouse pointer until the mouse button is again released. Other typical means to control panning involve a "center here" action.

Zooming is present in most toolkits and frameworks; under JAVA's SWING toolkit, zoom can be used when rendering almost any graphically displayed component, since the rendering process can be configured to use an intermediate step where an affine transformation (scaling, rotation or translation) of coordinates is performed. If this step is missing in a different toolkit, it can be implemented without great hassle.

When zooming into a view, a portion of the view will be enlarged. If the zoom gesture is invoked with the mouse (a typical binding is the scroll wheel found on most recent mice), it makes sense to enlarge the area that is nearest to the mouse pointer. This gives rise to the definition of *center of zoom*, which would be the point of the display which, after the zooming operation, preserves its previous location on screen.

Other types of zoom may be provided. When a graphical object (for instance, a vertex) becomes too small to make out the details, it is both more informative and more efficient to substitute these details for an intelligible, abstracted representation – or to elide the details completely. For instance, when presenting overviews of graphs with hundreds or thousands of vertices, it is common to elide the vertices and show only the edges; therefore, rendering can vary depending on the current zoom level. It is also possible to zoom the layout itself, instead of its rendering; this would make edges appear to shorten, while retaining the size of all vertices and text labels.

To speed up the rendering process, applications can avoid to render areas that are outside the viewport that will be presented to the user, and rendering those areas with a degree of detail that matches the current zoom level. While most toolkits and frameworks perform this optimization to a certain degree, *Zoomable User Interface* (ZUI) toolkits such as PICCOLO or JAZZ [26] take this to an extreme, providing extremely fast and smooth zoom and pan interaction accross scale intervals that can cover several orders of magnitude. For instance, SHRIMP [122] is a graph drawing framework built on top of PICCOLO; it is used, among others, by the JAMBALAYA [121] graph-based ontology visualization plugin used in the PROTÉGÉ [100] system.

Multiple views of the same graph layout are also generally available on any framework; by combining a zoomed-out view that shows the graph in its entirety (with very low detail), and a more detailed view where details can be appreciated, it is possible to build simple overview + detail interfaces. This can be seen, for instance, in JAMBALAYA, or in the JGRAPH-based JGRAPHPAD [6] graph editor, depicted in figure 5.5.

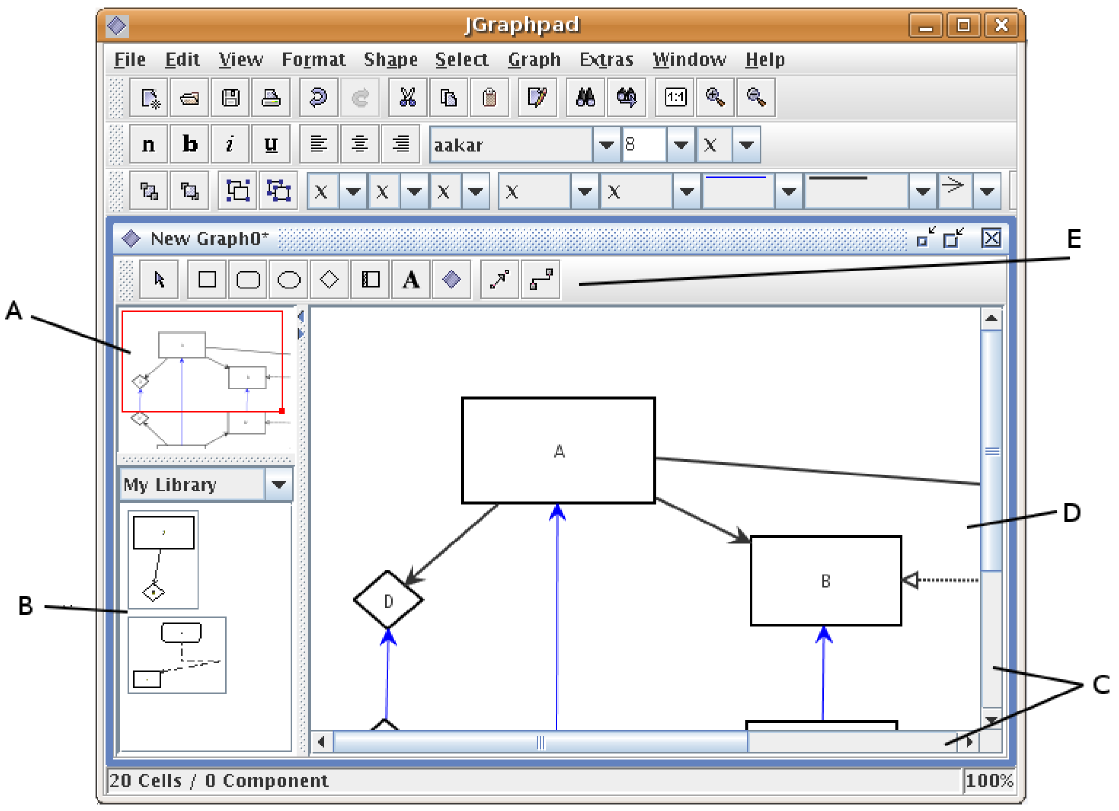


Figure 5.5: JGraphPad graph editor, displaying an overview (A) with the visible portion within a red boundary, a library of subgraphs (B), scrollbars to perform pan operations (C), a detail view (D), and a series of modal editing tools (E). JGraphPad is based on the JGraph[1] graph visualization library; this library is also used by Clover for graph representation.

5.4.1 Interaction

Interaction with graph interfaces is similar to that found in most graphical user interfaces: the view can be navigated with zooming and panning, displayed elements can be selected, edited, removed, cut, copied or pasted, and new elements can be introduced. Available operations on vertices vary depending on visualization framework and application, but most are relatively standard.

Many graphical user interfaces support the notion of *actions*, which can be described as an encapsulated user operation (*actions* correspond to the *Command* pattern described in [67]). Any user command, be it zoom, edit, copy or paste, can be considered as an action; the advantage of actions is that, once defined, they can be triggered in many different ways; for instance, through a keybinding, or a mouse gesture, or a contextual popup menu. Additionally, if actions contain enough information to allow being "undone", it is trivial to implement "undo last action" and "redo last action" actions, which are key in allowing users to explore the interface without fear of entering a blind alley.

Whenever an application makes use of actions, it is very simple for developers to alter the key bindings, or mouse actions, or contextual popup menus which trigger these actions. Therefore, few if any scholarly publications describe the exact bindings which are used in the systems they describe. Although a good selection of action triggers is important from a usability point of view, it is trivial for an implementer to swap these triggers for an entirely different set.

Actions are not all used with the same frequency, and their distribution depends on the context. At times, a user may want to edit the graph itself, adding new vertices and edges, and at other times, the goal will be to browse quickly in order to find a certain data item. The available action triggers, however, are not all equal. It is much more immediate to click on a vertex to edit it than to bring up a contextual menu and select the "edit" option. *Modal* interfaces allow the user to alter the set of action bindings (that is, the meaning of certain action triggers) according to the currently-selected interaction mode. For instance, many photo editing applications allow the user to enter "selection" mode, where a dragging mouse gesture over the photo will select a portion of it, as opposed to a "pencil" mode, where the same gesture would cause a line to be displayed.

Although no clustered graph visualization environment is known by the author to implement a semantical fisheye lens on clustered graphs, equivalents can be found in systems designed for large hierarchy visualization, for instance in the PREFUSE [8] TREE-VIEW system [77].

5.4.2 Animation

Animation is a common topic when designing graph interfaces that seek to minimize loss of mental map. Indeed, it is relevant to any system where changes are going to be displayed but context (an specially mental context) must be preserved.

The general idea of an animation is to present many intermediate, continuous frames over a span of time, instead of directly displaying the final state of a given view. The brain has a tendency to see only what it concentrates on, and movement is very concentration-worthy [117]. Animation can be used to create a sense of continuity, and to direct attention to elements that are actually worthy of it.

Animation can also be used to explain a change to the user. In the case of the MAREY [62] graph animation tool, Friedrich and Eades propose the use of animation to convey the structure of a layout change, by presenting it a set of structured, animated updates to be performed on the previous layout, instead of performing direct interpolation. An example of this process is provided in figure 5.6.

A simple animation infrastructure consists of a loop where a delay is introduced, a new frame displayed, and the process is repeated until the last frame has been reached.

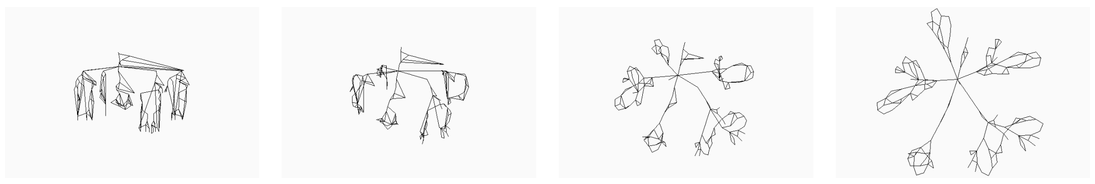


Figure 5.6: Animation of a graph layout change in the Marey tool, from [62]. An optimal three-dimensional rotation is first performed, and finally details are interpolated. Direct interpolation would have been much harder to follow.

When multiple animations for different visual objects must be performed at the same time, the existence of multiple loops is not efficient and can introduce race conditions. A centralized approach can prevent this problems: each "animation plan" must be sent to this module to be executed, where a single timer can request request each plan to draw the next frame when appropriate, avoiding unnecessary duplication of timers. A central approach also allows detection of conflicts between plans.

This is a rough outline of the animation process as present in the Piccolo [26] ZUI toolkit. Another system with an animation framework is VisnaCom [104]; however, Piccolo's infrastructure is considerably more powerful.

Chapter 6

The Clover Framework

This chapter presents the actual design of the CLOVER framework. The main highlights of CLOVER, when compared to alternative clustered graph visualization approaches, are the following:

- Completely automated propagation of changes from any stage to all "downstream", dependent stages and views. This requires fully automated hierarchy generation and (incremental) update, a key feature of CLOVER.

- All updates to graph views are smoothly animated and performed incrementally. Graph layouts are incrementally updated with changes, striking a balance between context preservation and overall layout quality.

- Powerful clustered graph navigation, with support for more than simple cluster expansion and collapse. For instance, a semantical fisheye lens is available, and cluster navigation history can be revisited.

The first pages contain an overview of the framework's architecture, which is structured in a similar fashion to those discussed in section 5.1, with a pipeline that spans from data acquisition to its final presentation in the graphical user interface.

After the architecture has been presented, the remaining sections provide greater detail regarding the *model*, *view* and *control* issues, following the Model-View-Controller (MVC) paradigm.

- The **model** section describes the successive transformations undergone by data from its initial source to the graphs that are finally presented on the screen.

- The **view** section deals with the presentation of the graphs on-screen, including graph layout, visual graph attributes, and the implementation of graph animations.

- The **control** section discusses the acquisition of user input and its effects on both model and view.

Clover is available for download under an open-source software license at
http://tangow.ii.uam.es/clover

6.1 Architecture

The Clover pipeline is illustrated in fig. 6.1; it is similar to other visualization pipelines found in the literature, particularly to that of 5.1 *(b)*. Distinctive features of Clover are highlighted with thicker lines: clustering can be updated, and focus+context navigation is supported. Top lines represent user interaction with the interface, and should be understood to be mediated through the interface itself.

The whole framework is designed to accommodate multiple views of the same base graph. Views may differ at any intermediate pipeline step, including filtering, superimposed clustering hierarchy, slice selection, and graphical properties such as zoom and layout. Pipelines can therefore branch into several different views. However, for simplicity, most diagrams, such as those of figs. 6.1 or 5.1 *(b)*, show only one active view.

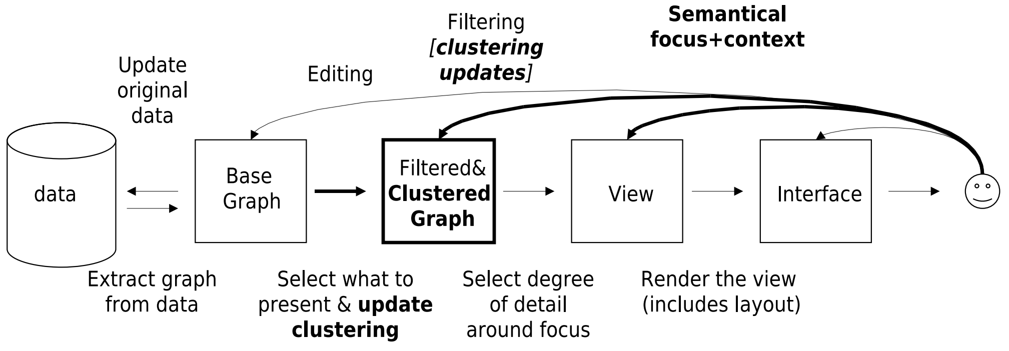

Update original data
Editing
Filtering *[clustering updates]*
Semantical focus+context
data
Base Graph
Filtered& **Clustered Graph**
View
Interface
Extract graph from data
Select what to present & **update clustering**
Select degree of detail around focus
Render the view (includes layout)

Figure 6.1: Pipeline used within Clover. Text in boldface and other highlighted elements correspond to distinctive features not found in other graph visualization frameworks.

Expanded pipeline

A more detailed overview of Clover's architecture is provided in fig. 6.2. The clustering hierarchy is separate from the base graph, which may be filtered prior to hierarchy construction. Maintenance of multiple filters, cluster hierarchies, clustered graphs and so on is possible. A clustered graph is constructed form the cluster hierarchy and the filtered graph; only a cut-set (*slice*) of the hierarchy clusters can be visible at any given moment. The operations of expanding or contracting clusters lead to changes in the visible slice, which are animated. Animation may require incremental layout, but can also be used to highlight specific graph elements (such as vertices or edges) without altering their positions, varying other visual attributes (such as color or line width) to draw the user's attention.

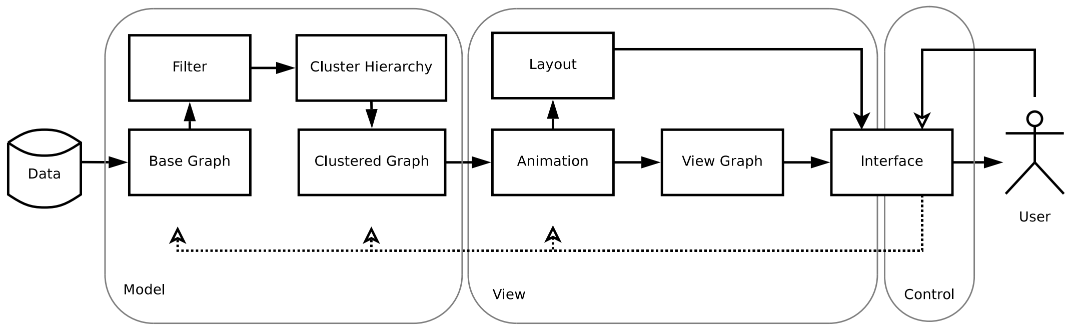


Figure 6.2: Detailed Clover pipeline. The dashed arrows located at the bottom represent interaction. Note the location of animation and the use of a "view graph".

The "view graph" component depicted in fig. 6.2 between "animation" and "interface" customizes the representation of the vertices and edges of a particular graph as required by the application. This allows base graphs to be independent from their visual attributes, simplifying reuse. The view graph is closely coupled with the interface, but changes to the clustered graph can reach it only after the animation component is ready to display them. A delay in change representation is at the heart of animation, and is greatly facilitated by this arrangement.

Interaction is also depicted in greater detail; all changes to other components are mediated by the control part of the interface component, which in turn performs the relevant changes to the interface itself, or to any prior step; these have been represented with dashed lines and empty arrows near the bottom of fig. 6.2.

Since Clover follows the Model-View-Controller design pattern, figure 6.2 highlights which components fall into each category (see large, rounded rectangles). The interface component combines responsibilities from both *view* and *control*; this is an approach found in many graphical toolkits.

6.2 Model

The model covers the creation of a graph from a data source, and the successive operations that are performed on this original graph in order to obtain the clustered graphs that will be displayed. A subsection is dedicated entirely to the subject of graph updates and how they are propagated to later stages, since Clover uses a fairly complex update mechanism. Indeed, this update mechanism is one of the main distinctive features of clover: any change to a model stage is automatically propagated to all dependent stages, triggering an animated update to the corresponding views.

Model graphs are represented internally in a adjacency-list like datastructure, with a small space tradeoff that provides nearly $O(1)$ edge lookup time. This is the same approach that is followed in graph libraries such as Jung. Clover's internal representation is supplied by the JGraphT library, described in [98].

6.2.1 Creation and Filtering

The base graph is the first step towards visualization: an abstract vertex-and-edge representation of the data that will later be displayed. Base graphs are expected to contain (or at least reference) all the information that will be somehow displayed further in the pipeline; CLOVER does not place any restrictions on the datatype of vertices or on the extra data that can be included within edges. At the very least, however, each vertex and edge should include a *label* that will be used when displaying them, and each vertex should have a unique *ID*. Vertex IDs are used in later stages when saving and restoring hierarchies and views. Check appendix B.2.1 for further details.

Graph data can be acquired from a variety of data sources; depending on the source, data acquisition can be a one-shot process (eg.: reading a file) or incremental changes can be expected to arrive over time (eg.: monitoring an online device network). The base graph is expected to perform the initial import, listen to any changes that may be forthcoming, and in the case of editors, write back changes to storage as required.

Data sources are implementation-dependent. The following list of sources is for illustrative purposes only.

Creating random graphs

During testing, generation of random graphs has been found to be useful. It is possible to generate random graphs to a plethora of specifications: with or without small-world properties, ensuring connectivity or not, or ranging from small to huge in size.

The random graph generator included in CLOVER for testing purposes uses a very simple algorithm to creating "random", connected graphs, taking only two parameters $|V|$ and $|E|$, with $|V| < |E|$.

```
createRandomConnectedGraph(vertexCount, edgeCount)
    create vertexCount vertices
    create a path through all those vertices
    for each of the (edgeCount - vertexCount)+1 remaining edges,
        choose a random start vertex
        choose a random, different end vertex
        if edge not repeated, establish the edge
```

These graphs do not exhibit small-world phenomena. Their main use has been during testing, to ensure that layout and clustering can deal with large, chaotic graphs.

Creating a graph from a database

Relational databases are common sources of data. They are also relatively straightforward to convert into graphs, using the following strategy:

```
createGraphFromDatabaseView(tables)
    for each row r in each table t
      create a vertex r of type t
    for each foreign key included in each row r
      create an edge from r to the destination row r2
```

Notice that, although this strategy will result in a graph that preserves all relationships, tables used to represent N to N relationships will result in spurious vertex types. For instance, if a doctor can attend several patients, and a patient can be attended by several doctors, the “Attends” table is an artifact of the database to represent this fact, but should not be included as a vertex in a graph generated from the database. Such relationships with only two participants can be expressed as simple typed edges. The transformation from “relationship tables” to typed edges can be easily automated.

Creating a graph from an XML or text file

A second common data format is *XML* documents. Although *XML* documents are hierarchical in nature, they can describe graphs easily through the `id`/`refid` mechanism: first, a sequence of *XML* elements is labeled with unique IDs (using the `id` datatype), which can then be referenced as attributes of the `refid` type from other elements. A common way of expressing graphs as *XML* (used for instance in the *GXL* graph-representation dialect) is to first provide the list of vertices and further on reference these vertices from a list of edges.

A very similar strategy is used in other textual graph representation formats. For instance, the format used in GraphViz[54] relies on first enumerating vertices and assigning a unique label to each, and then enumerating edges, including in each the pair of labels that identify its endpoints.

Creating a graph from an online system

In general, all graphs can be created in a similar manner, first locating the “entities” involved (which will be translated as vertices, or if they are actually reified relationships, edges), and then locating the edges. In online systems, the set of entities and relationships to be included is usually not known ahead of time; instead, it must be discovered at creation time. This discovery can be performed in an efficient manner using depth-first search. Once entities and relationships have been identified, graph creation proceeds as normal.

Search may also be required for database or file datasources. If only a subset of the file or database is to be included, the subset can usually be defined as the closure obtained from a given starting point; and this closure can be determined through search.

Filtering

Clover uses a simple filter model, with the filters located at the same pipeline position as in Huang, Eades and Lai's approach depicted in 5.1 (*b*), that is, before clustering is performed. This placement makes filtering entirely invisible to steps further down the pipeline; these steps will be unaware of wether filters, or filter chains, have been applied..

Since hierarchies are built and rebuilt dynamically from the results of the filtering stage, care must be exercised when sharing cluster hierarchies among graphs that have been filtered differently. Therefore, Clover hierarchies are only shared as long as the filter is the same. When a view switches to a different filter, the hierarchy is recreated, and from then on will remain completely independent of any other hierarchies.

6.2.2 Cluster Hierarchies and Clustered Graphs

Clusters are nested collections of vertices from the base graph. Being nested, each cluster forms the root of a tree, where the leaves of the tree are vertices from the base graph, and the intermediate nodes are other clusters. At the root of the hierarchy is a cluster which contains *all* leaf vertices. This cluster is referred to as the *root cluster*.

Section 5.2.2 includes a comparison of the performances of several clustered graph internal representations. The use of additional data structures can dramatically increase the efficiency of operations on such structures. Clover uses a variant of the **Buchsbaum-naive** approach: each cluster vertex u keeps hashed lists $N(u)$ of the real (ie., not induced) edges between its direct children, and also lists $\mathrm{IN}(u) = \{(v, w) : w \in desc(u)\}$ and $\mathrm{OUT}(u) = \{(v, w) : v \in desc(u)\}$ of all incoming and outgoing edges where either target or source are descendants of u. Additionally, each cluster vertex u keeps a hashed list of all its leaf vertex descendants. The value of this tradeoff from an efficiency point of view not clear; it has been included only to facilitate implementation (see Appendix B.2.5), since it can be used to provide very fast answers to containment questions. The list of leaf vertices accounts for the factor $|V| \cdot D$ found in table 6.1, which extends table 5.1 with the space and time complexity of this work.

The use of $\mathrm{IN}(u)$ and $\mathrm{OUT}(u)$ makes for extremely simple implementations of `edgeQuery` and `edgeReport`(u, v): simply calculate the set intersection $\mathrm{OUT}(u) \cap \mathrm{IN}(v)$. Since set intersection using hashes is almost linear, complexity does not suffer significantly. However,the implementation of clustered graphs has been chosen more for simplicity than for efficiency, and the design is modular enough to allow a full replacement of the structure should it prove too expensive. The time requirements of all other operations are on par with more complex implementations.

	Raitner	This work
Additional space	$O(\|E\|\ D)$	$O((\|E\|+\|V\|)D+log^2\|V\|)$
`edgeQuery`(u,v)	$O(log\|V\|+k)$	$O(\|\{(w,z) : w \in desc(u)\}\|)$
`edgeReport`(u,v)	$O(log\|V\|+k)$	$O(\|\{(w,z) : w \in desc(u)\}\|)$
`expand`(v)	$O(Opt_{G_C}(v)\ D\ log\|V\|)$	$O(Opt_{G_C}(v))$
`collapse`(v)	$O(Opt_{G'_C}(v))$	$O(Opt_{G'_C}(v))$
`newEdge`(u, v)	$O(D\ log\|V\|)$	$O(D)$
`deleteEdge`(u, v)	$O(D)$	$O(D)$
`newLeaf`(u)	$O(D)$	$O(D)$
`deleteLeaf`(u)	$O(D)$	$O(D)$

Table 6.1: Space and time complexity of clustered graph update in CLOVER, compared to that [109]. D is the depth of the hierarchy tree, $s = min\{D, log|V|\}$, and k is the size of the output for the `edgeReport` operation. **Raitner** is described in [109]. This table is an extension to table 5.1

Generating a Hierarchy

Since the domains where CLOVER has been used so far (particularly AH) did not require sophisticated network analysis, the default clustering algorithm has been kept simple, using a rule-based clusterer which can be either subclassed to refine the default rules, or completely exchanged for any other algorithm. The main advantages of a rule-based system are

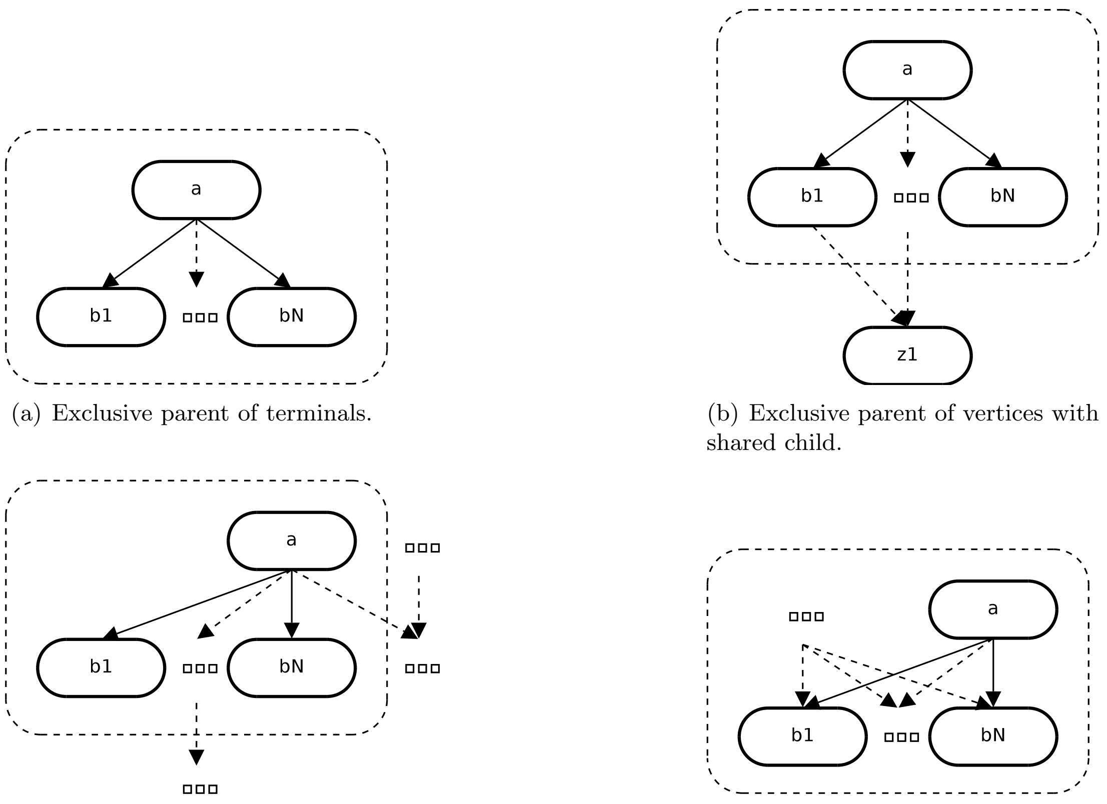
- Simple design and easy to understand – Operates as a transformation grammar, and particular rules can be easily described. In the default setting, CLOVER uses only 4 rules to generate graph clusterings. They are designed to generate a tree-like hierarchy if the input graph is a tree.

- Ease of customization – A particular application can easily define a set of rules that will treat certain vertex types and edges in a domain-dependent way.

Hierarchy generation does not need to be monolithic; indeed, rule matching is performed incrementally: rules are executed on partially clustered versions of the base graph, and each successful rule execution results in a partial clustering with less vertices, until finally the root of the hierarchy has been reaches. This incremental clustering approach can also be used to chain several clustering algorithms one after another; for instance, if a framework user decides to implement a more general clustering algorithm, a rule-based clusterer could still be retained to be used as a first preprocessing step.

The default clustering algorithm itself uses the four rules illustrated in fig. 6.3. These rules are expected to yield a tree-like clustering for tree-like graphs, because an initial objective of CLOVER was to provide tree-like expansion and collapse for general graphs. The algorithm executes each rule, starting from the highest-priority one (priority is $(a), (b), (c), (d)$, until no more matches can be found, and only then tries the next rule.

Whenever a match is found, search is restarted from the highest-priority rules. This algorithm is not efficient, and could be greatly accelerated by avoiding repeated tries to match a rule against the same subgraph again and again. However, no such optimization effort has been attempted. Implementation details can be found in B.2.8.



(a) Exclusive parent of terminals.

(b) Exclusive parent of vertices with shared child.

(c) Exclusive parent of some vertices.

(d) Fallback rule.

Figure 6.3: Rules used in default clustering engine. Three small squares represent "zero or more vertices", and dashed arrows represent "zero or more edges". If a rule matches, vertex a will be used to label the cluster, composed of $\{a, b_1, \cdots, b_n\}$. The order of application is important, and strictly enforced.

Tree-like clustering for trees can be achieved using only rule (a). Further rules are used to accommodate graphs that deviate from strict hierarchies: rule (b) collapses chains and small forks that merge immediately, rule (c) allows clustering of "children" vertices as long as they are exclusive children to the cluster representative, and rule (d) is intended as a fallback if no other rule can be applied.

The end result of the clustering process is a tree superimposed on the original graph (again, note that this "original" graph may instead be a filtered version of the true base graph; clustering should not be concerned with this). The tree not only establishes a hierarchy; cluster vertices can be as general as graph vertices, in the sense of representation flexibility, and can contain anything in the programmer's fancy; most obviously, special labels to denote what children they contain.

Hierarchy update

Hierarchies may need to be updated after a change to the graph on which they were built. Once notified of a change to the base graph, the clustering engine is requested to generate an updated hierarchy, and to annotate the differences between the old hierarchy and the new one in an "event". This event will be used to notify components further down the pipeline of the exact nature and extent of the change.

The default CLOVER clustering engine generates this hierarchy update event by performing a tree difference between the old and new hierarchies. The tree matching algorithm is presented in appendix B.2.6.

Some clustering algorithms can efficiently detect changes to their runtime behavior. For instance, if the sequence of rule matches is stored, a rule-based clustering engine could keep track of the rule matched in each moment, and locate deviations as they occur. This approach has not been implemented.

6.2.3 Clustered graphs

Clustered graphs are built by combining a base graph, a clustering hierarchy built upon this graph, and a *slice*. A slice is a cut-set set of hierarchy clusters; representing only these clusters as vertices with all required induced edges results in an abstracted view of the base graph.

Cluster vertices are assigned a distinctive color, a slightly larger size than normal vertices, and a label. The label is that of the first child cluster that was included when the cluster was originally created; this first child is the cluster's *representative*. In each the rules presented in fig. 6.3, vertices labelled as a were used as representatives. This choice of labels is not always correct; therefore, clusters can be manually named, and the name will persist when cluster hierarchies are saved and later restored.

An excellent discussion of the importance of correct hierarchy labelling when providing information scent and the problems that it brings up can be found in [66].

Visibility

Operations on a clustered graph include expansion of a cluster vertex (substituting a non-leaf cluster vertex for its immediate children) and collapse (the reverse operation). In CLOVER, both operations can be performed in optimal time (see 5.1).

Navigation through a clustered graph can be performed via expansion and collapse of vertex clusters; but it is cumbersome to perform repeated expansions and collapses. In addition, default cluster labelling may create serious difficulties when browsing for graph vertices that have not been chosen as representatives.

If a the name or id of a vertex is known, or retrieved in the application through an appropriate query mechanism, clustered graphs support a "make visible" operation that performs any required expansions and collapses to ensure that the selected vertex is made visible.

Degree of Interest

Another method to navigate the graph is to expand a certain neighborhood around the *Point-of-Interest* (*PoI*) selected by the user, and collapse clusters that are further away. This semantical fisheye lens was first proposed by Furnas in [65].

The implementation found in Clover uses a simple distance metric (number of hops in shortest path to the PoI) to calculate the *Degree-of-Interest* (DoI) for all visible clusters. When counting hops, breadth-first graph traversal can be used. DoI for cluster vertices is defined as the lowest DoI of any of their component vertices.

When the graph is displayed, clusters that are near enough to the current PoI will be expanded. Those that are further away will be collapsed. The algorithm is the following:

```
selectSlice PoI, frozen, focusSize, maxVisible
  calculate distances to PoI (= DoI for all vertices)
  collapse all clusters, excluding the PoI or frozen vertices
  ensure that a radius of focusSize around the focus is visible
  while the visibleVertices < maxVisible,
    find the most important cluster
    expand it
```

Where *frozen* vertices are those that have been marked by the user to not participate in automated expansions or collapses. This is particularly handy when exploring a far-away part of the graph without losing view of the current area: just "freeze" the vertices that should be preserved, and return to them later.

The algorithm results in a new slice. This slice is compared with the previous one, and the minimal set of differences is used to transform the previous slice into the new one, animating the transformation so as to preserve the user's mental map.

It is possible to speed up this algorithm, particularly the calculation of degrees of interest, by estimating an upper bound for the size of the set of "important" clusters that will need to be expanded. Calculating more degrees the degrees of interest of vertices that will not be included in this set is clearly unnecessary, and can greatly speed up execution for large graphs, and avoid the $O(|E|)$ memory consumption associated with breadth-first exploration.

6.2.4 Events and updates

Clustered graphs can change over time. User navigation requires expansion and collapse of clusters, which results in added or removed vertices and edges. Changes to a filter may result in cascaded updates to the clustering hierarchy, that should in turn be reflected in the final clustered graphs. Changes to the base graph, either due to user edits or update events received from the data source will require filters to be reapplied, with similar effects.

The *Observer* design pattern described in [67] is used throughout CLOVER to connect pipeline components to each other, while maintaining a very low degree of coupling between each component. The following events can be launched and received:

- *Structure events* reflect changes in the graph structure: vertices or edges may have been added or deleted, or attributes requiring a visual refresh may have occurred to one or more vertices or edges (for instance, vertex label may have been updated). Structure events can be launched by all graphs, be they base graphs, filtered graphs and clustered graphs alike.

- *Hierarchy events* reflect changes in the hierarchy. They may be triggered for lower-level structure events (for instance, new graph vertices have been introduced, or revealed by a different filtering), or may reflect user intervention on the clustering, or the application of a new clustering algorithm that changes the current clustering. Hierarchy events are only launched by a cluster hierarchy.

- *Clustering events* are the final results of the above events, and can also be triggered from the user interface during graph navigation. They reflect changes in the level of visibility of the hierarchically clustered graph. A vertex expansion or collapse would fall into this category. Clustering events are launched by clustered graphs.

If an edge were to be removed from a base graph, a structure event describing the change would be generated. This event would be sent to all filters, which would process it, and those with filtered graphs that were affected would launch the corresponding structure events to their own listeners. Hierarchies defined on top of any of these graphs would reevaluate their clusterings, and report any change as a hierarchy event to each of the clustered graphs registered to receive them. A clustered graph receiving a hierarchy change would evaluate it to check for any necessary changes (including, if the focus was affected, focus changes). Finally, a clustered graph which had changed would send a structure event describing the nature of the change, and one or more clustering events describing the associated expansions and collapses, to any interested components (probably animators in charge of maintaining the corresponding graph views).

Expanding and collapsing clusters

Cluster expansion and collapse results in two types of events. As expected, a clustering change event is delivered to observers. Furthermore, since a clustering change involves addition and removal of edges and vertices, a structure change event is also generated. However, this change does not correspond to alterations in the underlying base graph – it is only due to clustering. Therefore, structure changes generated by a clustered graph due to clustering changes are annotated as such, allowing observers that are only interested in "real" changes to ignore them.

Clustering changes also include any possible updates to the point of interest. These updates are frequent during navigational operations, even if the semantical fisheye lens is not used. Expanding a cluster which was previously selected as PoI should pass the PoI status to one of its descendants (currently, the "cluster representative" is selected for the honour). Likewise, collapsing a cluster that contained the PoI results in the newly-collapsed cluster becoming the new PoI.

Cluster expansion and collapse does not generally occur in isolation. It is common for slice changes in a clustered graph to require several expansions and collapses. Clustering events contain the whole sequence of expansions and collapses that have taken place, enabling observers to "see" the whole slice change, instead of the piecemeal operations that compose it. Keeping related changes together is important to help a user understand the intent of the change.

Changing the hierarchy

The generation of hierarchy change events has already been explored. However, their interpretation when received by a clustered graph using this hierarchy merits discussion of its own. Since Clover's clustered graphs observe only their hierarchies, and not the graphs they are based upon, hierarchy changes contain information both on the changes performed to the cluster hierarchy and the related graph changes.

A clustered graph must evaluate each of these changes to evaluate, given the current visible slice, whether alterations to visible vertices and/or edges are required. These alterations may involve a need to recalculate the whole slice. The algorithm used in hierarchy change processing follows:

```
clusteredGraphUpdate( hierarchyChange )

  for each vertex changed upstream
    if it is visible, update it

  for each removed cluster
    if it was not added elsewhere,
      remove it or any of its visible children
```

```
for each gap in the slice created by removals
  plug the gap

for each newly added cluster
  add the relevant descendants, if any, to the slice

for each added cluster,
  locate any edges that may need to be added

for each visible edge,
  if one or both endpoints is missing, remove it

for each newly introduced base edge,
  if the induced edge should be visible, add it

create a structureChange event with all the above changes,
deliver this event to all registered observers

update the PoI // may cause other events to be launched
```

A *gap* in a slice corresponds to an illegal state, and may arise due to removal of outdated clusters (clusters that belonged to the previous hierarchy, but not to the updated one). Slices should always be cut-sets of the graph hierarchy; "plugging the gaps" requires clusters without any representatives (neither ancestors nor descendants) in the slice to be located, and appropriate representatives to be added.

The last line in of the algorithm calls for an update to the point of interest. This can be required if, for instance, the previous PoI was removed from the graph. Since the PoI must be visible at all times, changing the PoI may require several expansions and collapses, triggering slice changes and the delivery of the corresponding events.

6.3 View

The final component in the view section of the CLOVER pipeline is a graphical component that displays a graph. The JGRAPH [1] library for graph representation. Use of this library provides a wide array of visualization choices, including the most usual types of vertex and edge decoration.

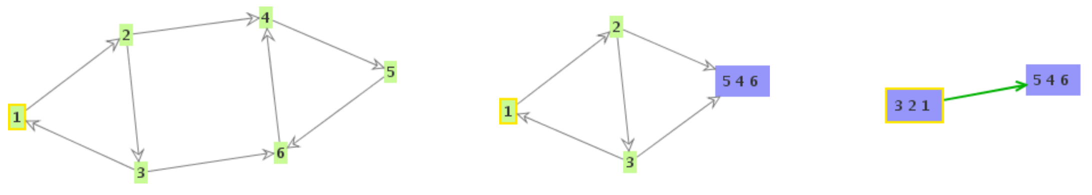
In order to bridge the gap between the final representation and the clustered graph model that is to be represented, each graph element (vertex or edge) must be provided with a visual representation and a screen location. The visual representation of individual elements is delegated to the *view graph*, while the screen location is determined by the visible portion of the overall graph layout. Changes to a graph do not reach the view immediately. Instead, they are collected by the animation component, where they are structured and gradually introduced.

The remainder of this section presents the main components of the view module: visual representation of graph vertices and edges, layout algorithms and strategies, and animation. Clover's incremental layout strategies and the animation support for structured view changes are distinctive features, not found in comparable systems.

6.3.1 Representation

The JGraph library used for vertex rendering offers a very rich set of shapes, arrow types, borders, label dispositions, and other element decorations. Support for curved or orthogonal edge drawings with multiple control points is also available, although Clover currently uses only straight-line edges. The view graph component can also customize the label that is to be displayed along with each element, specify a tooltip (which would be displayed when the mouse hovered over the relevant component), or refrain from labels altogether and use icons or rich components instead (albeit at a certain coding effort). However, this is generally not needed, since vertex and edge labels and tooltips can display basic HTML, including styled text, images and tables.

The series of sequential screen captures of figure 6.4 illustrates the display of a simple clustered graph with default settings. Intermediate animations have been omitted to save space.



(a) A fully expanded view.

(b) View after collapse of $\{1, 2, 3\}$.

(c) After collapse of $\{4, 5, 6\}$.

Figure 6.4: Example of clustered graph representation in Clover. Large, blue boxes are cluster vertices. Clustered edges are highlighted using thick, green lines. A yellow border marks the focus vertex.

Both clustered and unclustered vertex and edge representations are expected to be adapted to each particular application (see appendix B.2.13 for details). However, the default should allow easy distinction between clustered and "normal" edges and vertices, and the use of a distinctive yellow border for the point of interest is expected.

6.3.2 Layout

Although Clover delegates large parts of abstract graph manipulation and visual graph representation to external libraries, no appropriate libraries have been found for graph layout. Fortunately, layout algorithms are generally well described in the literature, and incremental layout can be achieved with force-directed layouts, which are simple to tune and implement.

Graph layout is not a monolithic process; instead, several layout algorithms are chained to achieve a visually pleasing final result. A small layout framework is included in Clover, providing algorithms suitable for initial vertex placement, force-directed placement from those initial positions, vertex overlap prevention, and connected-component separation, among others. The layout framework supports pluggable algorithms (which implement the *Strategy* design pattern described in [67]), making it easy to tune for different requirements. Details can be found in appendixes B.1, B.2.19, B.2.20 and B.2.22.

Users require a considerable time to come to grips with a completely new layout, but can integrate small, incremental changes almost immediately. The surrounding layout itself provides a powerful sense of context, referred to in section 3.2.2 as the *mental map*. Mental map preservation is a key concern in a clustered graph interface. In addition to incremental layout and animation, Clover also uses layout caches and a layout history to assist in mental map preservation.

Layout is also a computationally-intensive process, and incorporating layout into an interactive graph application makes it a potential bottleneck. Indeed, one of the main benefits of using clustered graphs is that, if the number of simultaneously visible vertices is kept low, layout *can* be performed within interactive time constraints. In order to achieve fast layout speeds, Clover uses an alternative graph representation for layout purposes, providing fast and simple access to vertex positions and boundaries. This alternative layout-oriented representation presents a secondary benefit of decoupling layout from representation. This allows layout changes to be introduced smoothly via animation.

Interactive application time constraints can result in layouts that do not meet a user's aesthetic requirements. Users request additional layout to be performed, or perform manual adjustments to the current layout. Clover attempts to preserve any manual layout changes during further navigation.

Layout algorithms

Figure 6.5 provides an overview of the layout process used in Clover. Each solid, rounded box represents a different layout algorithm. Layouts are chained together into logical phases: overall vertex placement, compactification, and overlap removal. Initial layout is performed only once, when the graph is first displayed. Incremental layouts are performed afterwards, with emphasis on minimizing displacement of non-changed vertices.

The following list provides a description of the layout algorithms used in Clover

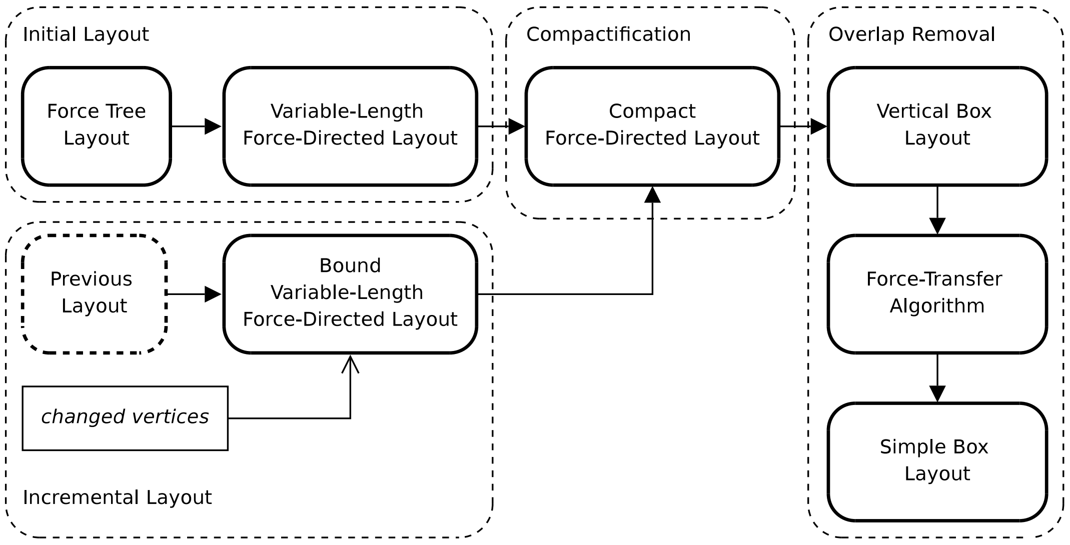


Figure 6.5: Layout in CLOVER. Incremental layout and initial layout undergo different vertex placement. Compactification and overlap removal are common to both. Solid, rounded rectangles represent algorithms.

and their purpose.

- *Force-Tree Layout* – Selects a spanning tree of the graph, and performs layout of this tree using a simple one-pass force directed placement: once the root vertex is placed, each additional vertex is placed at a fixed distance from its parent in the spanning tree, away from the repulsive force exerted by other vertices. Each previously-placed vertex at distance d exerts a force proportional to its distance:

  $$f_r(d) = \frac{1}{d^2}$$

  This layout is designed to untangle the graph prior to force-directed layout. It is particularly efficient when untangling tree-like graphs. Although the idea of using a tree layout to seed further layouts is certainly not new, this particular implementation is, to the best of the author's knowledge, novel.

- *Variable-Length Force-Directed Layout* – a force-directed layout with support for edges of variable length. This is achieved by annotating each edge with a "strength", and decrementing edge length according to this strength. The algorithm is a variation of the well-known Fruchterman-Reingold layout [64] described in section 3.1.1; edge attraction for an edge with strength s_e is computed as if the edge length were $d' = d \cdot s_e$ instead of d.

  Simulated annealing is performed using an exponential decay formula, with a constant k_c chosen so that the minimum temperature t_{lo} will be reached at the last iteration, n:

$$k_c(n) = \left(\frac{t_{\text{lo}}}{t_{\text{hi}}}\right)^{1/n}$$

This algorithm is used to perform the brunt of the layout on graphs without previous vertex positions. As most force-directed layouts, vertices are treated as points, and no provision is made for possible overlaps. Resulting layouts for large graphs also tend to be very sparse, with large, empty areas.

- *Bound Variable-Length Force-Directed Layout* – This is a variation of the previous layout; a previous layout is available, and mental map preservation requires minimal changes to vertices that have not "changed" (have not been recently added or had edges added or removed). To keep vertices close to their previous positions, an additional force, f_p, has been added. This force is calculated as $f_p = k_p\ d$, where k_p is a constant. High values of k_p can produce layouts that vary only minimally; but mental map preservation must be offset against layout quality.

- *Compact Force-Directed Layout* – this layout attempts to compact the results a previous layout, by using powerful attractive forces between edges. The vertex repulsion formula from Eades' original spring layout (refer to section 3.1.1) is combined with a very powerful attractive force. Again, edge strengths are factored into attractive forces, by using $d' = d \cdot s_e$ instead of d in their calculation. Edges with low strengths are allowed to stretch much further than those with higher strengths. An additional constant k_r is used to balance the repulsive force.

$$f_a(d') = \begin{cases} d'^2 - k^2 & \text{if} \quad s_e \geq 1, \\ d' - k & \text{if} \quad s_e < 1 \end{cases}$$

$$f_r(d) = \frac{k_r}{d^2}$$

This layout by itself would not be very effective, since repulsion is much too low to untangle graphs. It is only effective when starting from an initial layout such as those produced in the previous steps.

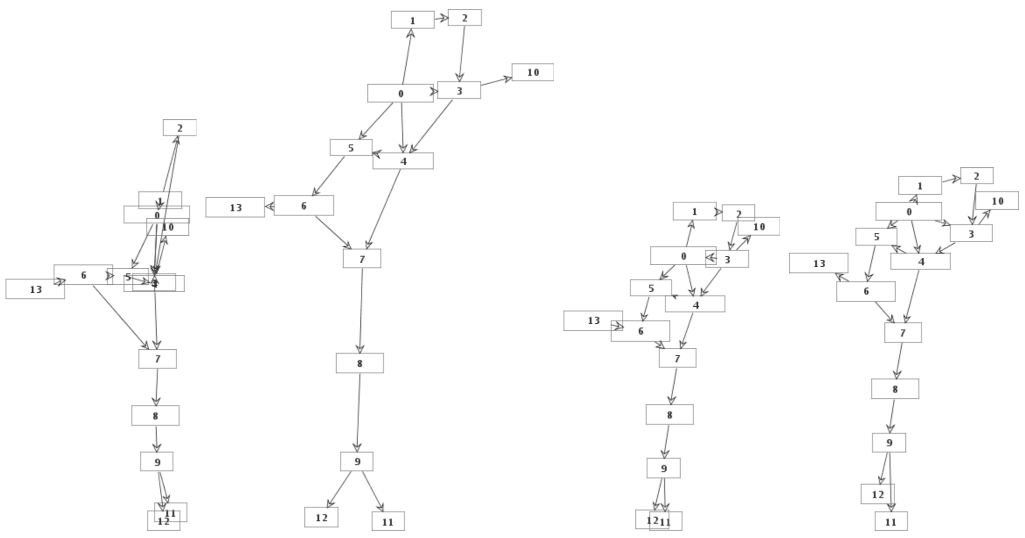
- *Vertical Box Layout* – All previous layouts avoid computing repulsive forces between vertices of different connected components. This layout separates all connected components by first calculating their bounding boxes, and then placing all components in a single vertical column. It is intended as a preparation step for the next layout.

- *Force-Transfer Algorithm* – Described in [83], this is a fast algorithm that eliminates vertex overlaps. Despite its name, no force model is used. The effects, however, resemble those that would be obtained using a more computationally intensive force-based overlap removal algorithm.

After this layout has ended, there will be no overlaps left in the graph. Run-time depends heavily on the number of overlaps that must be resolved, and wether or not resolving one overlap causes yet another one that will also require solving. Initial force-directed layouts should have eliminated most intra-component vertex overlaps at this point, and inter-vertex overlaps should have been entirely eliminated by a previous box layout.

- *Simple Box Layout* – Similar to the vertical box layout, bounding boxes are used to displace entire graph components. However, the goal of this layout is to place components next to each other, minimizing total graph layout space while avoiding component overlaps. An initial component (or "box") is first placed in the top-left corner of the layout area. For each additional box, its top-left corner is matched against the exposed corners of all previously-placed boxes, starting with those corners nearest to the top-left corner of the layout area. The first position where no overlap occurs is used.

  Although more powerful algorithms have been described in the literature, it is uncommon to work with multiple graph components at the same time, and filters could be used to hide components that were not of immediate interest. Therefore, a fast and simple approach has been preferred.



(a) Force-tree (b) Variable-Length FDL (c) Compact FDL (d) Force-Transfer

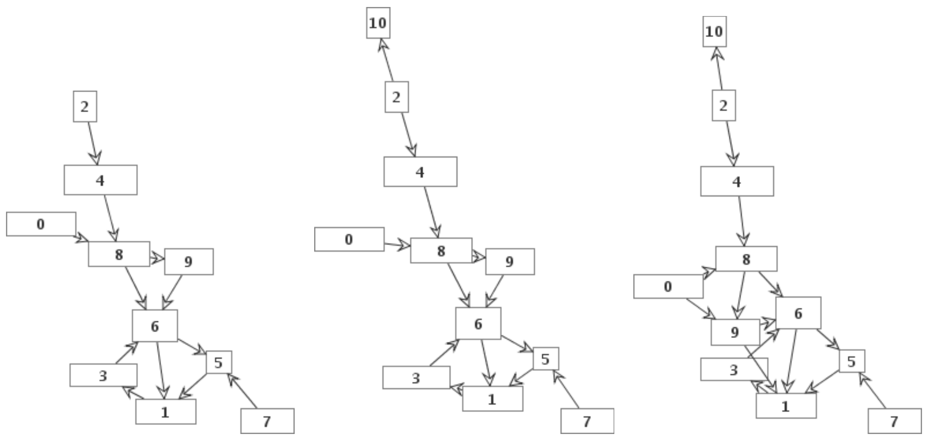
Figure 6.6: Layout sequence when no previous layout is available. Box layouts have been omitted.

Figure 6.6 illustrates the layout sequence for a new graph. If a previous layout had been available, the first two steps would have been substituted for an incremental layout. Note that layout compaction constitutes a tradeoff between aesthetics and size. Component overlap removal has been omitted from the sequence.

Incremental layout

If a previous layout is available, the layout sequence changes. Prior to the start of the sequence, a set of *free* vertices is constructed. This set includes all vertices that were not present in the previous version, and all vertices that have had edges added or removed. The *Bound Variable-length FDL* (BVFDL) behaves exactly like the *Variable-length FDL* regarding free vertices; but all vertices that are not free are considered to be *bound*. Bound vertices are anchored to their previous positions using springs of considerable strength, restricting their movement.

The use of springs to anchor vertices in-place has a secondary advantage; force-directed layouts tend to present "layout drift", where the whole components rotate slowly around an axis. Anchored vertices prevent such broad movements, which would otherwise require correction.



(a) Original graph (b) Added vertex 10 and edge 2→10 (c) Added 0→9 and 9→1

Figure 6.7: Incremental layout example. Layout from (*a*) to (*b*) uses a single run of the incremental force-directed layout algorithm; layout from (*b*) to (*c*) uses two, since changes are of greater significance.

When incremental layout is performed, the number of runs of the BVFDL is tailored to the extent of the changes. Small changes, such as adding a vertex connected to the rest of the graph with a single edge, require only one run. Broader changes, such as removing and adding several edges, require a greater number of runs to achieve a pleasant layout. A simple heuristic is used to select the number of runs according to the nature of changes. Figure 6.7 illustrates this behaviour with an example.

Layout cache

A layout cache is used to store each layout as it is generated. Manual changes to a layout overwrite the corresponding cache entry, allowing user changes to be preserved. The existence of a layout cache allows a lookup to be performed prior to any layout; if the destination graph already had a good layout in the cache, the stored version is used. A cache hit has two important advantages: mental map is preserved, and speed is greatly boosted, since lookup is much faster than actual layout.

A near miss can be almost as good as a cache hit; if the miss is close enough, incremental layout starting from the "best" match can be used, yielding a result that will be closer to the user's mental map and require less computation than if no cache had been used.

In order to compute sloppy matches, a text representation of a view graph is used. This representation relies on a string representation that includes the unique identifiers of all visible vertices and edges, in a canonical ordering. Sloppy matching counts how many vertices and edges are common to both graphs. Exact key matches can be located in $O(1)$ time, using hashes from the keys. Sloppy matches are currently much slower, requiring $O(N \cdot M)$ time, where N is the number of keys stored in the cache and M is the size of the keys to be compared, and therefore proportional to the number of vertices and edges in the visible graph.

A least-recently used (LRU) replacement policy is used within the layout cache, with the understanding that layouts that are seldom used are much less relevant towards mental map preservation than those that are accessed with a greater frequency.

Note that graphs themselves can change, outdating large numbers of cached layouts. Without sloppy matching, outdated layouts would be lost unless the changes were reverted. Sloppy matching allows reuse of layouts when the amount of changes is low. However, it is insufficient when dealing with more extensive changes. Possible approaches to deal with this problem are listed in section 9.3, as future work.

6.3.3 Animation

Within the detailed clover pipeline displayed in fig. 6.2, animation acts as a bridge between model and view. All changes to the model are animated prior to their representation in the view, possibly after performing incremental layout to ensure that the layout of the updated graph remains relevant.

Apart from this role within the main pipeline, animation is also a general mechanism built into Clover, and can be used to highlight or animate any type of alteration to a graph. The separation between these two roles can also be observed at the implementation level; refer to appendix B.2.16 and B.2.18 for details. Since pipeline-related animation is based on the general animation framework, the framework will be described first, and its use in mental map preservation via layout animation immediately afterwards.

The animation framework built into Clover is very similar to that of Piccolo, and partly inspired by it. Instead of Piccolo's *activities*, Clover uses *animation plans* – but the differences are not significant. In both systems, an animation is composed of a series of steps, and every step lasts a certain time interval. During this interval, steps are periodically requested to update themselves, affecting the display of one or more graph elements.

Plans, Moves and Steps

More formally, animation plans are composed of a sequence of *moves*, and each move contains one or more individual *steps*. All the steps within a single move are executed simultaneously, and should not conflict with each other. Moves themselves are executed in strict order, and last as long as their longest step.

Each step has a duration and a frequency (how often it is requested to alter the graph). Steps may perform changes to the graph; this is very useful when representing multilevel expansions and collapses, since the plan would encompass all transformations, and the visible graph is bound to change during the steps – it would be impossible to build a step to highlight vertex a vertex when this vertex is only visible for a few instants in the middle of the plan. Additionally, steps need not be fully specified before being added to a plan. Instead, they are configured before being called for the first time. Configuration can also involve a short burst of layout, or propagation of model changes to the view graph.

Typical animation steps include:

- Highlight: sets of vertices or edges are surrounded by a coloured border of increasing intensity. This is intended to draw user attention towards the highlighted items.
- Movement: shift a series of graph vertices from one position to another. Used when animating layout updates due to graph structure changes, or navigational expansions and collapses.
- Graph update: propagate sets of model changes to the view graph, and then perform incremental layout. The updated layout positions are used in an embedded movement step to animate the changes.

Because the intended use of the application is interactive, certain plans may cease to be necessary before they have finished executing: a plan may be a response to a situation that is no longer present, or the user may have requested an operation that requires a different animation to be presented. All plan steps can be immediately aborted when such a situation arises. For instance, imagine a plan triggered by hovering the mouse pointer over a certain node, with the effect of slowly highlighting a set of related vertices. Should the mouse move out of the area of influence, it would be expected that the vertices now in mid-highlight be left in their original state. It could also be the case that the user decides to click on this vertex, and this may trigger a totally different plan, which should also bring about the demise of the previous one.

A simple priority mechanism is used to ensure that higher-priority plans (for instance, user-initiated cluster expansion) displace lower-priority ones (for instance, a mouse rollover), and conversely, that no low-priority plan will run until any high-priority plans have finished execution. Plans with equal priority can either be appended to each other or replace one another.

The priority mechanism is also intended to prevent plan conflicts – if one plan expects to find a vertex, and another plan removes it first, one of them will fail. A more complex mechanism could be implemented. For instance, Piccolo associates plans to visual objects, and conflicts are therefore easier to manage. However, Clover's present priority system is sufficient for most animation tasks.

Animating structural graph changes

When structural changes are received by the animation module within the Clover pipeline, the view graph is immediately updated, removing old vertices and edges, and adding new ones. Initial layout of any new vertices requires them to be placed at a starting position. If the new vertices are connected via edges to any prior vertex, the position of one of these neighbors is selected as a starting position.

A short, incremental layout is then started. As described in section 6.3.2, the amount of layout time and the degree to which existing vertex positions will be altered depends on the extent of the changes to be introduced. This new layout is gradually introduced by moving each vertex along an interpolated path, from its old position (previous to incremental layout) to the destination as determined by layout results. Movement speed is not constant; instead, vertices start movement slowly, accelerate, and decelerate towards the end of the movement animation. Movements animations last 500 milliseconds; this requires vertices that move a large distance to move much quicker than those that move a shorter distance.

Friedrich and Eades [62] present arguments against the use of simple interpolation in graph animation. Their arguments are strongest if layout changes are truly significant. Incremental layout in Clover attempts to minimize changes, and therefore the techniques proposed in [62] have not been considered a priority, and are currently not included.

If further information had been available on the semantics of the change, additional highlighting could have been performed, emphasizing the exact nature of the change. Users of the Clover framework can easily define new behaviours for structural animations.

Animating clustered graph navigation

In the case of cluster expansion and collapse, additional information is made available through the use of cluster events (see section 6.2.4). This allows animation to highlight the semantics of the change, instead of only the end results.

Cluster events include sequences of cluster expansions and collapses that, when performed in order, lead to the same graph that would have been obtained through a direct structure change. The animation of cluster events is performed stepwise, in two phases:

- *Collapse* – Clusters that should be collapsed are collapsed in layers. Layering reflects the fact that clusters can contain other clusters, and it is desireable to allow collapse of a cluster only after all its children have been collapsed. Vertices that will be affected by a collapse are highlighted in red. After each layer collapse, a small incremental layout is performed.

- *Expand* – Cluster expansion is performed in layers. Again, layering obeys cluster inclusion; a cluster that is child of another cannot be expanded until its parent has been expanded. Clusters that are undergoing expasion are highlighted in green. After each layer expansion, a small incremental layout is performed.

Vertices that are undergoing collapse receive a red border, and move towards the point where their parent cluster vertex will be placed. Currently, collapsed clusters appear at the center of masses of their children's positions. Expansion is the reverse operation; the expanded cluster vertex is first substituted for all its children vertices, which are placed at the same position previously occupied by their parent. The newly-placed vertices receive a green border, and radiate outwards until they reach their final positions.

Figure 6.8 illustrates the progress of a layered cluster collapse. Figure 6.9 illustrates the animation triggered by a focus change. In both examples, a sample graph with artificial clustering settings has been used; in a real application, such a small graph would be totally expanded.

6.4 Control

Within the Model-View-Controller design pattern, control refers to user interaction, more specifically to how user input is obtained and processed. This section oversteps these bounds, and also attempts to explain how the previous steps of the Clover pipeline are integrated into the final interface presented to the user.

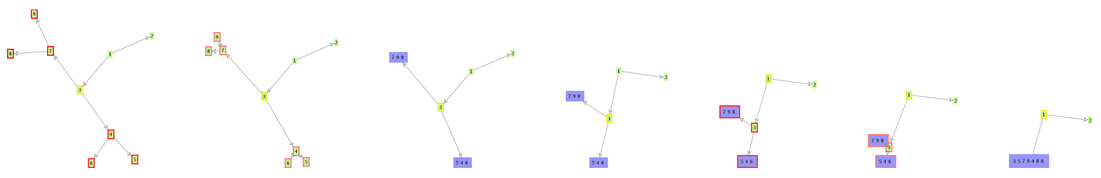


Figure 6.8: Animation of a cluster collapse. Red borders mark vertices which will not be visible once they are collapsed into their parent cluster vertex. Collapse proceeds in layers: children of clusters scheduled for collapse are collapsed before their parents.

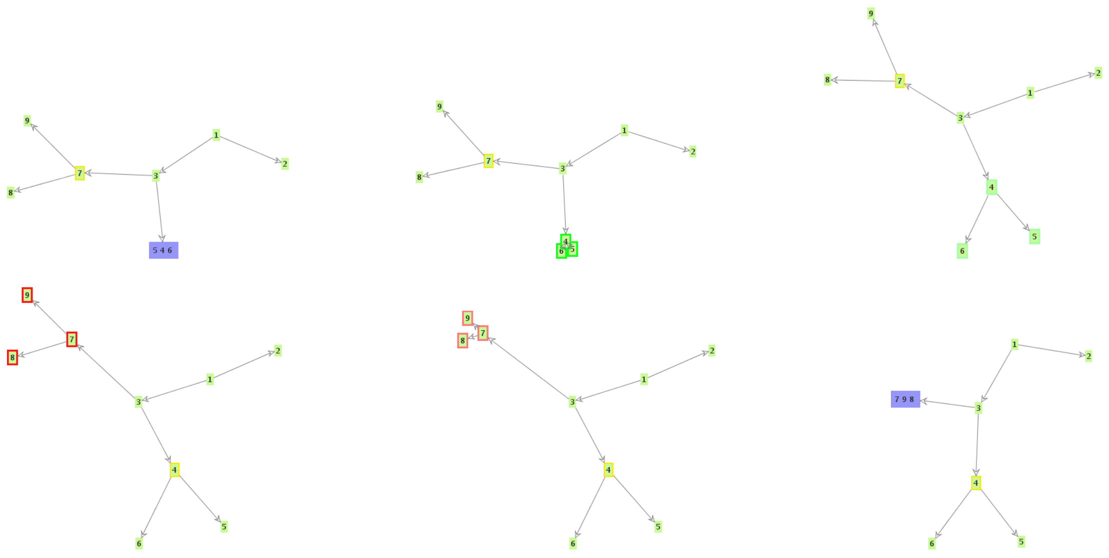


Figure 6.9: Focus change animation. Order is left to right and top to bottom. First, the focus is made visible; this requires an expansion. Expanded vertices are highlighted in green. Far-away vertices are then collapsed.

Figure 6.10 is a screenshot of this basic interface, using the same graph displayed in figures 6.8 and 6.9. The default interface is composed of a series of graph views and a single cluster tree, labelled as “Tree view” in fig. 6.10. All views are built upon the same base graph model, but different filters and hierarchies can coexist, yielding different graph and tree views.

The cluster tree displays an editable tree of the cluster containment hierarchy for the current graph view. The tree is editable; dragging and dropping a cluster into another, for instance, will result in a change to cluster containment for the current view. Highlighted nodes within the cluster tree correspond to visible vertices in the currently active graph. The tree can also be used to collapse or expand cluster vertices within the graph; for instance, a “make vertex visible” action can be triggered on any tree node.

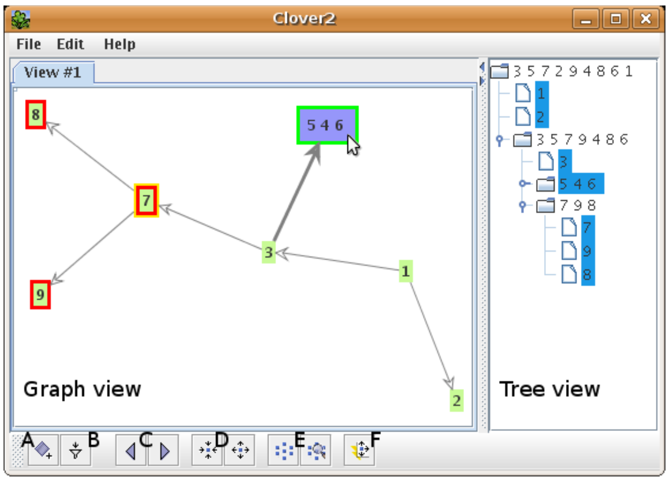


Figure 6.10: Basic Clover interface. Text in large, bold face is intended to guide this caption and is not part of the interface. A tabbed panel is used to hold graph views; the displayed view illustrates the aura that results when hovering the mouse over a vertex. An editable cluster tree displays clustering structure. Buttons labelled A-F correspond to actions; A: create view, B: change filtering, C: navigational undo/redo, D: decrease/increase number of visible vertices, E: relayout/configure layout settings, F: toggle automatic expansion. Coloured vertex borders and the bold edge connecting vertex 3 to cluster {5 4 6} are described in section 6.4.2.

6.4.1 Application and Actions

Clover makes heavy use of *actions*, reusable objects that can trigger a specific change to the model or view. As argumented in section 5.4.1, actions allow the same change to be triggered in many different ways. For instance, “perform incremental layout” is an action which can be invoked with a mouse-click on a toolbar icon, or by typing `Ctrl+B`. Certain actions may require parameters; to trigger “edit vertex”, the particular vertex to be edited must be specified beforehand.

Many actions need to share information. For instance, “paste” actions expect to obtain their operands form previous executions of “copy” or “cut”. Clover uses the base interface, depicted in fig. 6.10, to hold this shared data, which is common to all views. The interface contaions two main types of shared data: clipboard (used for cut/copy/paste operations) and editing history (enabling edit undo/redo). Navigation history is particular to each view, as is the layout cache.

Basic actions

A number of actions implementing common used operations are included as part of the framework. Applications built on top of the framework are expected to complement the default set of actions with domain-dependent ones. Applications are also expected to extend existing actions whenever possible. As argumented in section 3.2.3, extending the scope to which an operation can be applied results in enhanced usability. Therefore, unless an operation really makes no sense with a particular set of operands, it should be extended to accept those operands and yield expected results. This is particularly true of the "edit" and "cut, copy, paste, delete" set of operations.

The default actions, grouped by category, are the following

- *Cluster navigation* – Actions intended to alter the clustering, allowing clustered graph navigation.

  Toggle cluster lock Enable or disable the use of automatic degree-of-interest focus behaviour. When the cluster-lock is not active, changing the focus leads to DoI recalculation as described in section 6.2.3.

  Set focus Change the focus to the selected vertex. DoI recalculation will be triggered unless the cluster-lock is enabled.

  Make visible Expand or collapse clusters as needed to make the specified vertices visible.

  Toggle freeze Toggle the "frozen" status of a series of vertices. Frozen vertices do not participate in DoI-triggered cluster expansion or collapse.

  Visible vertex count increase / decrease Increase or decrease the maximum number of simultaneously visible vertices after DoI recalculation. Vertices exceeding this number and with low DoI will be collapsed into clusters.

  Collapse / expand vertex Collapse or expand the selected vertex cluster. Only cluster-vertices may be expanded, but collapse is interpreted as "collapse parent cluster vertex", and is therefore available for all vertices except the root cluster vertex.

  Undo / redo navigation Undo or redo navigation action; all navigational actions are triggered by clustering events. Storing these events in a navigation history allows navigational actions to be undone or redone.

- *View navigation* – Actions intended to alter a view, excluding cluster navigation and changes to the underlying graph structure.

  New view Instantiate a new view, as a copy of the current one (that is, using the same filter and clustering hierarchy). Changes to either cause an entirely new filte and/or hierarchy to be created, making the new view independent from the old one. All All views share the same base graph, and will update themselves when notified of any changes.

  Filter Alter the filter settings for the current view, causing the view and its associated cluster hierarchy to be updated.

Incremental layout Perform incremental layout on-demand, beautifying the current view.

Alter layout settings Change layout settings. Many layouts contain a large set of configurable parameters, including maximum layout iterations or total layout time.

Zoom / Pan Change the zoom factor and/or pan to a different location within the graph. For instance, center the view on a given vertex.

- *Editing* – actions that change the contents of the graph.

Edit Edit or view the properties of a vertex or edge; by default, only clusters are "editable", and editing is limited to renaming them.

Delete Delete a set of edges and/or vertices. Deleting a vertex also results in removal of all edges that used the vertex as an endpoint.

Cut Copy a set of edges and/or vertices to the clipboard, storing any additional information needed to enable future "paste" operations. Delete all cut items from the graph.

Paste Re-introduce the contents of the clipboard into the graph. If old positions had already been occupied, ask the user for remedial action, offering to create clones of already-existing graph components.

Copy Similar to *Cut*, but do not delete copied items after introducing them to the clipboard.

Undo / redo edit Undo or redo editing action; user-triggered (as opposed to datasource-triggered) editing actions generate graph editing events that are stored in an edit history, and can later be undone with these actions.

- *External* – actions that are external to the graph model and view.

Create new Create a new graph. The default implementation is to generate a random test graph; applications are expected to change this behavior.

Save / Load Save or load the current graph, its views, and all associated hierarchies, together with their layout caches. An XML dialect (see appendix B.3.1) is used to store all Clover-related information; applications are expected to store and maintain their own graph model.

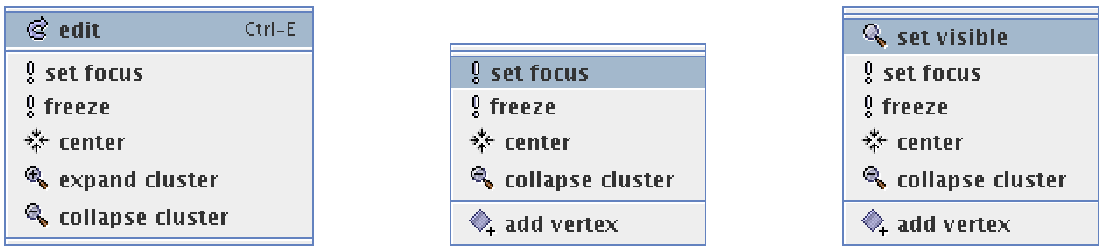
Print Print the current view in a printer or to a PDF file. This requires support from the underlying operating system.

Not all available operations have been wrapped as actions. In particular, infrequent operations and operations that are expected to be performed by direct manipulation, such as selecting or moving a vertex, have no corresponding actions. They are described in the "Interaction" subsection.

6.4.2 Interaction

Interaction with the graph is common to many other graph-based interfaces. With default settings, panning can be performed by using scrollbars, and zooming can be controlled with the mouse wheel. Vertices and edges can be selected either by clicking on them or by including them in the rectangle defined by a dragging gesture on top of the graph. Once selected, they are eligible for actions that requires one or more vertices or edges as operands, such as "copy" or "delete". When a selection is active, any other selection (including one that does not actually select anything) deselects previous elements.

As mentioned in section 5.4, most graph visualization applications delegate common interaction mechanisms such as zoom and pan to their underlying graph representation libraries. In the case of Clover, zoom and pan are provided by the underlying JGraph library. JGraph places greater emphasis on ease of use than on rendering performance, and therefore zooming is not as smooth as in ZUIs such as SHriMP[122]. Complementing JGraph's default zoom behaviour (which enlarges or shrinks all graph elements, including vertex bounds and contents), Clover includes an additional zoom setting, which affects only the layout positions of vertices, increasing or decreasing edge lengths without affecting any other graphical element. For zoom levels ≥ 1, layout zoom is used instead of graphical zoom.



(a) Visible cluster vertex.

(b) Visible, non-editable leaf vertex.

(c) Hidden, non-editable leaf vertex.

Figure 6.11: Contextual popup menus. The actions included in the popup depend on the edge or vertex for which the popup menu has been requested. Popups for non-visible vertices or can be launched through the cluster tree view.

Right-clicking on any element (edge, vertex or cluster-tree node) brings up a contextual popup menu, populated with suitable options as determined by the current application. Figure 6.11 illustrates popups for different graph elements using the default test application.

All interface elements, from vertices and edges to menu items and action-invoking elements, include tooltips. These small fragments of floating text appear after the mouse pointer has hovered over an element for a few seconds, and provide helpful advice on the the element's use, or in the case of data elements, their contents. For instance, tooltips for cluster vertices include, by default, a list of their contents.

Hovering the mouse pointer over graph edges or vertices also triggers simple animations. When hovering the mouse pointer over graph edges, the edge and its two endpoints are highlighted. Hovering over vertices highlights the vertex itself, and all outgoing and incoming edges. This type of highlighting is expected to be a useful visual aid when dealing with large graphs with long edges, or in the presence of edges that intersect non-endpoint vertices or other edges.

Clustered graph navigation

Clustered graph navigation has received special attention, since it is central to Clover. Two navigational modes are available, *automatic* and *manual*. The user can toggle between them by using the "cluster lock" action described above.

Manual navigation mode has the advantage of being straightforward: users must left-double-click on a cluster vertex to expand it, and shift-double-click on any vertex to collapse it into its parent cluster vertex. Both actions are also available from the corresponding contextual popup menus. In a sense, this is equivalent to classical interaction with tree interfaces: expansion and collapse must be triggered explicitly by the user.

Automatic navigation mode implements a semantical fisheye view as described in section 6.2.3. In this mode, changing the focus to any vertex (by left-clicking it once) causes visibility to be recalculated. Far-away vertices will be collapsed to make space for nearby vertices which will undergo expansion. Multiple mechanisms are available to control this operation; the user can set the desired number of simultaneously visible vertices, and the radius around the current focus where full expansion should occur. And, to prevent unwanted expansion and collapse of certain vertices, they can be explicitly marked as "frozen". Frozen vertices are not expanded or collapsed as a result of degree-of-interest recalculation.

To improve the predictability of automatic navigation mode, hovering the mouse pointer over a vertex triggers an animation that explains the results of shifting the focus to the hovered-over vertex (see fig. 6.10). Vertices that would be collapsed are highlighted with a red border, and vertices that would undergo expansion are higlighted in green – the same colours used for clustering collapse and expand animations. This *aura* mechanism allows users to prevent unwanted view changes by alter clustering settings or freeze vertices that should be unaffected before changing the focus.

During both manual and automatic clustered graph navigation, users can alway retrace their steps. Undoing a focus change or a cluster collapse is simplified by the existence of a navigational action history (its controls are labelled with a bold `C` in fig. 6.10). Requesting an "undo" of the last navigational action will revert any focus change and related cluster collapses and/or expansions. Because of the existence of a layout cache, no processing time will be required to recalculate old layouts; and the exact same layout will be used.

Recalling section 3.2.2, predictability, change minimization and traceability are all important parts of mental map preservation. The Clover framework uses incremental layout, animation, layout history, and predictable cluster navigation to deal with each of these issues.

Chapter 7

Application to Adaptive Hypermedia

This chapter presents WotEd, an adaptive hypermedia course authoring tool for the Wotan system which has been implemented on top of Clover. Wotan [57] is an upgrade to the Tangow (**TA**sk-Based lear**N**er **G**uidance **O**n-**W**eb) adaptive hypermedia course system described in [40, 41].

WotEd uses the facilities provided by Clover to address many of the problems presented in section 4.4:

- Large course graphs can be visualized and manipulated, using Clover's support for clustered graph visualization. Clustering is performed automatically, and allows authors to adjust the degree of detail to concentrate on the part of the course they are currently working on.

- Full round-trip support is available. Legacy courses can be read and displayed, even if they did not contain any information on graph layout. If layout, clustering and filtering information is available, it can be used to restore all views to their previous state.

- Monitoring is built into the tool, addressing the testing and maintenance stages. Student progress can be tracked in real time, with full access to the contents of each user model. This mechanism can also be used offline, allowing easy analysis of log files-

The first section of this chapter describes the Wotan system, and the general structure of Tangow courses. WotEd, which stands for **WOT**an **ED**itor, is then presented. A final section presents an experiment that compares WotEd's clustered-graph interface to a functionally equivalent tree-based interface, and tracks improvements in WotEd's support of clustered graph navigation tasks.

7.1 Wotan and Tangow

Wotan is a significant upgrade to the Tangow adaptive hypermedia course system. It includes a superset of its functionality, and is designed to be extensible regarding adaptation methods and techniques. However, the default behavior is to emulate the previous Tangow version. Throughout this chapter, Tangow will be used to refer to the adaptation scheme, and Wotan will be used to refer to the actual system.

Wotan is a server-side web application. Several interfaces are available to users, depending on their *role*. A user with the *learner* role can take courses, whereas an *author* can create and edit courses, enroll learners into these courses, and monitor their progress. Users with the *administrator* role can add or remove users, and change their roles.

Since Wotan is a web-based application, users are expected to interact with it using a browser. However, a series of simplified HTTP-based interfaces have been opened to allow automated (non-browser) HTTP clients to perform simple operations. Figure 7.1 illustrates both uses. A learner is depicted as using a web browser to access course presentation, while an author can monitor the active course, or author a course using WotEd's internal HTTP client.

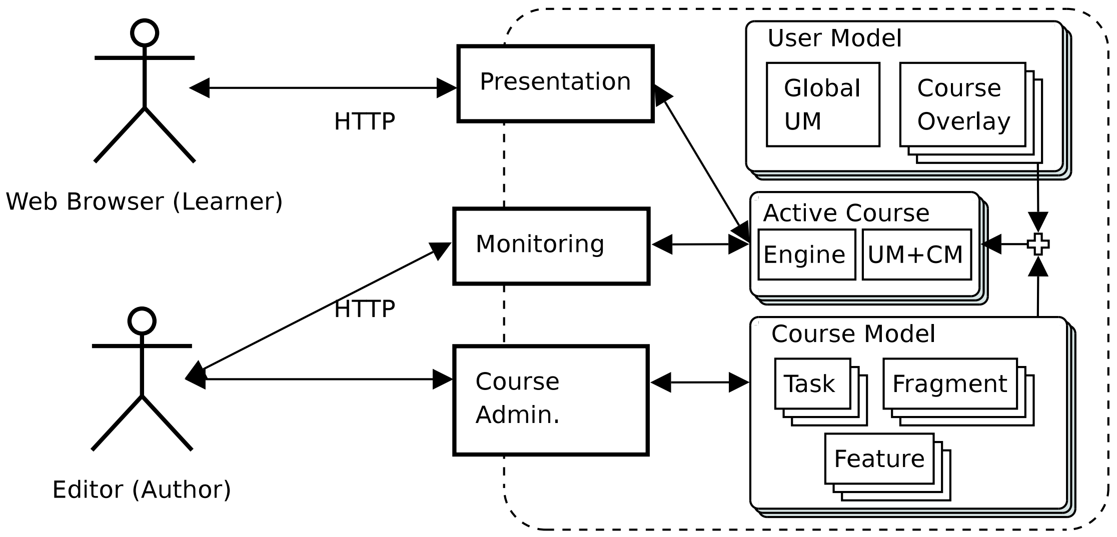


Figure 7.1: Wotan interfaces. Boxes with thick outlines represent some of Wotan's external interfaces, accessible via HTTP. Course delivery requires a course model, a user model, and an adaptation engine.

7.1.1 Tasks, rules and fragments

A Tangow *task* represents a concept. If it cannot be decomposed into subtasks, it is termed *atomic*. Atomic tasks only include an internal name, a user-visible title, a short textual description, and a list of one or more *fragments* that should be concatenated when presenting the task to a user. Non-atomic tasks, called *composite* tasks, also include one or more *rules* describing how they can be decomposed into further tasks. Since several composite tasks may use the same subtask in their decompositions, but no cycles are allowed, a TANGOW course can be seen as a directed acyclic graph.

A *fragment* represents a small, reusable content item. It can be referenced from different tasks, and can be reused throughout the course. Fragments are not directly representable. Instead, each fragment contains a list of alternative *fragment versions*, one of which will be chosen at run-time to represent the whole fragment within its task. Each version contains an expression to match against the user's UM. The adaptation engine selects the version which provides the best match, implementing adaptive presentation support.

Rules are used to decompose composite tasks. Each rule contains a list of subtasks to be used in decomposing the current one, and information on their sequencing: how many subtasks are required to consider the original composite finished? should users be allowed to skip from one subtasks to another?. Rule sequencing can require all subtasks to be visited in strict order (`and` sequencing), all subtasks to be visited, in any order (`any`), at least one subtask to be visited (`or`), and one and only one subtask to be completed (`xor` sequencing).

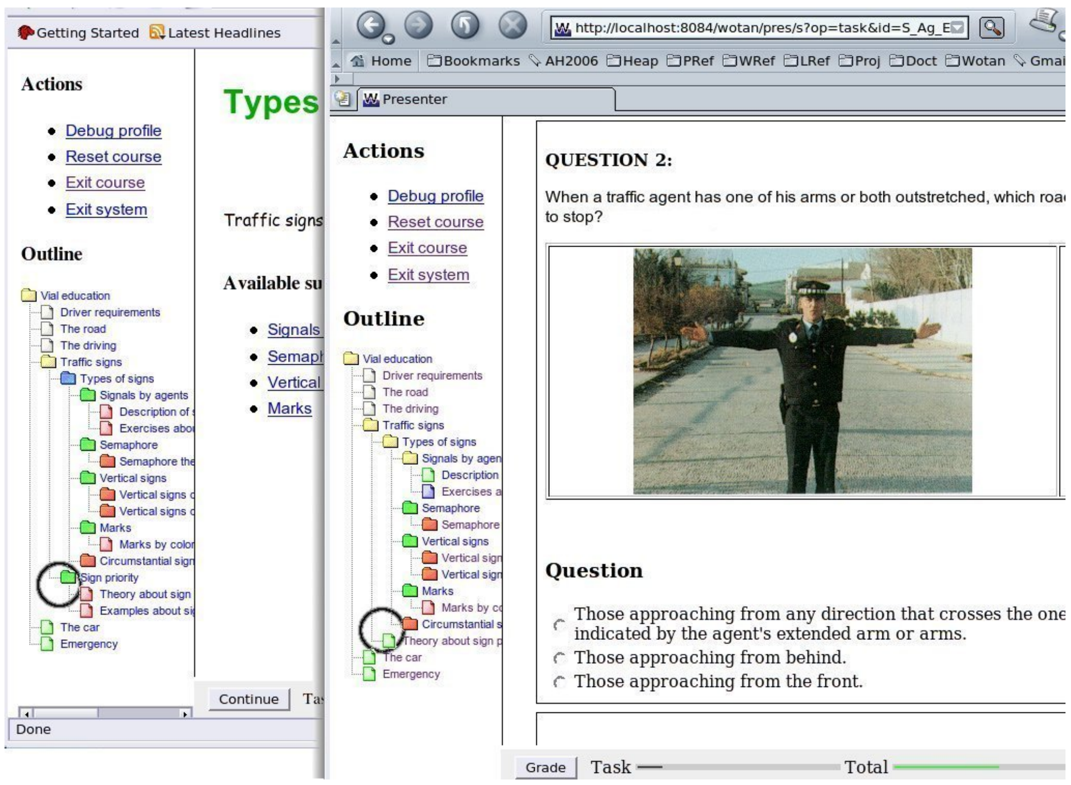
Additionally, rules can define conditions that must be met for the rule to be *active*. At most one rule may be active for any given task, and once a rule is activated, it can never be deactivated. Rule activation conditions are expressions on the user's UM, similar to those found in fragment versions, but of a boolean nature. Conditional rule activation implements adaptive navigation support.

Figure 7.2 illustrates the course structures perceived by two users. Due to different active rules, both structures are different.

7.1.2 User model

During course presentation, the adaptation engine annotates a copy of the course model with data on what is accessible, how many times it has been visited, what grade did a user achieve on a given exercise, and so on. The set of annotations form the *course overlay*, which forms an important part of the user model (UM). The course overlay and the rest of the UM are queried by the adaptation engine to decide on rule activation, parameter propagation (that is, modifications to the overlay itself), and any other adaptation needs.

At the end of each course session, the corresponding overlay is saved back into the UM. Several overlays can coexist, one per course. Whenever a user starts a new session on a course, WOTAN checks for a matching, previously saved course overlay. If an overlay is found, it is used to restore the user's session to the same state it had when the user left the course.



QUESTION 2:

When a traffic agent has one of his arms or both outstretched, which roa
to stop?

Question

- Those approaching from any direction that crosses the one indicated by the agent's extended arm or arms.
- Those approaching from behind.
- Those approaching from the front.

Figure 7.2: WOTAN course presentation. Each browser window contains a session, and has been logged in with its own user. Differences in rule activation for each user model result in differences in available tasks. The relevant portions of the task trees have been highlighted with circles.

When a user visits a course for the first time, no course overlay is available. In this case, a course-dependent questionnaire is used to prime the UM for that particular course. For instance, the questionnaire may require the user to specify goals for the current course or previous background in any of the course's subject areas. Questionnaires are built from a list of *features* specified within the course: the user is asked to set value for each feature, selecting among the possible answers.

Besides course-dependent overlays, user models also include a course-independent section, termed *global UM* in fig. 7.1. The global UM covers contact information, age, language preferences (multilingual courses are supported), and any other course-independent data that may be required by an adaptation engine. It is easily extensible through attribute-value pairs.

User model queries are constructed through the use of namespaces. For instance, to refer to the grade achieved by the current user in a task identified as `booleanAlgebra`, the user model would be queried using the key `map.task.booleanAlgebra.grade`. In this query, the `map` namespace selects the course overlay (as opposed to the `user` namespace, which covers the user's global UM), the `map.task` namespace refers to all task attributes (and not `rule` or `fragment` attributes), the task ID specifies the task, and the `grade` attribute name finally identifies the value to be returned. The exact set of attributes used for each course component, their valid ranges, and their update procedures are responsibilities of the adaptation engine.

In the default TANGOW adaptation engine, user model queries for adaptation purposes can be present in the following course elements:

- Rule activation conditions (boolean), to determine whether a given rule should be activated or not. If not specified, the rule is considered active.
- Rule parameter propagation (real), used when a composite task has been completed, to calculate how each of the rule's subtasks should contribute toward the value of the parent tasks' attributes. If not specified, averages values over all subtasks are used. For example,

  ```
  grade = map.task.s1.grade * .2 + map.task.s2.grade * .8
  ```

  would calculate the final grade as 20% of the grade obtained in `s1` and 80% of grade of `s2`.
- Task termination conditions (boolean); a task is not considered complete unless this condition is satisfied. The default is to consider a task complete as soon as all of its subtasks have been completed, with an average grade of 50% or more.
- Version selection (real), specified for each version of a fragment. The higher the result for a specific user model, the better the fit between that particular version and the user model. The version with the highest value is chosen as representative of that fragment when the fragment is presented as part of a task. The default is 0.5.

Several restrictions are imposed on TANGOW's user model. Assignment is only possible within rule parameter propagation, and is limited to parameter which must be propagated. Task termination and rule activation conditions are strictly monotonic: once set, they cannot be unset. These restrictions are intended to simplify course design and avoid possible "dead ends", where a user with a given UM can never finish a course. In spite of these measures, such courses can still be created. This concern is shared with many other adaptive hypermedia course systems; complex formal verification would be required to entirely avoid this risk.

Even after formal verification, a particular user may still encounter "weak" dead ends: course locations where it is not at all clear what actions are required to advance further. An effective approach to counter both types of dead ends is to use adequate course structure visualization to inspect the course for defects before they arise; and monitor the course during the first few runs to detect defects that may have been overlooked during the authoring phase.

7.2 WotEd

The WotEd editor is based on a simple mapping from course description to graph. WotEd can be used to open and edit any valid Wotan course, and provides full roundtrip support, even after editing with external tools.

A screenshot of the WotEd interface can be found in fig. 7.3. It is built on top of the basic Clover interface described in fig. 6.10, but has been extensively adapted to Wotan course authoring. For instance, a "task hierarchy" tree has been included in the right-hand pane.

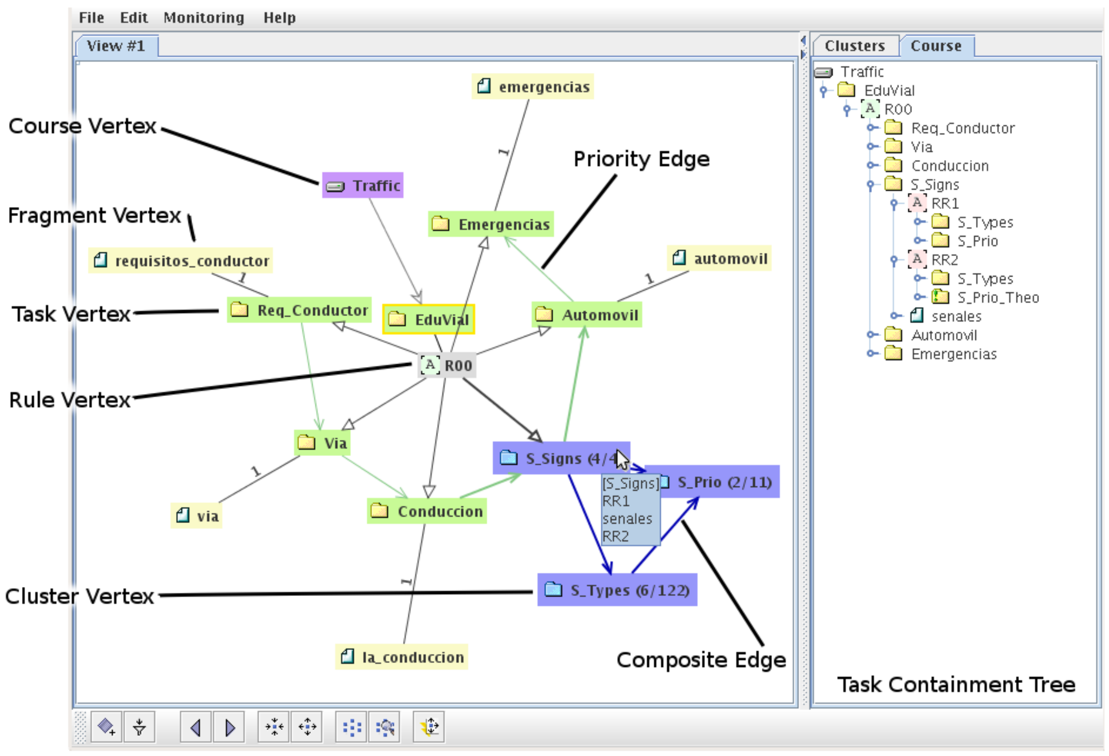


Figure 7.3: WotEd interface. Vertex and edge roles are identified by labels. Besides the default cluster tree, a second task containment tree has been included in the right-hand pane.

The task hierarchy provides an alternative display of the course's structure. It is similar to the task trees found during course delivery (fig. 7.2), an interface that all Wotan users are well acquainted with. Actions on vertices of the task hierarchy are treated exactly in the same manner as actions on vertices of the cluster hierarchy or on actual graph vertices. Users are free to mix and match interfaces according to the current tasks. Notice that, in the task hierarchy, the same task may appear under more than one branch; this was the initial problem that prompted work on a graph representation.

7.2.1 Mapping

The graph representation of a Wotan course includes a single vertex to represent the course currently being edited, and vertices for all tasks, rules and fragments. Fragment versions are currently not displayed as vertices, as they are not intended to be reused, and would always result in terminal leaves. Fragment versions are, however, visible in the task containment tree.

Graph construction is performed in two passes over the internal Wotan course representation, referred to as the *course model*. In the first pass, the course vertex and all task vertices and fragment are created. In the second pass, rules are added and *priority edges* are introduced between tasks. Priority edges are an artifact used to enforce consistent ordering of subtasks within a rule, since the order of subtasks is important for sequencing purposes. Even if `and` sequencing is not used (which would enforce strict ordering), subtasks will still be enumerated using their order-of-appearance within their rule when presented to users. The use of priority edges results in layouts that preserve this ordering.

Wotan course files are stored as (zip-) compressed archives. An XML file describing the course structure is located at the root of the archive, and all necessary fragment versions are stored within subdirectories. Since source code for Wotan is readily available, the course model to create the initial graph is loaded using Wotan itself.

The course model is kept during execution, and queried as needed to create forms that allow editing of course, task, rule, fragment and version properties. Any changes to the active course are first performed to the model. The model then triggers events to any listeners, one which include WotEd's task hierarchy tree and its Clover-based graph. When the graph receives a model update event, a suitable structure change event is generated, and the cluster hierarchy and all graph views are updated.

Clustering a Tangow course

Cluster hierarchy for a Tangow courses uses the same rule-based clustering engine described in section 6.2.2. An additional rule has been added, with a priority higher than all others. This rule ensures that each tasks will be head of a cluster that contains all its exclusively-owned fragments and rules.

The custom clustering engine also ensures that only "structurally significant" edges are considered during clustering. That is, only course→task, task→rule and rule→task edges are considered. If the course graph is a tree when only these edge types are considered, the resulting cluster hierarchy will be equivalent to the task containment tree.

Cluster labels consist of the name of the first vertex in the cluster (another cluster, a task vertex or the cluster vertex), followed by an indication of the cluster's size. Cluster size is shown as a fraction x/y, where x is the number of child vertices in the cluster, and y is the total number of leaf vertices that are subsumed below this cluster. For instance, the cluster vertex pointed to by an arrow in figure 7.3 is labelled as `S_Types (6/122)`; the first task it contains is named `S_Types`, there are 6 immediate children, and 122 total leaf vertices underneath.

Cluster names (and contents) can be changed manually. Changing a cluster name will affect the textual part of the label, but numbers will still be shown to provide an idea of the size and depth of the cluster.

Vertex and edge types

All vertices have been assigned a color code and an icon, depending on their roles. The course vertex is represented in a violet color, and uses a disk icon. Tasks are represented by yellow folder icons, and are colored green. Fragments are represented by a document icon, and are colored yellow. Clusters are colored dark blue, and use a blue-tinted task icon, because their label usually corresponds to a task.

The same icons are used to represent each element in the task containment tree. Fragment versions, not represented in the graph view, are given a darkened document icon in the tree view. Rules receive an icon composed of a single letter and a colored background. The letters used in the icon represent the rule's sequencing type. Letters for `and`, `any`, `or` and `xor` sequencing types are, respectively, `A`, `a`, `X`, and `O`. If an activation condition has been defined, then the icon background will be colored red. Otherwise, the background will be light blue.

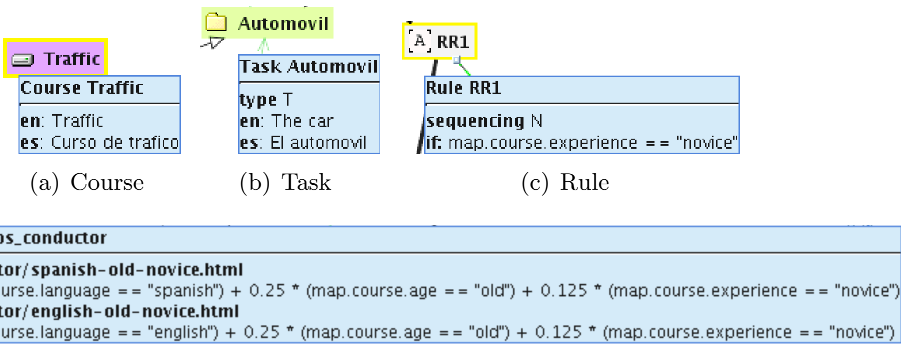
Edges are also color-coded. Each edge receives a different color and arrow head depending on its type, as determined by its two endpoints. When a cluster-to-cluster or cluster-to-vertex edge contains several smaller, internal edges, a dark blue "cluster edge" is used instead. Additionally, edge lengths are variable. For instance, task-to-rule edge lengths have been made much shorter than others, forcing rules to stay close to their parent tasks.

Custom edge and vertex type rendering is achieved by extending the default CLOVER view graph, and specifying the decoration for all graphical elements. Tooltip support is also provided at the same level; figure 7.4 illustrates tooltip support for each vertex type.

7.2.2 Interaction

The file menu allows courses to be opened, saved and printed. Courses can be loaded an saved from or to local files. Using the built-in HTTP client, it is also possible to download and upload courses to and from a running WOTAN server. This requires authentication, and relies on the additional interfaces displayed in fig. 7.1.

WOTAN files may or may not have been created with WOTED. If the file was created with WOTED, it will contain a single `clover.xml` describing the last active workspace.



Traffic

Course Traffic

en: Traffic
es: Curso de trafico

(a) Course

Automovil

Task Automovil

type T
en: The car
es: El automovil

(b) Task

RR1

Rule RR1

sequencing N
if: map.course.experience == "novice"

(c) Rule

Fragment requisitos_conductor

requisitos_conductor/spanish-old-novice.html
prio: 0.5 * (map.course.language == "spanish") + 0.25 * (map.course.age == "old") + 0.125 * (map.course.experience == "novice")
requisitos_conductor/english-old-novice.html
prio: 0.5 * (map.course.language == "english") + 0.25 * (map.course.age == "old") + 0.125 * (map.course.experience == "novice")

(d) Fragment vertex

Figure 7.4: Vertex tooltips in WotEd. An example tooltip for a cluster vertex can be found in fig. 7.3.

This file is generated whenever a course is saved. If the course has been created with another editor, or imported from the old Tangow course database format, `clover.xml` will not exist. In this case, initial layout will be performed, and default cluster names will be used.

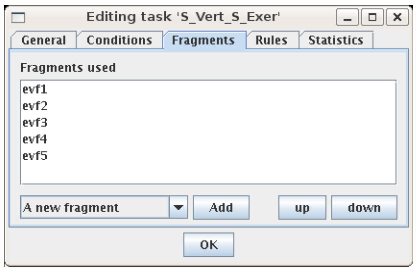
All default Clover actions have been extended to deal with WotEd graphs, triggering the corresponding events on the course model when launched. For instance, removing a task vertex may result in several cascaded deletions, depending on overall graph connectivity. This calculation is performed within the course model, and the results are reflected in all views.

Editing vertices

Editing (invoking the "edit" action) on any vertex will launch the edit form that corresponds to the vertex type. Sample edit forms for different vertex types are shown in fig. 7.5.

When an action on a form produces a change to the underlying model, this change is immediately reflected on all views. For instance, adding a new subtask to a rule using the a would result in the same model update that would be achieved if the same operation were to be performed directly on the graph. Certain structural operations can only be performed using forms. For instance, to change subtask order within a rule, the corresponding form buttons can be used. No equivalent operation can be performed directly on the graph, although drag&drop in the task tree would also work.

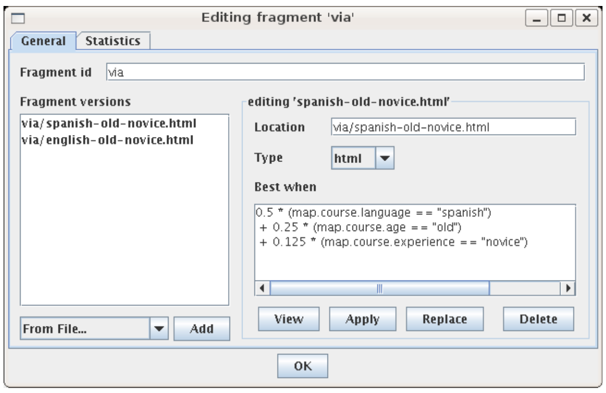
Forms are linked to each other. While editing a task with its associated form, double-clicking on a rule name causes the corresponding rule form to be displayed. The same is true fragment names, or for task names within rule forms. Double-clicking a fragment version is equivalent to selecting the fragment and selecting "view fragment": a browser window is created, and the fragment version is displayed within this window.



Fragments used

evf1
evf2
evf3
evf4
evf5

(a) Editing task fragments



Fragment id: via

Fragment versions

via/spanish-old-novice.html
via/english-old-novice.html

editing 'spanish-old-novice.html'

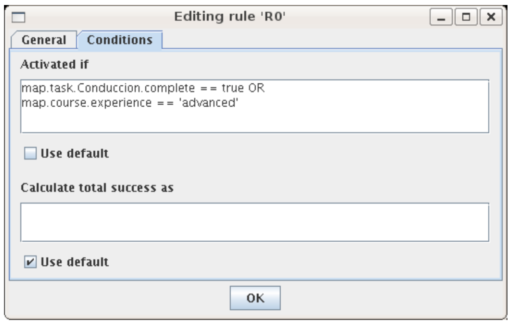
Location: via/spanish-old-novice.html

Type: html

Best when

```
0.5 * (map.course.language == "spanish")
+ 0.25 * (map.course.age == "old")
+ 0.125 * (map.course.experience == "novice")
```

(b) Editing fragment versions



Activated if

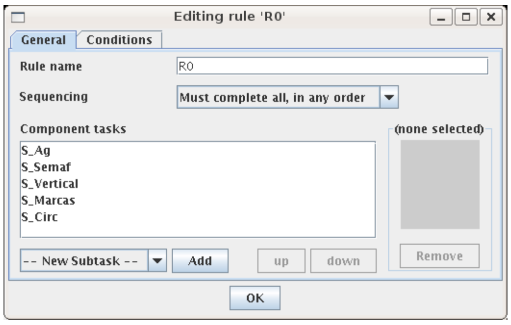
```
map.task.Conduccion.complete == true OR
map.course.experience == 'advanced'
```

☐ Use default

Calculate total success as

☑ Use default

(c) Editing rule conditions



Rule name: R0

Sequencing: Must complete all, in any order

Component tasks

S_Ag
S_Semaf
S_Vertical
S_Marcas
S_Circ

(none selected)

(d) Editing rule subtasks

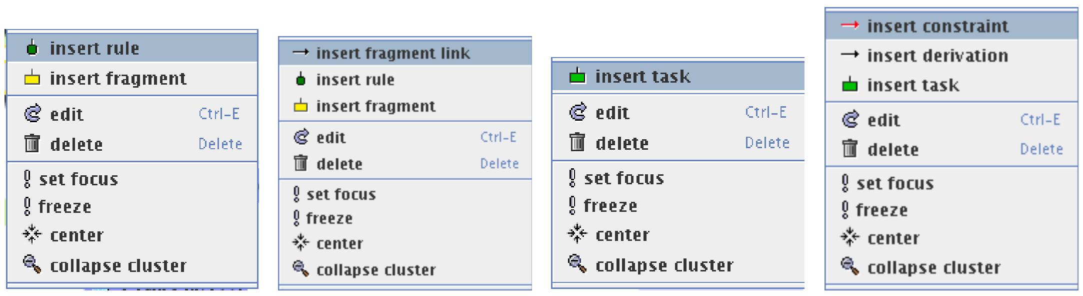
Figure 7.5: Forms in WotEd. Tabbed forms are used to edit the properties of each course element. Some of them also allow structural course changes to be performed; these are immediately reflected in all views.

WotEd incorporates a very basic fragment version editor, which can be used to generate simple HTML fragments that do not include images. More complex fragments must be imported from an external editor. Import can modify image and other resource paths to make them local, and bundle local resources together with the fragment version. Import is also available for exercises, which can be created with Wotan's online exercise creation tool.

Contextual menus in WotEd

Contextual menus for all vertex types have also been extended to support common editing tasks. Courses can be rapidly prototyped by inserting "default" tasks, rules and fragments from contextual popup menus triggered through right-clicks. Edges can be created by first selecting the edge destination, and then bringing up a context menu on the edge source. Figure 7.6 contains screenshots of contextual menus for different vertex types.

To simplify course prototyping via context menus (or even action shortcuts, which



(a) Task vertex (b) Task vertex, fragment selected (c) Rule vertex (d) Rule vertex, task selected

Figure 7.6: Context menus in WotEd for task and rule vertices. Menu contents vary depending on what is selected.

could be easily implemented), users are not required to fill in complete forms. For tasks and rules, the only the vertex name is required, as default values can filled in for all other attributes. In the case of fragments, a default version is created, using a simple HTML template; the user is then prompted to enter a fragment name and contents for the template. Any default values introduced for vertices generated in this manner can be modified by editing the vertices through their corresponding forms.

7.2.3 Monitoring

Authoring a an adaptive hypermedia course does not end when the first students log in. In a large, complex course, initial versions are likely to contain errors and omissions. Even when most errors are removed, requirements may change, and any update can cause unintended consequences. In order to track down problems with an adaptive hypermedia course, several approaches are possible. For instance, users may submit problem reports to course authors, or authors may resort to logfile analysis. A more proactive approach is to allow authors to examine student progress in real-time, with full access to each individual student's user model – ideally from within the same interface that was used to author the courses themselves. This approach, not found in any other AH system, has been integrated into WotEd.

During student monitoring, special vertices, representing monitored students, are visible inside the graph. Each "student vertex" is connected to the task that the student is currently visiting. Information on each user's model is available on request, in the form of tooltips. Other data, such as task availability and completion status for each student, can be queried directly from the graph. Monitoring as performed by WotEd has many applications, ranging from initial course debugging (authors can also monitor themselves) to actual student monitoring during initial validation phases. Since monitored sessions can be stored for later replay, this approach also allows a form of graphical log analysis.

To perform monitoring, three elements are necessary: the original UM at the point where the student begins the session, the course that is being visited, and the adaptation engine used to deliver the course. User model and course can be downloaded directly from a running instance of WOTAN. The default TANGOW-based adaptation engine is included as part of WOTED, since it is needed to perform filtering.

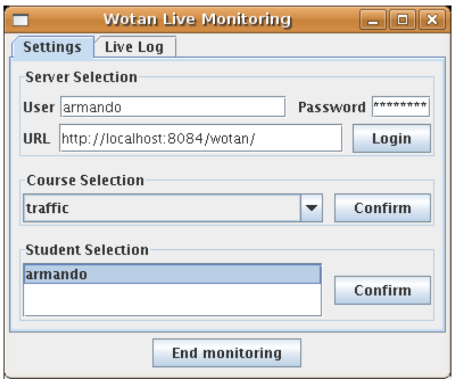
A monitoring API has been built into WOTAN. Authors can subscribe to username + course + action combinations, and will receive events whenever a monitored student performs an appropriate action on a monitored course. WOTED subscribes to these events, which are delivered via an open HTTP connection. Using a single long `multipart/x-mixed-replace` response, the server can keep the connection open for asynchronous replies in a protocol-conformant manner. Whenever WOTED receives a monitoring event, it is transferred to the active graph and simulated in the internal adaptation engine. This allows WOTED to synchronize the user models of monitored students with the actual UMs stored in the running WOTAN instance.

Events contain only user actions that result in adaptation engine updates. User models could have been sent and synchronized instead, but the chosen event-based approach is simpler to implement, and offers the added benefits of requiring a very low bandwidth and being easy to store for later playback. Figure 7.7 contains examples of both types of event sources: a live WOTAN connection on the left side, and a saved series of events on the right one. Playback control is only available when events are being read from a file.

Interface

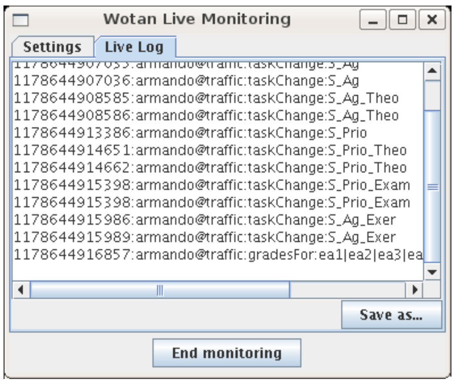
Tracking user progress is as simple as watching student vertices move throughout the graph, as depicted in 7.8, with further details available on demand. The graph is automatically expanded to ensure that all monitored student vertices remain visible. This is achieved by first ensuring that the vertices are visible, and then making them "frozen" so that they are never collapsed (see section 6.4.2 for details on the these operations).

Details on the monitored user models can be requested by hovering the mouse over tasks, clusters or monitored student vertices. When the mouse is hovered over task or cluster, monitored students are highlighted according to their current status regarding that task or group of tasks. If the task has been successfully completed, the student vertex is highlighted with a blue box. If the tasks are not available, red is used. If the task is available, but has not yet been started, the box will be green. When the mouse pointer is hovered over a student vertex, all tasks and clusters are highlighted using the same color code. See figure 7.9 for an example.



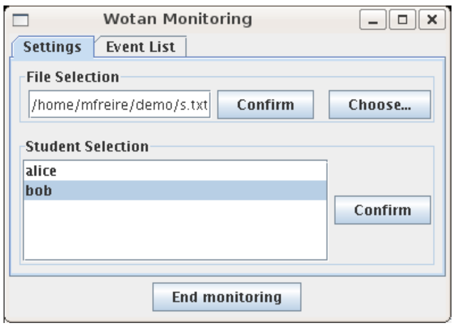
Server Selection
User armando
Password ********
URL http://localhost:8084/wotan/
Course Selection
traffic
Student Selection
armando

(a) WOTAN monitoring connection setup.



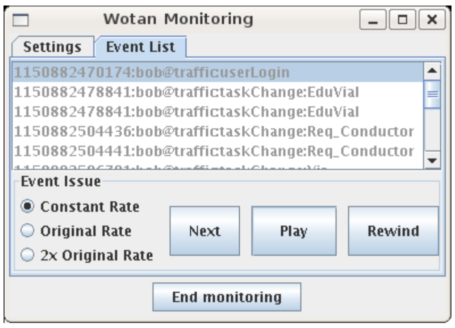
```
1178644907036:armando@traffic:taskChange:S_Ag
1178644908585:armando@traffic:taskChange:S_Ag_Theo
1178644908586:armando@traffic:taskChange:S_Ag_Theo
1178644913386:armando@traffic:taskChange:S_Prio
1178644914651:armando@traffic:taskChange:S_Prio_Theo
1178644914662:armando@traffic:taskChange:S_Prio_Theo
1178644915398:armando@traffic:taskChange:S_Prio_Exam
1178644915398:armando@traffic:taskChange:S_Prio_Exam
1178644915986:armando@traffic:taskChange:S_Ag_Exer
1178644915989:armando@traffic:taskChange:S_Ag_Exer
1178644916857:armando@traffic:gradesFor:ea1|ea2|ea3|ea
```

(b) Live monitoring event log.



File Selection
/home/mfreire/demo/s.txt
Student Selection
alice
bob

(c) Reading events from a file.



```
1150882470174:bob@trafficuserLogin
1150882478841:bob@traffictaskChange:EduVial
1150882478841:bob@traffictaskChange:EduVial
1150882504436:bob@traffictaskChange:Req_Conductor
1150882504441:bob@traffictaskChange:Req_Conductor
```

Event Issue
[✓ Constant Rate] Original Rate 2x Original Rate

(d) File event source controls.

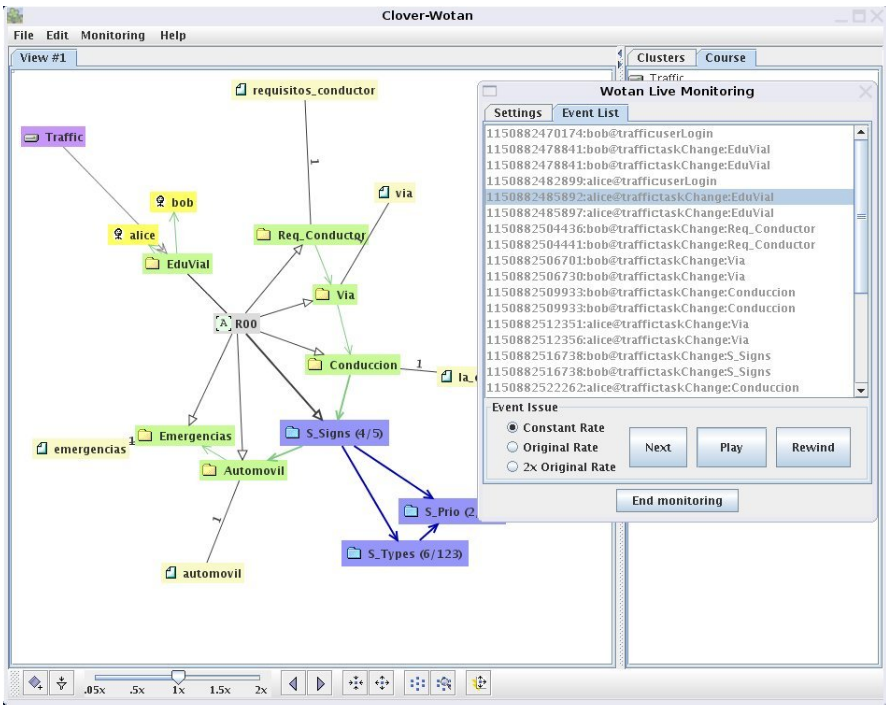
Figure 7.7: Course monitoring setup. Events can be received live from online students, or they can be stored in a file for later playback.

7.3 Experimental results

Two experiments have been performed on WOTED, targeted at comparing the ease of use of the graph interface versus the more familiar tree interface used in earlier versions of TANGOW course editing tools. These experiments are described in detail in [59] and [61].

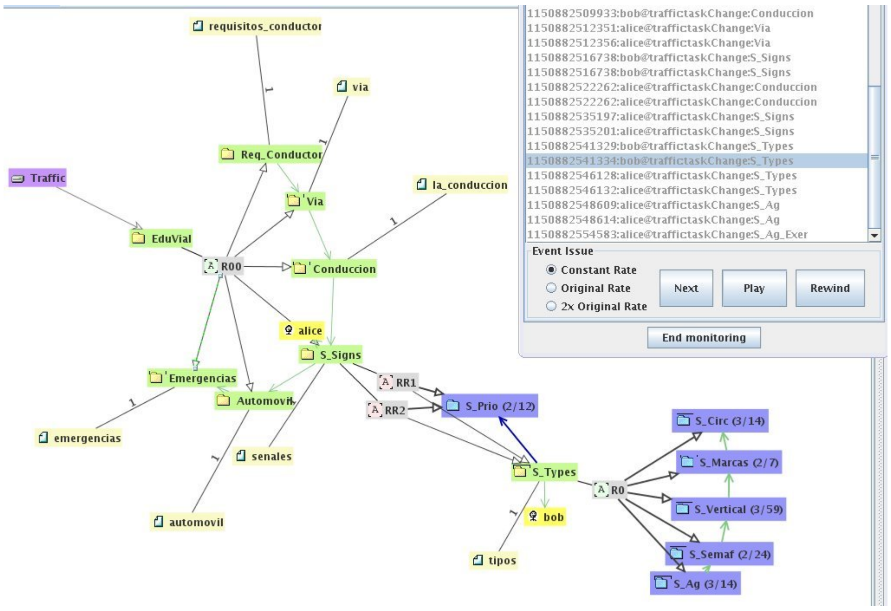
In the first experiment, the graph authoring tool was based on an older version of CLOVER, and did not contain a tree interface; indeed, the tree interface was built with the purpose of testing it against the graph-based authoring tool. In the second experiment, both interfaces were merged into WOTED (although users were only allowed to use one interface at a time), and the graph interface was upgraded to use the latest version of CLOVER. It is therefore interesting to compare the results of both experiments in light of the changes between the old and new versions of CLOVER.

Between both experiments, CLOVER underwent a major rewrite. From the user-interface point of view, the most salient changes are the addition of manual navigation mode, and multiple improvements designed to help in mental map preservation:



```
1150882470174:bob@trafficuserLogin
1150882478841:bob@traffictaskChange:EduVial
1150882478841:bob@traffictaskChange:EduVial
1150882482899:alice@trafficuserLogin
1150882485892:alice@traffictaskChange:EduVial
1150882485897:alice@traffictaskChange:EduVial
1150882504436:bob@traffictaskChange:Req_Conductor
1150882504441:bob@traffictaskChange:Req_Conductor
1150882506701:bob@traffictaskChange:Via
1150882506730:bob@traffictaskChange:Via
1150882509933:bob@traffictaskChange:Conduccion
1150882509933:bob@traffictaskChange:Conduccion
1150882512351:alice@traffictaskChange:Via
1150882512356:alice@traffictaskChange:Via
1150882516738:bob@traffictaskChange:S_Signs
1150882516738:bob@traffictaskChange:S_Signs
1150882522262:alice@traffictaskChange:Conduccion
```

(a) Users "Alice" and "Bob" at beginning of course.

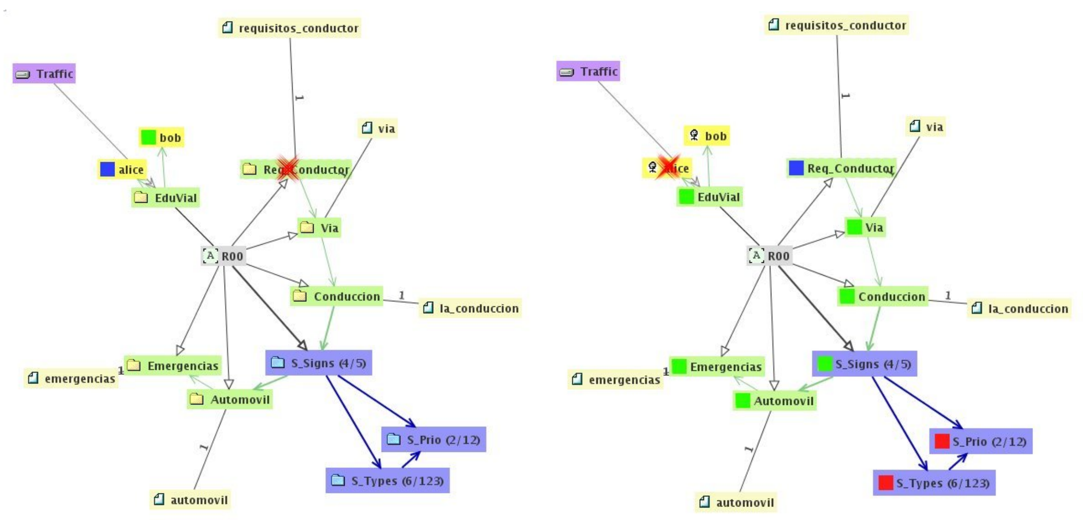


```
1150882509933:bob@traffictaskChange:Conduccion
1150882512351:alice@traffictaskChange:Via
1150882512356:alice@traffictaskChange:Via
1150882516738:bob@traffictaskChange:S_Signs
1150882516738:bob@traffictaskChange:S_Signs
1150882522262:alice@traffictaskChange:Conduccion
1150882522262:alice@traffictaskChange:Conduccion
1150882535197:alice@traffictaskChange:S_Signs
1150882535201:alice@traffictaskChange:S_Signs
1150882541329:bob@traffictaskChange:S_Types
1150882541334:bob@traffictaskChange:S_Types
1150882546128:alice@traffictaskChange:S_Types
1150882546132:alice@traffictaskChange:S_Types
1150882548609:alice@traffictaskChange:S_Ag
1150882548614:alice@traffictaskChange:S_Ag
1150882554583:alice@traffictaskChange:S_Ag_Exer
```

(b) Users after several monitoring events have been received and processed.

Figure 7.8: Course monitoring in WotEd. Users Bob and Alice are represented as vertices connected to their current tasks.

- Addition of manual navigation mode, to allow users fine-grained control on cluster navigation. Point-of-interest navigation can be confusing to new users.
- Predictability of point-of-interest changes during manual navigation was enhanced,



(a) Hovering over task; users are highlighted (b) Hovering over user; tasks are highlighted

Figure 7.9: Details on demand during monitoring. A color code is used to highlight task availability: blue represents "complete" tasks, red stands for "unavailable", and green is "available". Red crosses mark the hovered-over vertex. Note that both screen captures were performed with an old version of CLOVER's layout algorithms, so it may not be consistent with figures for chapter 6.

using the "aura" mechanism described in section 6.4.2.

- Incremental layout algorithms were changed, providing more compact layouts that do a better job of preserving parts of the layout during navigation changes. Graph layout cache and navigation history mechanisms were also introduced.

- Animation was totally reworked and made into a generic mechanism. Previously, animation was hard-wired into graph navigation, which proved difficult to test or improve. The new module allows non-navigational animations to be deployed.

The experimental setting and results for the second, most recent experiment (using the current versions of WOTED and CLOVER) will be presented first. A comparison with the old experiment will be performed at the end of the chapter.

7.3.1 Experimental setup

Since the tree and graph interfaces provide the exact same editing capabilities, the goal of the experiment is to measure author speed, accuracy and satisfaction with each of them. Before the experiment, participants received a brief primer to WOTAN course structure, an introduction to each of the tools, and a description of the tasks to be performed. They were also presented with a printed page with the symbols used in course representation and the list of requested tasks. The tasks were to be performed first with one editor and then again with the other one, on a single example course. The experiment was prepared so that half of the participants started with the tree-based tool, and the other half started with the graph-based one. As the same tasks are performed twice on the same course by users without previous experience with the system (or AH in general), we expected to find the timings of the second run to be lower than that those of the first run.

The 12 participants in the experiment were extracted from a relatively uniform population of computer science postgraduates, and had high familiarity with tree interfaces (ubiquitous in today's graphical user interfaces), and varying degrees of familiarity with graph interfaces, mostly used to read or generate documentation (as in UML). None had a background in information visualization or had worked with clustered graphs.

Tasks

All tasks were performed on the same example course depicted in figure 7.3. The tasks were designed to be simple but representative of a typical editing session, and were presented in order of increasing difficulty. For each task, the time required to complete it successfully (as determined by the experimenter upon request) was recorded. After all tasks for both tools had been completed, each user was requested to fill in a small "difficulty survey" assessing the perceived difficulty of each task/tool combination if he or she were asked to repeat it again on a different course. Difficulty for task/tool combinations was to be graded on a scale from 1 (trivial) to 10 (very complex). The task list was the following:

1. Find and change the name of a given fragment vertex, identified by a path that leads to it. Subjects have to navigate using the corresponding tool, which is the only problem presented by the task.

2. Locate all paths leading to a task vertex. One of the paths is provided in the statement. The task is representative of a typical problem in AH courses, and we expected users using the graph representation to perform better than those working with the tree representation.

3. Add a rule to a given task vertex, making the rule available only for users with a specific profile. Navigation to the task vertex is simple (it is located at the beginning of the course). The goal is to familiarize the user with the form-based editor used when editing rules; this familiarity should speed up Task 4.

4. Add a rule to a task vertex located deep within the course, and use this rule to provide a shortcut to another existing task vertex. This tests the creation of a variant in an AH course, and requires both navigation to the initial task vertex, and use of the rule editor.

7.3.2 Results

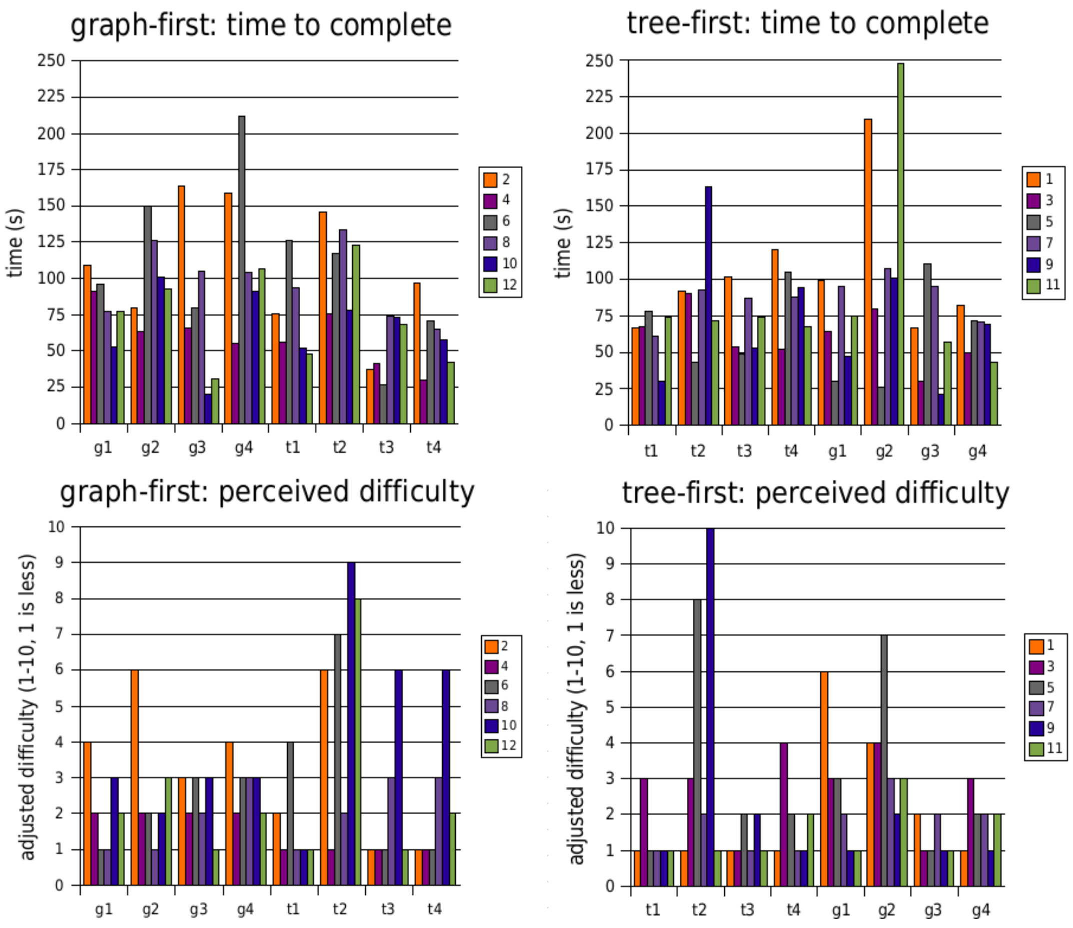


Figure 7.10: Results of the second experiment. Charts in the top row represent time in seconds; those in the lower row represent perceived difficulty (lower is easier). Column names reflect the interface and task involved; for instance, column **T3** contains results using the **T**ree interface to perform task number **3**.

Experiment results can be found in figure 7.11. Task difficulties have been adjusted so that the least difficult task for each user begins at "difficulty level" 1. The greatest differences when comparing average times between both groups of users can be found during the first run, where users that started out with the tree interface were consistently faster than those that started out with the graph. Comparing times for the leftmost columns of `t1` and `g1` shows that users required 20.8 seconds less to navigate to the required position using the tree as compared to using the graph. This is not a large difference; and its significance is questioned by a high variance and the fact that, when the same users tried out `g1` and `t1` later on, the difference changed signs; graph users required 7.0 seconds less to navigate to the same point than tree users.

Learning effects can be found by counting how many individual users were faster in their second run than in their first. No such effect is observed for tasks 1 and 2 – if interface order is ignored. However, when taking the interface into account, 4 of 6 users that started out with the tree were faster on their second run, while only 2 of 6 of those that started out with the graph were faster when performing task 1 with the tree. This suggests that it is easier to navigate a clustered graph as a tree than to apply clustered graph navigation to a tree.

Times for task number 2 were somewhat surprising, as users were faster at locating the number of alternative routes with the tree than with the graph. The difference is not significant (only 10 seconds faster, on average, during the first run, and slightly faster during the second one). However, for an experienced graph user it should be much faster to analyze connectivity on a graph than on a tree. A possible explanation is that users were indeed using the graph as a tree, expanding all possible vertices instead of searching for connections.

For tasks 3 and 4, learning effects are clearly observable: all 12 users managed to perform task 4 faster with the second interface than with the first. In task 3, the difference is smaller, with 4 out of the 12 users still performing better with the first interface. However, these 4 users are equally distributed among both interfaces.

Perceived difficulty

Times are roughly similar, and learning effects account for the greatest differences. However, substantial information is available from an analysis of perceived difficulty. For task 1, 7 out of 12 users considered the tree interface to be superior to the graph one when navigating trough a well-defined path. Only one user considered the tree harder to master. In task 2, preferences are reversed. 8 out of 12 (again, evenly distributed between both groups) preferred using a graph interface to discover topology. 3 users dissented. Opinions were divided almost evenly when assessing the difficulty of tasks 3 and 4.

7.3.3 Comparison with original experiment

In the original version of the experiment, only 3 tasks were required from the users, and a smaller sample size (only 8 participants) was available. The test setup was exactly the same: odd-numbered participants were asked to start out with the tree interface (then a separate application), while even-numbered ones started with the graph-based one.

All tasks were performed on the same example course as the new set of experiments. Tasks 1 and 2 are identical; Task 3 was different in the previous experiment: it required inserting a rule vertex, a task vertex, and a further fragment into the task vertex. Since, from the interface point of view, inserting one vertex is as complex as inserting any other, this task was simplified and subdivided in the new version of the experiment.

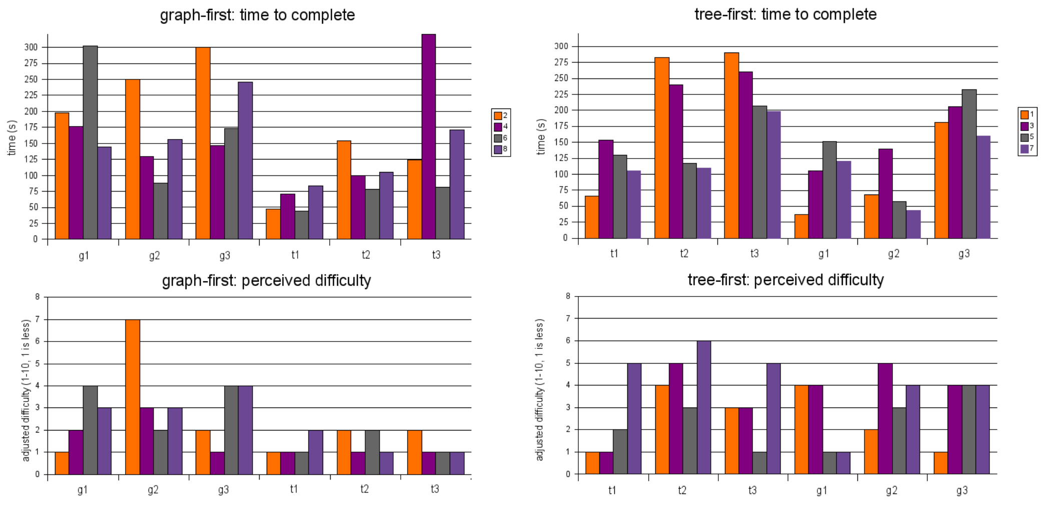


Figure 7.11: Results of the original experiment. Charts in the top row represent time in seconds; those in the lower row represent perceived difficulty (lower is easier). As in fig 7.10, column names reflect the interface and task involved.

Roughly, the old task 3 is equivalent to the new tasks 3 and 4. Results for the original experiment can be found in figure 7.11.

Lower times were expected for tasks 1 and 2 (which required heavy navigation), reflecting the improvements in graph navigation. Indeed, times for both tasks have descended, as shown in table 7.1.

Improvement	**Started with graph**	**Started with Tree**
task 1 with graph	29.3 s	136.6 s
task 2 with graph	85.3 s	43.4 s

Table 7.1: Graph navigation time improvement comparing the old graph-based authoring tool with WotEd.

No significant improvements can be attributed to the use of the tree or graph interfaces for other tasks; however, average times have also descended for the equivalent of task 3. Multiple interface glitches in the form-based interface used to edit rules were detected during the first experiment and addressed prior to the second one, accounting for this difference.

Chapter 8

Application to Other Small-World Domains

This chapter demonstrates the flexibility of the Clover framework, describing its used with a series applications developed for various domains:

- Targeteam is document preparation system. Its main strength is support for multiple delivery formats and a strong emphasis on document reuse in different contexts. A visualization tool based on an older version of Clover is described.

- Comet is a tool prototype intended to provide metadata annotation for ontology-based systems. Ontologies, and knowledge engineering in general, are a prime target for graph visualization.

- AC is a programming-assignment copy detection tool that uses graph representation to highlight cases of probable misconduct by students. Due to the collaborative nature of programming assignments, it doubles as a demonstration of Clover's applicability to friend-of-a-friend (FOAF) scenarios.

- Ulises is a real-time, graph-based monitor that represents the entities connected to an ubiquitous computing network. Entities include devices that are managed by the network and external objects 'known' by the ubiquitous environment, such as persons coming in and out of rooms.

Clover can be used to quickly develop clustered graph interfaces for many application domains. Development of a new interface requires, at the bare minimum, the implementation of a new base graph model capable of creating graphs for the target application domain. Later refinements would include customized vertex and edge representation, including labels and tooltips, customized actions, editing support, and a customized cluster hierarchy engine.

If so desired, Clover applications can present traditional, non-clustered graphs instead. Although this is not the intended purpose of Clover, the possibility certainly exists, as illustrated by AC's current use of non-clustered graphs.

8.1 Targeteam

Targeteam, developed at the Technische Universität München and described in [124], is defined as "a system for supporting the preparation, use, and reuse of teaching materials". The name is an acronym that stands for **TA**rgeted **R**euse and **GE**neration of **TEA**ching **M**aterials. The project is available online at [125].

Targeteam allows multiple courses, in different formats, to be generated from a single "content pool". Each course can be built using different content blocks, structures, levels of detail, and/or media types. A hypermedia output mode allows the creation of web-based courses. These courses, however, are static; once generated, no further adaptation is performed. Therefore, Targeteam can not be considered an adaptive hypermedia system. However, it would be possible to build adaptive hypermedia courses from Targeteam hyperlinked output; this approach has been briefly explored in [56].

Content reuse in Targeteam

Support for content reuse can be subdivided into metadata support, content abstractness, and support for adaptability. Targeteam supports only metadata that is directly related to the annotated content, such as `language`, `author` and `format`, or a predefined set of semantic relations between chunks of content, such as `motivation`, `illustration`, `exercise`, or `summary`. Metadata defined at one content level is inherited by lower levels unless overwritten.

Content in Targeteam, with the exception of non-text media objects, does not make any reference to presentation. The structure of content blocks is also abstract, in the sense that contents do not specify "chapters" or "paragraphs". The final structure of a course is generated from that of its components upon creation. This level of abstractness allows Targeteam to generate output in multiple formats, such as $\LaTeX$, PDF or HTML.

Content can be adapted and modified for inclusion into a course, by omitting, adding or replacing parts. The author can provide a series of transformations to be applied to the original content prior to inclusion. These transformations can be made at arbitrary levels of granularity. A copy is made of the contents to be transformed prior to transformation; course preparation only transforms an internal copy, leaving original contents intact.

Pools, modules and items

The basic unit of content is the *module*, a document written in the `TeachML` XML dialect. Modules can contain many different types of content, ranging from simple text to complex markup, tables, or mathematical formulas. All content is organized into *issues*. Issues represent a small topic, and are organized into a thematic hierarchy revolving around the *kernel* – the module's central issue. Modules can reference other modules for different purposes, the most common of which is direct inclusion (either of the whole module, or only of particular sections). Cross-referencing is also supported. Non-textual media items such as images constitute *atoms*, which can also be referenced for inclusion within modules.

A *pool* is a set of modules, organized into a directed acyclic graph structure. There can be multiple connected components within a Targeteam pool, since courses are not required to share any contents. A part of the TeachML language deals with the integration and adaptation of a module's descendants into itself, by defining suitable transformations in the XSLT transformation language. When these transformations are processed, a module effectively constitutes a comprehensive "macro-module" composed of its own contents mixed with those of its (transformed) descendants.

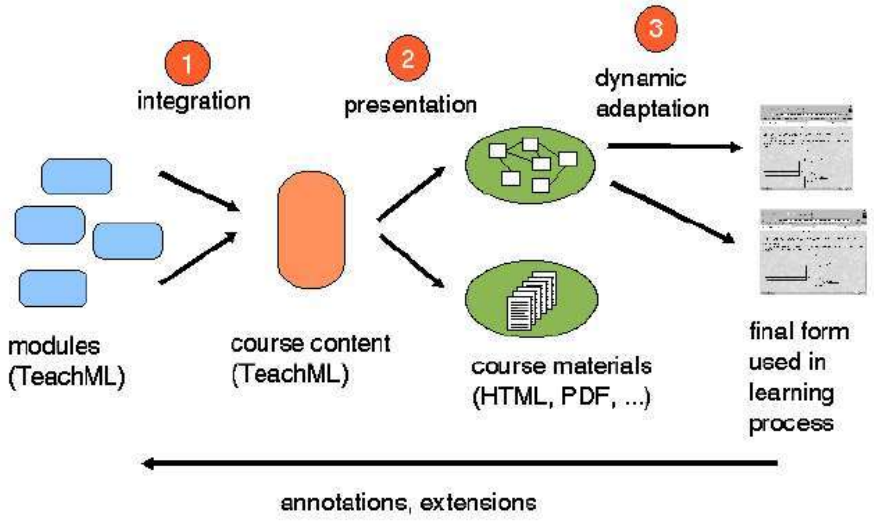


Figure 8.1: Overview of the Targeteam course creation process, from [125]. The course is first integrated into a single TeachML file (1), presentations in different formats can then be generated (2), and further processing can be performed (3) prior to delivery.

A course can be generated by specifying a root module. During generation, a single TeachML file is created with a snapshot of the contents of the root module and all its sub-modules. The creation of this intermediate file ensures that all presentations created from a given course correspond to the same version of the pool contents. This process is graphically depicted in fig. 8.1.

Graph creation

Targeteam pools can be stored inside databases, files, or distributed across the network and accessed through "pool servers". TARGETEAM's API isolates a programmer from the details of each location; this approach has been chosen for the implementation the initial conversion from pool to graph.

Graph creation begins with the URL used to access the pool. The pool's modules and atoms are then downloaded, and individual modules are parsed to locate statements that modify their submodules. Module inclusion can be also be determined directly from the API, yielding a tree-like structure.

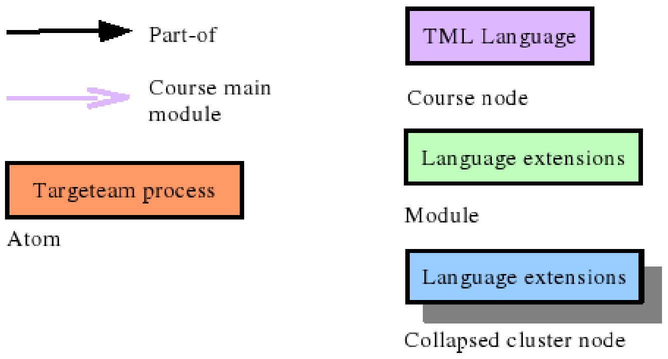


Figure 8.2: Symbols used in TARGETEAM course representation, from [56].

Figure 8.2 contains a description of the symbols used in TARGETEAM pool graphs. Course "main module" arrows are used to link course vertices with modules, while "part-of" edges are used for all types of inclusions. Although it would be possible to create many other arrow types, depending on the exact type of inclusion taking place, this would require a certain amount of heuristics to detect the intended type. Inclusion type may not be evident actual XSLT inclusion statement.

Interaction and comments

Interaction TARGETEAMpool graphs is very simplified in relation to that of WOTED's course graphs, as authoring is not required, and the only use of the interface is to allow a user to browse pool structure. For instance, a very simple (regular expression) filter is used, and clustering is performed by CLOVER's default rule clustering engine.

Figure 8.3 illustrates the interface used when representing TARGETEAM pools. Note that a previous version of CLOVER was used, which explains multiple changes in the basic interface layout, toolbar icons, and so on. Cluster labels include a textual name and the number of subsumed clusters, instead of the fraction notation used in WOTED. Tooltips for all vertex types are also available. Basic statistics on the contents of the vertex are displayed, as obtained by querying the TARGETEAM API.

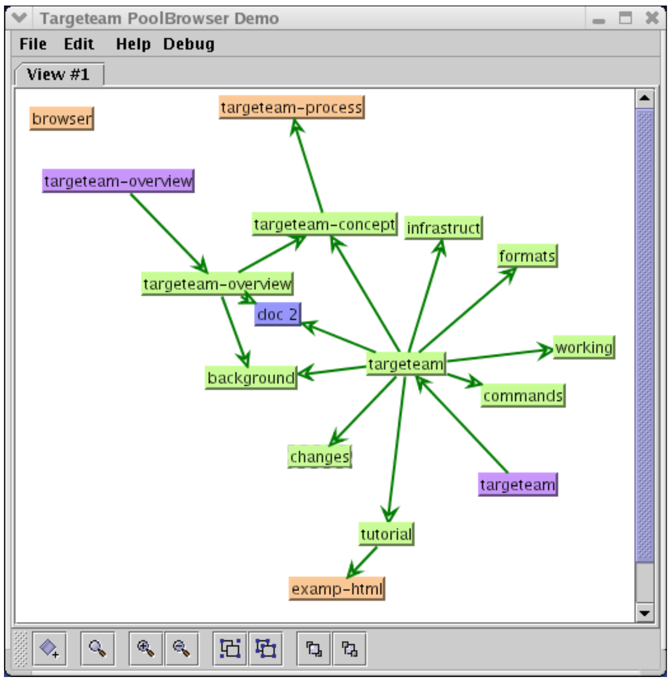


Figure 8.3: TARGETEAM pool browser interface. Green boxes represent textual contents, blue boxes are clusters, violet boxes stand for courses, and orange boxes represent "atoms" (such as images or Java applets). Manual adjustment of graph layout has was performed.

Although editing is disabled, double-clicking on any vertex triggers a (read-only) dialog with the vertex contents to be displayed. In the case of modules, the contents of this dialog will be a snippet of TEACHML text. XML is pretty-printed to improve legibility when displayed.

The use of CLOVER to explore large TARGETEAM repositories could readily be extended to other link structures. Many document repositories could benefit from similar facilities; large websites and code repositories come to mind. A key feature of such systems would be correct categorization of information; the default clustering engine used for TARGETEAM repositories is clearly insufficient for these goals.

As commented in section 4.2, document reuse can greatly benefit from standarized metadata anotation with a structured vocabulary geared towards classification, such as an ontology. If sufficient classification terms were defined for each reusable document in the repository, development of suitable automatic clustering engines would be greatly simplified.

8.2 Comet

COMET is an acronym that stands for **C**luster-**O**riented **M**etadata **A**nnotation **T**ool. Metadata annotation indicating a list of related concepts is an important aspect of many document reuse scenarios (see section 4.2). Ontologies are natural choices for metadata annotation. However, a user must be familiarized with the ontology in order to choose the most descriptive concepts. The COMET tool aims to visualize ontologies, allowing users to browse, explore, and select these "most descriptive concepts" for each document to be annotated.

At present, COMET is still in its initial stages of development. Ontology browsing is possible, but metadata annotation itself has not yet been introduced. Therefore, this section only presents the ontology visualization approach used in CLOVER. Ontology visualization itself can be found in applications such as JAMBALAYA [121]; however, no current ontology visualization tool supports clustered graph visualization.

The screenshot shown in fig. 8.4 of COMET's interface displays the ontology used in the ICCARS project [84], developed at the École Nationale Supérieure des Télécommunications de Bretagne. This ontology was used to annotate all documents within a large document repository. The adaptive hypermedia system used within the ICCARS project then uses queries to locate the documents of the repository that best match each user's current user model.

Graph creation

The goal of COMET is to represent many types of ontologies to enable metadata annotation; therefore, and intermediate ontology representation format is used to isolate the application from the details of each particular ontology representation. In the future, COMET should read both OWL and F-LOGIC ontology formats, which would be converted to the internal intermediate representation prior to graph visualization. Currently, only F-LOGIC is supported.

The intermediate representation format distinguishes the following elements:

- "Classifier" class – a class which is only used as a type for attributes.
- Normal class – a normal class.
- Instance – an instance of a class
- Relations – a relation between exactly two classes.

Hierarchical clustering is performed by following only "is-a" relationships. This strategy works fine as long as multiple inheritance is not used; in the presence of multiple inheritance, such as can be found in parts of the ICCARS ontology, some clusters result larger than would be expected. However, recall that users can manually modify cluster hierarchies once generated.

8.3 AC

AC is an anti-plagiarism system for programming assignments. It helps to detect software plagiarism within programming assignments written in C, C++ or JAVA. The

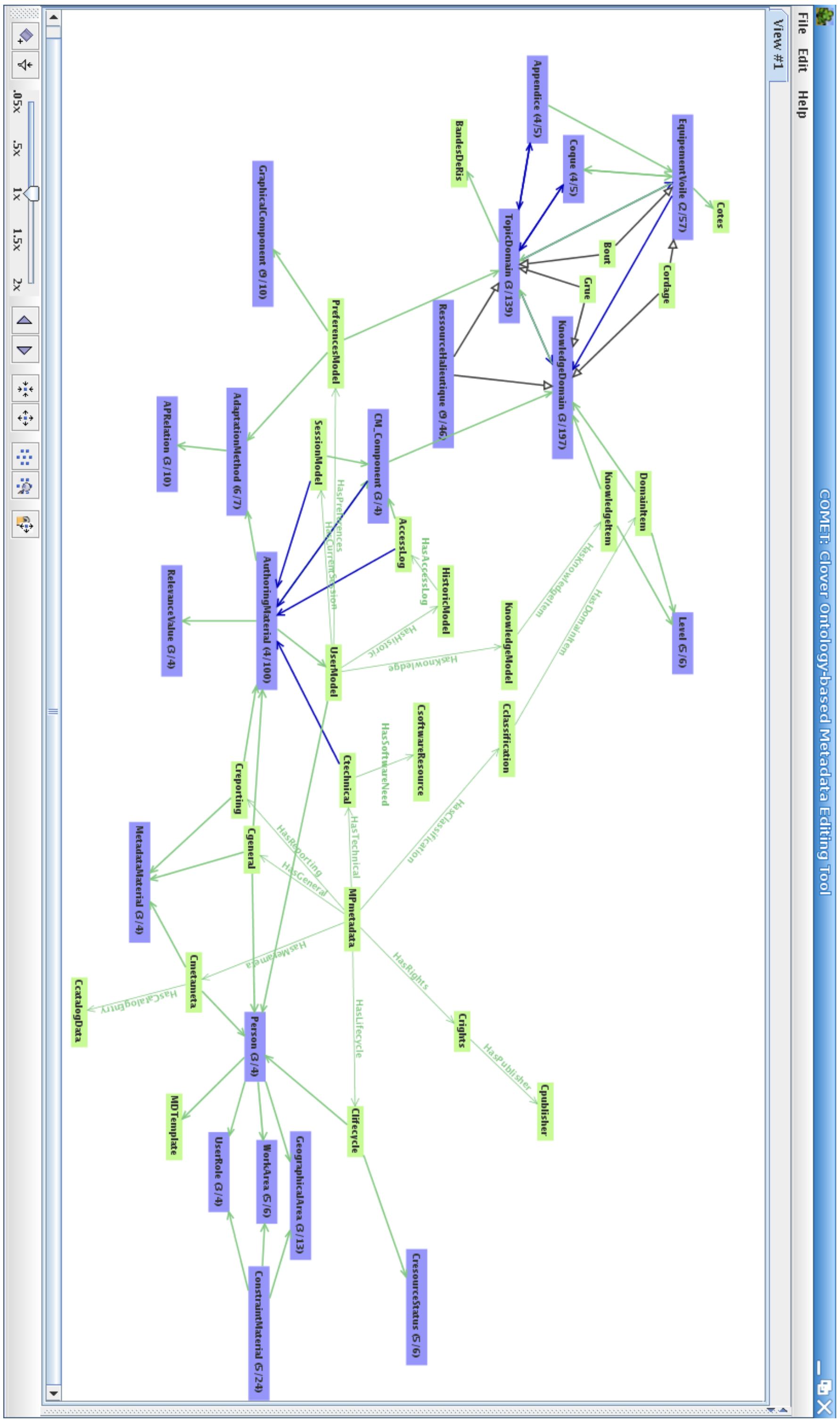


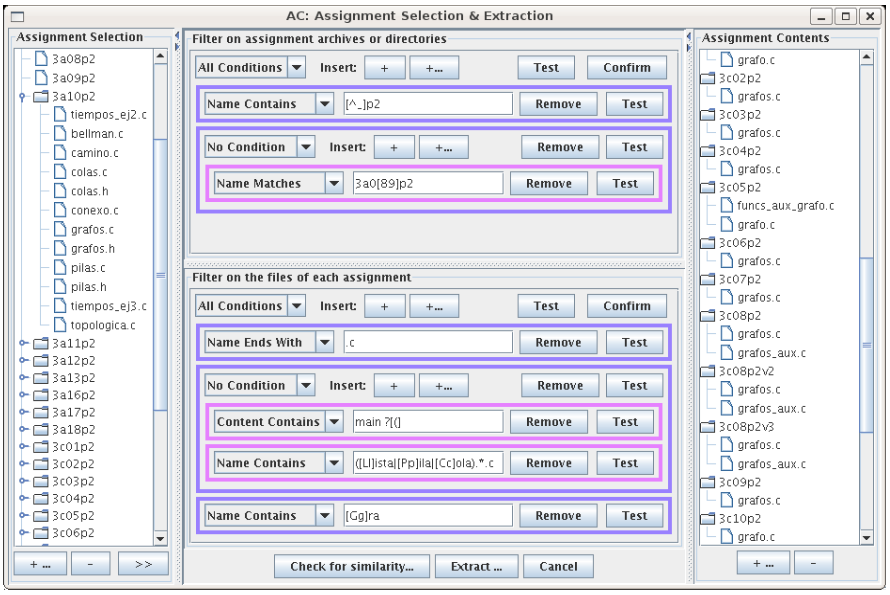
Figure 8.4: Comet interface and ontology used in Iccars.

system includes multiple algorithms to detect software similarity, including many found in the scientific literature on plagiarism detection and entirely original ones. A full description can be found in [58].

Key features of AC include an extraction/filtering utility designed to ease the initial task of preparing assignment submissions for analysis, simplifying an otherwise repetitive and error-prone task, support for pluggable source-code comparison algorithms, and visualization of comparison results using multiple views, including a graph view based on Clover.

Assignment extraction and similarity testing

Since the goal of plagiarism detection in an academic context is to assist a human grader, it is important to make this assistance as extensive as possible. The first step when performing any type of automated plagiarism detection is to prepare a set of submissions for analysis. This task of converting assignment submissions to the internal, standardized input format expected by the plagiarism detection tool can be referred to as *assignment filtering*. If assignment filtering requires too much of the grader's time just to "feed the tool", the grader may decide that using the tool is simply not worth the effort. This section describes the assignment extraction utility included in AC to ease this task, which can be divided into two main phases: assignment selection, and assignment standardization.


AC: Assignment Selection & Extraction
Assignment Selection
Filter on assignment archives or directories
Filter on the files of each assignment
Assignment Contents
Insert:
Test
Confirm
Remove
Check for similarity...
Extract ...
Cancel

Filter on assignment archives or directories

All Conditions
Name Contains [^_]p2
No Condition
Name Matches 3a0[89]p2

Filter on the files of each assignment

All Conditions
Name Ends With .c
No Condition
Content Contains main ?[(]
Name Contains ([Ll]ista|[Pp]ila|[Cc]ola).*.c
Name Contains [Gg]ra

Assignment Selection: 3a08p2, 3a09p2, 3a10p2 (tiempos_ej2.c, bellman.c, camino.c, colas.c, colas.h, conexo.c, grafos.c, grafos.h, pilas.c, pilas.h, tiempos_ej3.c, topologica.c), 3a11p2, 3a12p2, 3a13p2, 3a16p2, 3a17p2, 3a18p2, 3c01p2, 3c02p2, 3c03p2, 3c04p2, 3c05p2, 3c06p2

Assignment Contents: grafo.c; 3c02p2 (grafos.c); 3c03p2 (grafos.c); 3c04p2 (grafos.c); 3c05p2 (funcs_aux_grafo.c, grafo.c); 3c06p2 (grafos.c); 3c07p2 (grafos.c); 3c08p2 (grafos.c, grafos_aux.c); 3c08p2v2 (grafos.c, grafos_aux.c); 3c08p2v3 (grafos.c, grafos_aux.c); 3c09p2 (grafos.c); 3c10p2 (grafo.c)

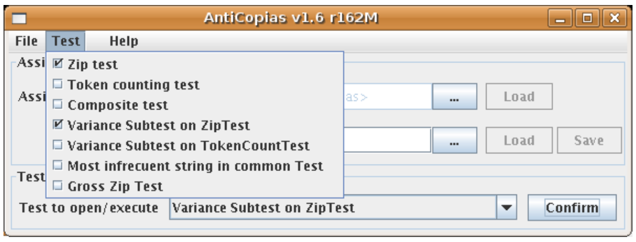
Figure 8.5: AC extraction interface. Support for similar complex filtering rules will be added to future versions WotEd.

The design of AC relies on a general framework for similarity measurement. This framework is based on requiring all similarity distance algorithms (also referred to as *similarity tests*) to return, for each pair of assignments A and B, a single value to represent the similarity distance between both, according to that algorithm. Values are normalized between 0 (identical sources), and 1 (minimal similarity, no common information). Therefore, this distance is also referred to as *normalized similarity*.

This standardization makes similarity distance algorithms interchangeable from the point of view of further processing steps, and facilitates comparisons between them: data sets can be examined using different similarity distance algorithms (or *test*), simplifying the evaluation of the relative strengths and weaknesses of each. It is easy to add new tests, and to compare them against the existing ones. Additionally, results from multiple tests can be easily composed into higher-level tests. The dialog used for test selection is displayed in fig. 8.6 (a).

The output of a test, or a composition of tests, is a *distance matrix* of N rows and columns, containing a single value between 0 (exactly equal) and 1 (totally different) for each pair of assignments. Extracting only the lowest distances of this matrix will hide broader trends in their overall distribution, and the identification of possible plagiarism patterns between assignments from this sea of numbers would require patient analysis.

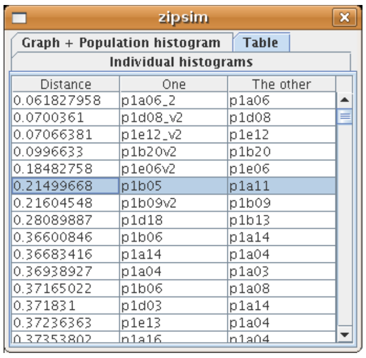
AC allows the use of the raw similarity values (see fig. 8.6 (b)), but also provides integrated statistical analysis and a set of visualizations to ease their interpretation (fig. 8.7). These visual aids are also critical in allowing users to assess the meaningfulness of the outputs obtained from each test: plagiarism detection shares many characteristics with outlier detection, and outliers are only such within the context of a broader distribution.



☑ Zip test
☐ Token counting test
☐ Composite test
☑ Variance Subtest on ZipTest
☐ Variance Subtest on TokenCountTest
☐ Most infrecuent string in common Test
☐ Gross Zip Test

Test to open/execute: Variance Subtest on ZipTest

(a) Test selection in AC



Distance	One	The other
0.061827958	p1a06_2	p1a06
0.0700361	p1d08_v2	p1d08
0.07066381	p1e12_v2	p1e12
0.0996633	p1b20v2	p1b20
0.18482758	p1e06v2	p1e06
0.21499668	p1b05	p1a11
0.21604548	p1b09v2	p1b09
0.28089887	p1d18	p1b13
0.36600846	p1b06	p1a14
0.36683416	p1a14	p1a04
0.36938927	p1a04	p1a03
0.37165022	p1b06	p1a08
0.371831	p1d03	p1a14
0.37236363	p1e13	p1a04
0.37353802	p1a16	p1a04

(b) Table view of test results

Figure 8.6: Test selection and raw results in AC.

Graph generation

The graph view is used to detect cliques of highly-related assignments. In the graph view, represented in fig. 8.7 (a), vertices represent assignments and edges are used to convey distances between assignments. As all vertices have a distance to all other vertices, the graph is a complete graph. However, a huge majority of distances correspond to pairs of assignments that bear very little similarity; representing these edges would result in a great quantity of noise. Therefore, only a chosen subset of edges is represented, using a lower threshold (recall that low distance values indicate high assignment similarity).

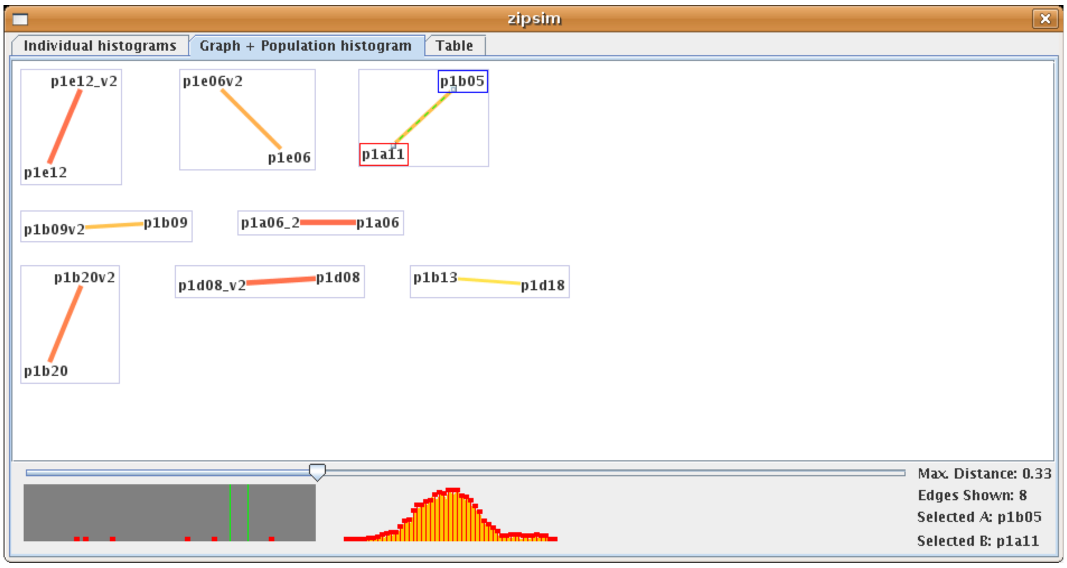
The threshold beyond which graphs edges are *excluded* can be manually set with a slider (again, see 8.7 (*a*)), bound between 0 (only edges which correspond to "exact copies" will be shown) to 1 (include all edges, regardless of similarity level). To aid the teacher in selecting a good threshold, the slider is placed above a histogram of the frequencies of each distance in the test's distance matrix, and a value corresponding to a statistically robust outlier identifier ([58]) is chosen as a starting point. The shaded part of the histogram of fig. 8.7 (*b*) represents the portion of edges that are currently being used in the graph.

Not all edges below the threshold are included, because this would still result in an unnecessarily cluttered graph, specially for high cuttof values. Edges that do not belong to each connected component's *minimum spanning tree* (MST) can generally be removed without much loss of information. This approach, suggested by Whale [139], results in a drastic simplification of each connected component, and speeds layout considerably. However, removing all edges not in the MST risks the loss of edges that have very high relevance (but result unnecessary in the construction of the MST), while other edges with lower relevance (but a role within the MST) are included. This can lead to a false impression of the component's structure. To characterize these edges that are important but do not belong to the MST, the following approach has been adopted: if a component has $|V|$ vertices, the lowest-distance $|V|$ edges are also retained, regardless of whether they belong to the MST or not. This additional heuristic ensures that structures such as triangles are not lost, with a minimal increase in graph complexity.

Histogram visualization

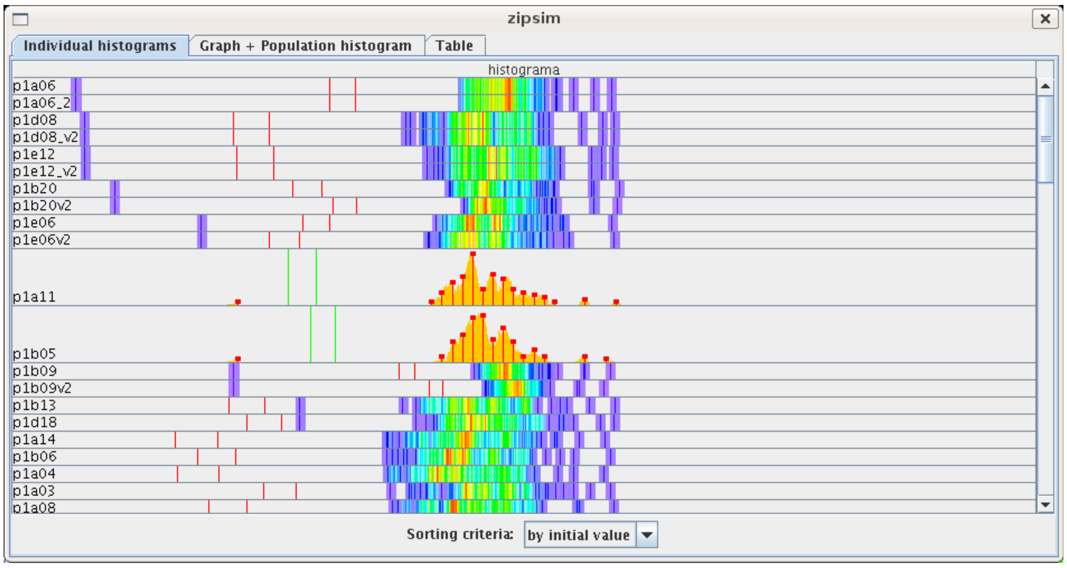
A third type of visualization is represented in fig. 8.7 (*b*). This view presents rows of "individual histograms", generated for each of the assignments. As in the general histogram found in the graph visualization, each of this histograms represents the frequency of each distance; unlike the general histogram, only distances between a single assignment (the row header) and all other assignments are considered.

Histograms in the histogram visualization are condensed, and require a small amount of training to get accustomed to: color coding, instead of bar height, is used to convey frequency. These condensed, color-coded histograms have been termed *hue histograms*; they are reminiscent of the colored bars found in chemical composition analysis [88]. In AC's hue histograms, blue is used to represent low frequency, red for high frequency,



Max. Distance: 0.33
Edges Shown: 8
Selected A: p1b05
Selected B: p1a11

(a) Graph view of test results



Sorting criteria: by initial value

(b) Histogram visualization in AC

Figure 8.7: Visualization of results in AC. CLOVER is used to implement graph-based result visualization.

and intermediate hues (in traditional visible spectrum order) are used for the remaining values. Selecting any row of the hue histogram brings up the familiar bar representation used in the graph visualization. Hue histograms are intended to save space, providing informative overviews; bar histograms and tooltips over each value provide details on demand.

Interaction and comments

Interacting with all three views (raw distances, graph and histogram) follows a similar pattern; double-clicking a single vertex (or histogram row) displays the source code for that vertex, and double-clicking an edge (or a distance value, in either table or histogram) results brings up a side-by-side comparison of both of the assignments it is connected to. Graders are expected to manually analyze suspects flagged by the system to determine whether actual plagiarism is present.

AC's use of Clover-based visualizations is interesting in several accounts. It illustrates Clover's use in a broader application where the main interface is not necesarily a graph, and demonstrates the use of Clover for simple automated graph visualization, without using any of its clustering features.

Later versions of AC will probably include graph clustering, as a tool to analyze the distribution of "assignments families" located with similarity tests. For instance, such an analysis could distinguish between assigments programmed using response X, Y or Z to problem P, where each answer could be characterized defining a filter and comparing filtered assignments to archetypical implementations. Analysis of these trends could provide insights beyond plagiarism detection, and constitutes an open research line.

8.4 Ulises

The Odisea system [76] is an ubiquitous computing environment developed at the Universidad Autónoma de Madrid. A centralized blackboard is used to maintain the system's *context*, which environment-aware applications can access to query sensors and affect changes on devices. Ulises is a proof-of-concept, Clover-based application that monitors and displays the contents of the blackboard.

Odisea's blackboard architecture stores information in a relationship graph, and queries are modelled as graph traversal operations. A subcription mechanism is supported, allowing clients to request to be notified when certain conditions are triggered (for instance, entities are added or removed). Figure 8.8 illustrates the structure of the relationship graph, which is similar to that of an ontology. Recent work in the system is oriented towards formalizing the schema used to classify entities (see [75]), strengthening this similarity.

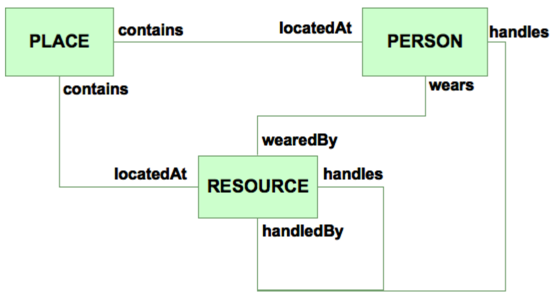


Figure 8.8: Simplified entity-relationship graph for the Odisea system. Multiple resource subtypes (not shown above) have been defined.

Graph generation and Interaction

Since the blackboard structure is already a graph, blackboard vertices and edges have been used for the graph representation. Access to the blackboard's structure is performed through Odisea's external API, which supports queries for entities and their relationships. Given an initial root vertex, the graph is explored in depth-first order following "containment" edges, until all devices and relationships have been located.

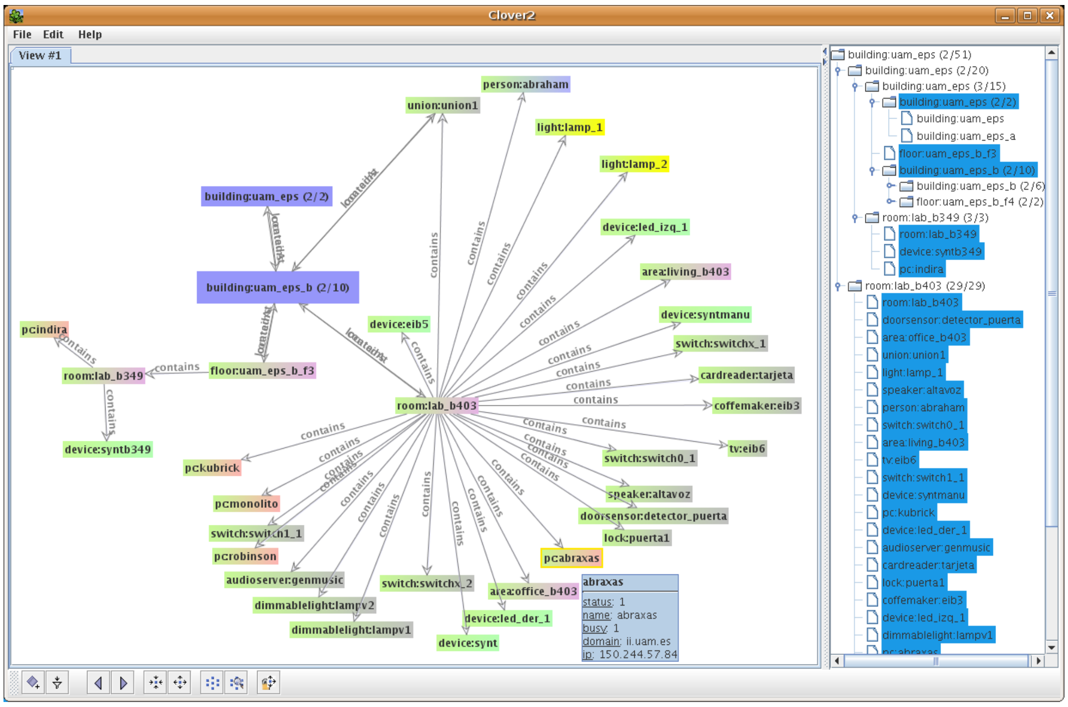
Once Ulises has generated a copy of the graph, it proceeds to monitor graph changes. Whenever it is notified of a change, the copy is updated, and the change is graphically animated in the view. This view is represented in fig. 8.9; it is embedded in the default Clover interface.

The cluster hierarchy is generated with a slightly modified rule clustering engine; only containment edges are considered. Since these edges (almost) form a tree, the result is a set of clusters that matches the physical containment relations present in the network. Notice that certain clusters are too large (room `lab_b403`, in the center of fig. 8.9). A better approach would be to subdivide these "large clusters", for instance taking device types into account.

Hovering the mouse pointer over any vertex queries Odisea for a list of vertex properties. The results are displayed as tooltips. Other than vertex tooltips, no additional interaction mechanisms have been added. However, dialogs from Odisea's own device-control interface could be easily merged in the future, allowing an "edit" action to be enabled.

Observations

The Ulises application is only a prototype, and many enhancements could be performed to increase its usability. In its current state, it demonstrates the use of Clover to visualize graphs that are dynamically queried from an online system, and later monitor graph changes over time. From this point of view, monitoring a device network very similar to monitoring a student's actions in WotEd; both applications require support



abraxas
status: 1
name: abraxas
busy: 1
domain: ii.uam.es
ip: 150.244.57.84

building:uam_eps (2/51)
building:uam_eps (2/20)
building:uam_eps (3/15)
building:uam_eps (2/2)
building:uam_eps
building:uam_eps_a
floor:uam_eps_b_f3
building:uam_eps_b (2/10)
building:uam_eps_b (2/6)
floor:uam_eps_b_f4 (2/2)
room:lab_b349 (3/3)
room:lab_b349
device:syntb349
pc:indira
room:lab_b403 (29/29)
room:lab_b403
doorsensor:detector_puerta
area:office_b403
union:union1
light:lamp_1
speaker:altavoz
person:abraham
switch:switch0_1
area:living_b403
tv:eib6
switch:switch1_1
device:syntmanu
pc:kubrick
device:led_der_1
audioserver:genmusic
cardreader:tarjeta
lock:puerta1
coffemaker:eib3
device:led_izq_1
dimmablelight:lampv1
pc:abraxas

Figure 8.9: ULISES device network representation. A vertex tooltip for a computer connected to the network is displayed in the lower-right corner of the graph.

for incrementally updating a graph representation when structural changes are notified. This is one of CLOVER's strong points.

Part III

Conclusions and Future Work

Chapter 9

Conclusions and Future Work

9.1 Conclusions

This work examines the use of hierarchically clustered graphs to interactively visualize large graphs. Although the primary goal is to use the resulting visualization in the field of adaptive hypermedia authoring, arguments are provided to support their use in any other domain where the small-world property holds true.

The proposed approach covers the full visualization process, from graph creation to its representation in a graphical user interface. Within this process, special emphasis is made on the features that make this approach different to conventional graph visualization: automatic generation and incremental update of the hierarchical clustering, the use of semantical fisheye-assisted navigation within the clustered graph, and the treatment of mental map preservation issues that result from navigational actions.

Clover, a framework that implements the proposed approach, is presented. The design of Clover is highly modular, and can be extended to support visualization for a wide range of applications. The framework itself is released as open source, and applications built on top of it can benefit of any improvements that may be contributed. The Clover framework is available for download at

http://tangow.ii.uam.es/clover

WotEd is a Clover-based authoring tool for the Wotan adaptive hypermedia course system. Since Clover performs layout and clustering automatically, WotEd can visualize externally created courses without any user intervention. However, users are free to alter both; WotEd will retain any user modifications between authoring sessions. Additionally, changes to any view of a course are propagated to all other open views, and layouts are reused whenever possible in an attempt to preserve the user's mental map. Comparison of the WotEd clustered graph interface to traditional trees shows that in simple tasks, clustered graphs are only slightly more difficult to use than trees, although as expected, the graph interface presents important advantages in connectivity-related tasks, and conveys the true course structure better.

WotEd allows authors to monitor the actions of one or more students through an adaptive hypermedia course in real time. Monitoring demonstrates the use of interactive graph visualization on a dynamically changing. The use of a single conceptualization for authoring and monitoring/tutoring is expected to have positive effects on the overall ease of use of the Wotan course platform. Since a large number of existing AH systems use domain overlays to represent a large part of their user models, the approach followed in WotEd for user model visualization can easily be extended to other systems.

This work also presents several other applications that demonstrate the use of Clover to visualize domains that are totally unrelated with adaptive hypermedia. This underlines the generality of the approach. Each is also a novel tool in its own right; dynamically changing device networks, aggregated in real-time; complex document reuse diagrams, source code similarity networks and clustered ontology visualization would merit separate research in and of their own.

Some parts of Clover build upon well-understood areas, such as traditional graph representation, graph drawing, or general interface design guidelines; others are more innovative and can be considered true contributions. Throughought the following section, key parts of the Clover framework are viewed from a distance, examining their novelty and questioning how they differ from other approaches. The final section is dedicated to future work.

9.2 Discussion

This work began as an effort to find an interface that retained the ease of use of trees, while gaining the expressive capability of graphs. The approach was to "abstract away" parts of the initial graph in a controllable, tree-like manner. The resulting interface was to be used in an Adaptive Hypermedia course authoring tool; lack of good authoring and maintenance tools is partly responsible for the limited use of AH systems, despite their potential advantages regarding content relevance and reuse. Within an AH authoring tool, the visual interface is responsible for helping authors to manage the complexity of multiple, alternative navigation paths. Similar interface problems can be found in other fields, which could also benefit from better graph visualization.

Tree-based interfaces are limited to representing structures where each element is reachable only through one path. Within many traditional tree interfaces, branches can be easily expanded or collapsed, and the displacement to surrounding nodes is localized and simple to understand. On the ther hand, a graph interface can represent much more complex structures; in particular, there can be any number of paths leading to any given vertex. However, the concept of "branches" cannot be directly applied to general graphs; the nearest equivalent are clusters. Once suitable clusters have been located, cluster expansion and collapse requires careful incremental layout to preserve locality, and multiple other mental map preservation techniques are needed to make navigational changes predictable and easy to follow.

Graph clustering and cluster hierarchies

Clusters are widely used in graph visualization. They are generally represented as large, box-like vertices that hold subgraphs inside them. However, the substitution of a cluster's contents for a single (much smaller) cluster vertex is less frequent, because unless the graph layout is recalculated, a vacant areas proportional to the size of the old cluster will be left in its place. Recalculating the layout introduces mental map preservation issues, while leaving a vacant area is sub-optimal, in the sense that no space saving is achieved in exchange for a much less informative representation of the cluster's contents.

Besides the substitution of cluster contents for single vertices, Clover is different from mainstream graph visualization interfaces in the use of deep cluster hierarchies. Typical approaches that represent whole subgraphs within "cluster boxes" are inherently limited to shallow hierarchies, because layout of large cluster boxes without overlaps unnecessarily distorts the overall graph layout, and large clusters take too much space to represent in fully expanded form. A deep cluster hierarchy implies relatively large top-level clusters, which are unwieldy without true cluster collapse. This requires automatic, incremental layout to economize graph layout area. The use of deep cluster hierarchies (instead of simple one-level hierarchies) also allows abstracts of large graphs to be generated at arbitrary levels of detail. Abstracting away unnecessary details has an added advantage from the point of view of layout complexity: a lower number of visible vertices and edges can result in a great increase to layout speed.

The use of clustered graphs requires meaningful clusters to exist, and ideally the collapse of a given "layer" of clusters should yield a graph that can also be clustered. The analysis of "natural" graphs provided at the start of the work shows that this is indeed the case for many domains. A graph where the small-world property holds will have a high clustering coefficient (and therefore well-defined clusters probably exist, and can be located) and a low average path length. A low average path length is interesting because abstracting away clusters results in a much more compact graph; however, it is not truly necessary, since chains of related vertices can also be collapsed into clusters. To ensure that the process can be repeated to generate a complete and meaningful cluster hierarchy, the graph must also exhibit a degree of self-similarity. Again, this property has been shown to be common in many domains.

The widespread existence of these properties supports the feasibility of using hierarchically clustered graph visualization for large graphs, with hundreds, thousands or even millions of vertices. Clover has not been tested with more than hundreds of vertices, and would probably require more efficient algorithms to reduce interface update delays to the levels expected of an interactive application. Extending Clover in the direction of truly large datasets is not a current goal. However, the representation of hundreds of vertices in abstracted form is in itself very worthwhile. Even below the hundred-vertex mark, many graphs currently prove unwieldy and difficult to understand.

Notice that the small-world property and related characteristics rely only on the graph's vertex-edge structure, not on its actual semantics (which is currently an intractable problem). In a good graph representation, the graph's structure will closely follow its semantics; and in an ideal scenario, the clustering algorithm will be able to locate optimal vertex groupings based on this structure. This is by no means guaranteed, and depends heavily on the clustering algorithm used. Many clustering algorithms have been described to locate clusters based on graph connectivity and (for certain domains), domain-specific information. Others place the burden of locating clusters directly on the user. In Clover, the use of a fully automated clustering mechanism is an important feature of the approach, even though the current rule-based clustering engine is rather simplistic. More general connectivity-based clustering engines are expected to be incorporated in the near future.

In many graph visualization systems that use clustering, visualization of domain information requires at least minimal user intervention to create clusters, label them, or even perform layout. Even if initial defaults can be found for all these operations, changes to the base graph may not result in incremental update of existing graph views. Clover makes a point of allowing users to represent a graph as a clustered graph without any type of user intervention. The cluster hierarchy is automatically generated, as are cluster vertex labels, initial level of detail, and the starting layout for graph vertices. This allows Clover-derived applications to immediately visualize graph data which was generated externally from the application. Furthermore, Clover supports automatic propagation and animation of any update to the base graph (or indeed any other pipeline stage).

Fully automatic generation of hierarchy and graph layout may result in suboptimal results; specially so since both of these properties are somewhat subjective, and are difficult to quantify. It is very frequent for systems that perform automatic layout (and clustering) to allow users to manually adjust both clustering and layout, but it is rare to find systems where manual adjustments are preserved after further automatic operations. Clover allows both types of adjustments, and preserves user-modified layouts to a certain point, but currently the default clustering engine loses track of user adjustments to clustering. This is considered a serious issue, and will be addressed in future versions.

From the point of view of a user developing an application on top of Clover, providing a graph model is enough to obtain a basic clustered graph visualization. In creativity support terms (see section 3.2.3), the shallow end is shallow indeed. Correspondingly, there is also a deep end: all parts of the framework's workings can be adapted, and even basic tinkering can result in worthwhile improvements. The applications described in chapter 8 validate this statement.

Adaptive hypermedia authoring

The initial goal was to apply clustered graph visualization to the domain of Adaptive Hypermedia. Within AH, graph representations make sense for navigational structures, domain models, and the parts of user models that represent domain model overlays. Log files can also be seen as graphs, and this approach has already been followed for "normal" WWW log analysis. From the point of view of content organization and reuse, graph structures can also be used to represent metadata taxonomies or ontologies.

WotEd can represent adaptive hypermedia course structures found in Wotan courses. Wotan courses currently follow the Tangow formalism of tasks, rules and fragments, which certainly simplifies their representation when compared to more expressive adaptation formalisms such as, for instance, adaptation grammars. Also, Wotan courses are "closed-corpus", in the sense that all available content must be present when the course starts, and run-time queries to select contents based on user model are restricted to contents explicitly listed by the course author during course design. In systems with complex adaptation mechanisms or open corpora, visualization is still required; indeed, course authors may find outcomes even harder to model during the course design phase. However, it is probable that, for such cases, attempting to represent the whole course structure at once, as done in Clover, would be less appropriate. Instead, it could be more useful to represent the domain model. A course, from a particular user model's point of view, would be like a filter on the domain model graph.

WotEd also integrates a fully functional adaptation engine into the authoring tool. This is judged as an interesting contribution to adaptive hypermedia content authoring, since it allows an author to follow the behavior of students graphically on top of the very same interface used to design the course in the first place. This approach avoids having to interpret log files (a tedious process), and reuses the mental map that the author built during course creation.

Other graph visualization approaches

Many lessons can be learned from alternative graph visualization approaches. Three particularly interesting ones are the success of the comparably simpler TouchGraph interface, the use of 3D in WilmaScope, and the ease of use zoomable user interfaces (ZUIs).

Thresholded graph interfaces, such as those produced by TouchGraph (see section 5.3.1), present the disadvantage that the existence of significant paths beyond the cutoff point may well have been elided, and the user has no way to determine the existence of such paths without extensive navigation. In a clustered graph this cannot occur, since complete (although maybe abstracted) connectivity information is available at all times. On the other hand, thresholded graph interfaces are remarkably easy to use, and are probably the most popular interactive graph visualization interfaces to be found online. The lesson would be that ease of use can be much more important than theoretical robustness.

The decision to limit the application to 2D could be revised. 3D hardware is becoming ubiquitous, and graph visualization applications such as WilmaScope [50] demonstrate that in spite of the traditional problems associated with 3D graph interfaces (see section 3.2), they can yield very promising results. Although 3D graph layouts suffer from occlusion, they also have advantages, such as reduced edge crossings and more "space" in which to represent vertex clusters in expanded form, with lower distortion to the surrounding layout than would be possible in 2D. Even implementing a 2D interface on top of a 3D layer may be worthwhile. Accelerated 3D graphics hardware allows complicated calculations and rendering operations to be offloaded from the main CPU into the graphics card, providing an important boost to demanding 2D visualization applications.

Paradigms such as Zoomable User Interfaces (ZUIs, introduced in section 5.4) are also worth considering; ZUIs are extremely easy to grasp for novice users, and their fast rendering times and built-in animation support are inherently well suited to large graph visualization [121]. A problem with using ZUIs for this task is that of detail versus context; if the context changes, the whole graph layout may need to be updated, an issue that current implementations do not address. However, this can be seen as a problem with current implementations rather than with the idea of using ZUIs for graph visualization.

9.3 Future Work

It is said that in a work such as this one, development is never truly finished, it is only halted. This is certainly true in this case; there are just too many things to test, to polish, to add. But at some moment the line must be drawn. In the case of this work, it has been drawn at the point in which the approach is beginning to prove effective: a worthy competitor of trees for the particular case of AH authoring, and a promising visualization approach.

Since the author was well aware of the changes that would need to be made, design has been kept as modular as possible, allowing easy replacement of almost any part of the process. The research lines and enhancements proposed in this version would require a large investment of time an effort to be tested and refined. Even though only some of them will be actively pursued, they are all worth mentioning.

Graph clustering and cluster hierarchies

Clustering generation is at present very simplistic, and is only effective for relatively regular graphs. In more complex or dense graphs, more general clustering engines would need to be applied. Additional clustering rules could still be used as an initial step, as hinted in section 6.2.2.

The resulting cluster hierarchy may still require user customization. A fully automatic clustering engine can not expect to find the same clusters and cluster labels that a human would find, if only because the algorithm can only work with the graph's explicit information, but cannot yet understand its (possibly subjective) semantics. User edits are allowed in WotEd, but at present the default rule-based clustering engine does not preserve these edits when automatic clustering is repeated. To preserve user edits, the clustering engine would have to be "incremental", in the sense that changes with a lower degree of importance than a given threshold would not be allowed to affect user-modified parts. When this threshold was crossed, user intervention could be requested, or automatically derived changes could be directly applied.

The efficiency of the internal cluster hierarchy representation may also need to be addressed. The graph hierarchy structure described in [109] is a prime candidate to replace the current ad-hoc implementation. Incremental clustering update could also be significantly optimized, but any optimization (other than a better tree comparison algorithm) would have to be integrated with the particular clustering generation algorithm, and would have to accept and maintain possible user adjustments.

Graph hierarchies should not need not be unique. Different points of view are possible, and should be accepted by a clustered graph interface. At present, Clover has no specific support for views of a single graph that have been clustered using different hierarchies. It would be interesting to allow users to dynamically exchange the clustering hierarchies used on a graph, enabling a visual comparison of the differences and coincidences between multiple alternative cluster hierarchies. The problem of hierarchy comparison is frequently encountered in many disciplines, such as biologists comparing phylogenetic trees. In this case, the most common approach is to visually represent both trees with similar layouts, and highlight differences and matches. The problem of comparing graphs with different superimposed clustering hierarchies can be attacked in a similar manner (comparing only the hierarchies themselves), but it would be even better to enable comparison of graphs, using layouts as similar as possible in both.

Incremental layout

Support for user changes to graph layouts is implemented in the form of a layout cache. Since the layout of a particular graph view is saved, and the saved version is updated after manual adjustments, the adjustments will still be there when the same view is later revisited. However, cache entries that were made prior to the adjustment are currently unaffected. A cross-cache incremental layout would need to be performed to truly accept user adjustments to automatic layout. As an added constraint, it must be noted that adjustments to one view may conflict with adjustments to another view. Users would therefore need to be aware of (at least some of) these cascaded layout updates, which would be performed in the background to avoid lack of interface responsiveness.

Better layout techniques should be explored, since layout quality and speed are major components of the user experience. A particular point that will need to be revised is the vertex overlap removal strategy. The use of the Force-Transfer Algorithm described in [83], while extremely fast, introduces a measure of noise into the drawing, since it is not truly force-directed and therefore departs from the physical model used for other layouts. Incremental extensions to the main layout algorithms could be investigated, extending the very efficient force-directed layout algorithm described in [134], yielding interactive update speeds even for large graphs.

Adaptive Hypermedia

Filtering is currently underused in WotEd. Only very simple filters have been tested, but full user-model filters would be a very important addition for authors, allowing them to explore course contents from a particular user's point of view. A library of user models could allow quick testing of the main facets of the course, allowing test cases to be developed for particularly difficult areas. Addition of complex filters would not substantially complicate the interface, except for the fact that there would be frequent changes of hierarchies (recall that filtering is performed prior to automatic clustering); such changes could be hard for users to follow.

Another extension would be to integrate tutoring into the WotEd application, allowing tutors monitoring a course to open instant-messaging sessions with students that appear to be stuck, maybe even presenting aid requests as small notes directly on the interface. Student collaboration could also use a simplified monitoring interface to stay aware of each other's virtual location.

Monitoring and log file analysis currently requires actual user data. Extending WotEd with quick artificial "log files" creation would allow authors to debug adaptive courses without leaving the authoring interface. Better visualization of existing user log files would, and support for the aggregation of several user navigation patterns in the course graph would also be natural extensions. Several systems already display web site usage as graphs, with links that are more frequently followed represented with thicker edges than those that are less popular. Indeed, there is already work underway to aggregate individual user histories and automatically generate populations of user models and plausible navigation paths in Wotan [31], with the goal of discovering course weaknesses and stress-testing courses with a variety of different user models.

Interface enhancements

It would be convenient to allow users to explore vertex contents as cluster trees before actually expanding them, and allow expansion to a particular level. A similar strategy could be used to control cluster collapse operations. The interface would require two small icons to be added to each cluster vertex, say a small upright triangle (for collapse) and a downwards triangle (for expansion), with the following semantics:

- Collapse – the list of parents of the vertex would be shown as a temporary popup. Hovering the mouse pointer over any of the displayed parents would highlight all currently visible vertices that shared the same parent. Clicking on any of the parents would initiate the collapse.

- Expand – (only visible for cluster vertices) the tree of children of the cluster vertex would be shown, as a temporary popup. This would be a multilevel-popup, where hovering over an item with subitems (a child with children of its own) would display the next level of items. Clicking on a child would trigger the expansion.

One-click expansion on either of the two triangles would bring about the corresponding default, single-level collapse or expand.

Another interesting visual aid would be to present cluster contents as small sub graphs on demand. This could be achieved integrating Clover into a Zoomable User Interface (ZUI) framework. Although in ZUIs interface elements are expected to keep a constant degree of zoom overall, an “asymmetrical” graph ZUI would be very similar to a clustered graph. For instance, if clustering expansion and collapse could be performed fast enough, these operations could be implemented as a kind of “vertex zoom”, and bound into a traditional zoom trigger such as the mouse wheel. To change the degree of (semantical) detail of a part of a graph, one would only have to (semantically) zoom in and out – triggering the necessary expansion and collapse of vertices. This could exploit the idea of variable representations for different degrees of detail found in many ZUIs; at a low degree of detail, only the vertex label would be shown. Zooming into the cluster vertex, a miniature view of its contents would begin to unfold, until the component vertices reached their full sizes.

Part IV

Appendix

Apéndice A

Introducción

Este trabajo describe una propuesta para la visualización y creación de Hipermedia Adaptativa, basada su representación mediante grafos clusterizados (agregados) jerárquicamente. La propuesta es extensible a cualquier campo que se pueda representar mediante grafos con propiedades de "mundo pequeño".

La introducción comienza con una descripción de la motivación inicial y los objetivos buscados. En el siguiente apartado, se procede a enumerar y describir una serie de áreas de conocimiento relacionadas con la propuesta, y, a continuación, se presenta la propuesta en sí. En la última sección se describe, a grandes rasgos, la organización del resto del trabajo.

A.1 Motivación y objetivos

La representación mediante grafos es una práctica habitual en un gran número de dominios, tales como la programación, el diseño y mantenimiento de redes de comunicaciones o la representación de mapas de correferencias. En muchos de estos casos, la estructura de los grafos resultantes contiene patrones que una buena representación debería realzar. Por ejemplo, en grafos que representen llamadas a funciones dentro de una sección de código fuente, o en grafos que representen correferencias científicas, es habitual encontrar grupos de vértices fuertemente interrelacionados. En inglés, estos grupos o agregaciones se denominan *clústers*.

El uso de interfaces interactivas basadas en grafos resulta de evidente utilidad para revisar y editar datos correspondientes a cualquiera de los dominios anteriormente mencionados. No obstante, a medida que el tamaño de los grafos aumenta, el uso de interfaces tradicionales conlleva una progresiva sobrecarga de información, asociada además a tiempos de procesamiento cada vez más prolongados. La sobrecarga de información está relacionada con la dificultad asociada a mostrar tanto detalles de bajo nivel como una visión general "de conjunto" dentro de la misma interfaz. Aunque éste es un problema común a todas las interfaces de usuario, en el caso de la representación de grafos se ve exarcebado: añadir o quitar aristas individuales de un grafo puede ocasionar cambios importantes en su estructura general; y en este caso, la representación gráfica tendría que actualizarse considerablemente para evidenciar la nueva estructura. Actualizar la representación de un grafo de tamaño considerable es una operación costosa; y un cambio de representación demasiado brusco puede causar desorientación en el usuario de la interfaz.

En nuestro caso, las dificultades que se acaban de describir, habituales cuando se trata de grafos particularmente complejos, o de tamaño relativamente grande, surgieron mientras se diseñaba una nueva herramienta de autor para la creación de cursos hipermedia adaptativos. Inicialmente, se intentó usar una representación en forma de árbol, ya que la estructura de los cursos correspondía, en gran medida, a tal representación: en este sistema, los cursos estaban constituidos por tareas, cada una de las cuáles podía descomponerse en subtareas, y así sucesivamente. Sin embargo, la estructura no era realmente jerárquica, ya que algunas subtareas resultaban ser accesibles desde más de una tarea o subtarea de nivel superior. Esto ocasionó grandes dificultades a la hora de intentar representar la estructura de tareas, que obedecía a una estructura de grafos dirigido acíclico, mediante árboles.

Los problemas que presentaba esta primera interfaz motivaron el desarrollo de una nueva, en la cual se representaban los cursos mediante grafos "tradicionales". No obstante, los grafos eran relativamente grandes y complejos, difíciles de presentar correctamente en una pantalla de ordenador, y la interacción con esta segunda interfaz resultaba mucho más compleja que el uso de los árboles de la interfaz anterior, donde, por ejemplo, el nivel de detalle se podía modificar fácilmente mediante la expansión o contracción de las ramas correspondientes. Entonces surgió la idea de tratar de combinar ambas interfaces: ya que la estructura de los cursos era "casi" de árbol, y éstos son fáciles de usar, ¿porqué no combinar sus ventajas con la capacidad expresiva de los grafos? ¿Sería factible realizar esta combinación, sustituyendo *clústers* (grupos de vértices fuertemente relacionados) por ramas?

Una opción es la siguiente: al igual que en las ramas inferiores de los árboles, que se pueden ver como descendientes de las ramas que las contienen, los *clústers* de un grafo se pueden ver como descendientes de aquellos que, a su vez, los contienen. El resultado de esta operación sería una jerarquía de *clústers*. Las operaciones de expansión y contracción de un *clúster* se podrían definir en función de lo anterior, y el problema estaría resuelto, excepto por algunos detalles muy importantes. En primer lugar, mientras la definición de lo que es una "rama" de un árbol no presenta ninguna ambigüedad (todos los descendientes de un nodo dado), la de un *clúster* no es tan evidente, siendo necesario utilizar heurísticas adecuadas, e incluso la intervención directa del usuario, para definir y mantener la jerarquía de *clústers*. En segundo lugar, es mucho más complicado disponer gráficamente los vértices de un grafo (el problema de la "disposición", también llamado "dibujo de grafos", ha atraído una atención considerable en los últimos años) que los nodos de un árbol. La disposición gráfica en pantalla de grafos pequeños, o estáticos aunque sean de gran tamaño, no es un problema, pero es necesario que la disposición gráfica en pantalla sea automática si se requiere que las porciones visibles del grafo puedan variar en dinámicamente como resultado de expansiones o contracciones del mismo, tal y como sucedería con las ramas de una interfaz de tipo árbol. También, en este último caso, es necesario tomar otras medidas adicionales sobre la representación visual del grafo, encaminadas a evitar la desorientación del usuario, ya que, después de una expansión o contracción, deberá poderse visualizar el "mismo" grafo, pero con un nivel de detalle distinto. Es decir, la nueva disposición de los vértices y aristas no debe variar demasiado con respecto a la anterior, o el usuario se verá forzado a "reaprender" la estructura del grafo tras cada operación de navegación, lo cual sería, desde el punto de vista de la usabilida de la herramienta, desastroso. Finalmente, en una interfaz de tipo árbol, debido a la correspondencia directa entre estructura y disposición gráfica, la actualización de la representación visual tras un cambio estructural es relativamente sencilla. Esto no se cumple para el caso de grafos agregados jerárquicamente, cuya actualización puede requerir también cambios en la estructura de los *clústers* que definen su jerarquía.

Sin embargo, los obstáculos anteriores no parecían insuperables, ya que existen abundantes publicaciones acerca de la mayor parte de los problemas individuales (agregación, disposición gráfica, desorientación, etcétera). En nuestro caso, tras una fase inicial de análisis, se decidió separar claramente los módulos directamente asociados a la visualización y edición de grafos agregados jerárquicamente de aquellos que estaban relacionados con dominios concretos (como por ejemplo todo lo relacionado con la creación y edición de Hipermedia Adaptativa). Esta decisión dio lugar a un *framework* (entorno) de visualización de grafos basado en la agregación jerárquica de los mismos. Poco después, el diseño de dicho entorno para la visualización de grafos, así como su implementación, se vieron fuertemente impulsados por la lectura del artículo de Watts y Strogatz [137] sobre grafos de mundo pequeño – grafos que son altamente estructurados (y, por tanto, clusterizables), pero no enteramente regulares – y su prevalencia en muchos dominios del mundo real. Igualmente, estas ideas motivaron la utilización del entorno previamente diseñado en diversas áreas de aplicación, algunas de las cuales están muy alejadas de los sistemas hipermedia adaptativos iniciales.

A.2 Aspectos teóricos

La propuesta que se esboza en la sección anterior tiene relación con varias áreas de conocimiento. Los primeros capítulos de este trabajo se incluyen en la Parte I, Preliminares, donde se describen las principales áreas de conocimiento asociadas a la propuesta. A continuación se presentan brevemente los campos que se tratarán en la Parte I, así como las razones que han sugerido incluirlos en dichos capítulos.

Como hemos mencionado, este trabajo comenzó como un intento de conseguir una mejor visualización para un sistema de cursos hipermedia adaptativos. El objetivo de la Hipermedia Adaptativa (abreviada en inglés como AH) es adaptar espacios hipermedia a usuarios individuales, en lugar de usar la misma versión del espacio para todos los posibles usuarios. Esta adaptación puede abarcar tanto los contenidos en sí (ya se trate de texto, imágenes, sonidos o animaciones, denominados colectivamente *media*) con los posibles hiperenlaces que se puedan definir entre ellos. La hipermedia adaptativa hace uso de un modelo de usuario (una representación de las características, trasfondo y objetivos del usuario) que debe ser construido y actualizado por el sistema AH. El modelo se usuario se utiliza para decidir qué mostrar a cada usuario, y cómo presentarlo. Aún compartiendo un objetivo general y el concepto de modelo de usuario, los sistemas AH difieren en sus campos de aplicación (por ejemplo, educación, búsqueda o referencia), y existen múltiples diferencias en torno al tipo de modelo de usuario y su representación, adquisición y actualización, el tipo de adaptación que puede realizar el sistema, y las técnicas usadas para implementarlas.

Por otra parte, el uso de Hipermedia Adaptativa requiere la preparación de distintas vistas sobre el dominio utilizado por el sistema. Así, los materiales que han de adaptarse serán más fáciles de crear y mantener si se gestionan como módulos relativamente independientes entre sí. Por otra parte, dado el alto coste de preparar estos materiales, es muy importante que se facilite su reutilización en otros dominios o contextos. Sin embargo, esta reutilización a gran escala requiere que se disponga de información adicional accesible a sistemas automáticos de indexación. La información adicional, también llamada *metadatos*, debería incluir información acerca de los contenidos y objetivos de cada unidad, y el contexto dentro del cual tiene sentido usarla (por ejemplo, conocimientos previos requeridos). El tratamiento automático require que los metadatos usen vocabularios estandarizados, lo cual se puede conseguir usando ontologías. Las ontologías son un formalismo para la representación del conocimiento mediante conceptualizaciones de entidades y sus interrelaciones, que se puede ampliar con soporte para razonamiento automático. Además, las ontologías son, en sí mismas, representables como grafos.

Una vez creados u obtenidos, los módulos correspondientes deben ser integrados para formar un único espacio adaptativo. La autoría de la estructura de adaptación correspondiente puede ser una tarea compleja, ya que las decisiones de adaptación se toman en función de un modelo de usuario que está siendo constantemente actualizado. Sobre este modelo cambiante, el sistema hipermedia debe decidir qué presentar, y cómo debe ser estructurado y realzado. El uso de herramientas de autor resulta imprescindible para permitir a expertos en un área crear estos espacios hipermedia que representen sus conocimientos – sin obligar a estos expertos a conocer en profundidad los entresijos del sistema en sí. Éste era precisamente el objetivo que perseguía la herramienta de autor basada en grafos que dio pie a este trabajo.

Es común usar grafos para representar la estructura definida por los enlaces de espacios hipermedia clásicos. Además, estas reresentaciones se pueden extender para abarcar la estructura de muchos sistemas hipermedia adaptativos. El estudio de los grafos en sí, como entidades matemáticas abstractas, pertenece al campo de la *Teoría de Grafos*, que a su vez forma parte del campo de la Matemática Discreta y Combinatoria. El campo relacionado del *Trazado* o *Dibujo de Grafos* (*Graph Drawing*) trata de su representación planar y espacial. En general, un grafo grande resulta difícil de dibujar e interpretar, especialmente cuando se el área de dibujo es particularmente pequeña, como por ejemplo una pantalla. En estos casos, la *clusterización* (agregación) se puede usar para abstraer detalles innecesarios. La sección sobre clusterización incluye una discusión acerca de *gramáticas de grafos*, un formalismo similar a las tradicionales gramáticas independientes de contexto que puede ser utilizado para localizar *clústers* basándose en reglas que describen patrones a buscar en la topología local del grafo.

La agregación jerárquica sólo es aplicable si el grafo contiene *clústers* fácilmente identificables. Para obtener una jerarquía es requisito, además, que esta propiedad también se cumpla para cada una de las posibles versiones resumidas del grafo. Muchos tipos de redes naturales y artificiales exhiben la llamada *propiedad de mundo pequeño* (*Small-World Property*): un alto grado de clusterización cuando se comparan con grafos generados al azar del mismo tamaño y densidad, pero una distancia media entre vértices similar al caso aleatorio. Esta propiedad está relacionada con la existencia de invarianza ante cambios de escala y autosimilaridad. La propiedad de mundo pequeño, la invarianza a escala y la autosimilaridad han sido descritas, en diferentes grados, en redes sociales (denominadas también *Friend-of-a-friend* en inglés), mapas de referencias científicas, redes de distribución eléctrica, mapas de interacción entre proteínas, y grandes secciones de la WWW. Por tanto, la agregación jerárquica puede constituir una buena solución a la hora de visualizar un amplio espectro de grafos.

Cuando se representan grafos u otros tipos de información abstracta, es importante mencionar el campo de la *Visualización de Información.* La visualización de información se centra en el estudio de la forma más efectiva de representar información abstracta a usuarios; y en este caso, está fuertemente relacionada con el *Dibujo de Grafos.* La visualización de información también se puede considerar una parte importante del campo de la *Interacción Persona-Ordenador*, que estudia la interfaz entre hombre y máquina, considerando todo el proceso de interacción y realimentación necesario para llevar a cabo tareas ante un ordenador.

A.3 Arquitectura de Clover

El núcleo de la propuesta está contenido en el diseño del entorno Clover, cuyas siglas corresponden a la traducción al inglés de *Entorno de Visualización Orientado a CLústers*. El *framework* resultante constituye una base independiente de dominio sobre la cual construir visualizaciones interactivas basadas en grafos jerárquicamente agregados. Aunque Clover se puede usar para visualizar cualquier estructura de grafo, el uso de agregación jerárquica lo hace especialmente indicado para grafos de mundo pequeño.

Clover ha sido diseñado como una serie de pasos encadenados de montaje (pipeline), ilustrada en la figura A.1. Los datos iniciales se transforman a un grafo, que luego se filtra y clusteriza, se realiza una selección de los "clústers más importantes" (que se usarán para representar al resto), se dispone gráficamente, y finalmente se representa en la interfaz que ve el usuario. Cada etapa de esta cadena transforma el resultado de la etapa anterior para su posterior transformación en la etapa siguiente, hasta que la vista resultante se le muestra al usuario. También se puede ver todo el proceso como un bucle, ya que el usuario puede manipular la vista final, ajustar el nivel de detalle de zonas del grafo (mediante la expansión y colapso de clústeres, o indirectamente mediante la selección de distintas "puntos de interés": ver triángulos negros a la derecha de la figura A.1), cambiar el filtrado, o incluso modificar el grafo base, lo cual puede ocasionar variaciones a los datos sobre los que originalmente fue construido. Cualquier cambio en la configuración de una etapa puede puede ocasionar actualizaciones en etapas posteriores, desembocando en una actualización de la interfaz presentada al usuario. Es posible mantener múltiples vistas simultáneas sobre un mismo grafo, diferenciadas entre sí sólo por cualquiera de los pasos intermedios; esto permite analizar un mismo grafo desde varios puntos de vista.

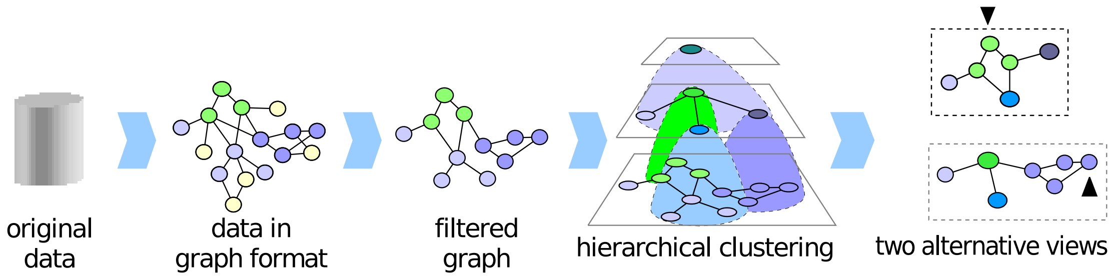


Figura A.1: *Pipeline* usado en Clover

Clover se enfrenta a muchos problemas considerados clásicos dentro del campo de la visualización de grafos. La disposición de vértices y aristas (el *layout* o trazado del grafo) se realiza de forma totalmente automática, permitiendo a los usuarios empezar a trabajar con el grafo sin necesidad de realizar una colocación manual; no obstante, se permite la realización de ajustes manuales, los cuales se mantienen en lo posible durante cambios posteriores. Cuando se producen cambios en el nivel de detalle, éstos se ven reflejados mediante modificaciones incrementales en el trazado del grafo, con los objetivos complementarios de optimizar el uso del espacio disponible mientras se minimizan, en lo posible, los cambios del trazado previo. Cualquier modificación de la representación del grafo es resaltada y animada, centrando la atención del usuario sólo en los cambios, pero manteniendo un contexto estable para evitar desorientarle. El uso de un “histórico” de trazados permite la reutilización de representaciones previas para las vistas más recientes, colaborando al mantenimiento de la sensación de contexto.

La arquitectura de Clover es altamente modular, y todas las etapas de su comportamiento pueden ser alteradas o extendidas para adaptarse a las necesidades de la aplicación que está usando el entorno. Para desarrollar una nueva visualización basada en agregación jerárquica para una aplicación dada, sólo es necesario proporcionar una etapa inicial de generación del grafo apropiada. No obstante, es posible mejorar mucho la interfaz así generada ajustando el mecanismo de filtrado, la estrategia de agregación, los parámetros de disposición, la representación de vértices y aristas o el uso de animaciones y los mecanismos de interacción para la aplicación concreta a desarrollar. Las múltiples aplicaciones basadas en Clover que se describen en este trabajo proporcionan ejemplos de lo que se puede conseguir adaptando el entorno base a diversos campos concretos.

En comparación con otros sistemas de visualización de grafos, las principales contribuciones de Clover son el uso de agregación jerárquica para permitir la representación de grafos grandes a niveles arbitrarios de detalle; la naturaleza completamente automatizada del proceso de visualización, donde un cambio en cualquier fase del proceso se propaga automáticamente, sin necesidad de intervención por parte del usuario, a todas las vistas dependientes; y el uso de un mecanismo de disposición y animación que asegura que todo cambio a una vista resulta en una actualización incremental y progresiva que resulta fácil de seguir desde el punto de vista del usuario.

La aplicación de referencia para Clover es WotEd, una herramienta de autor para el sistema de cursos hipermedia adaptativos Wotan. Usando Clover, WotEd muestra cursos hipermedia adaptativos como grafos clusterizados. Esto permite editar cursos de gran tamaño y complejidad – incluso si fueron creados por una tercera aplicación, o retocados a mano. Además, WotEd introduce un novedoso soporte para monitorizar el progreso de estudiantes dentro del curso. El sistema de monitorización permite a un autor o tutor realizar un seguimiento en tiempo real de las acciones de uno o más estudiantes, usando la misma interfaz con la que se ha estado editando el curso. Este mecanismo se basa en el soporte de Clover para propagar y animar de forma automática cualquier actualizacion a una estructura de grafo.

A.4 Estructura de la tesis

La tesis se organiza en tres partes:

Parte I – Preliminares Introduce los términos y conceptos usados en la propuesta presentada en la Parte II, proporcionando, para cada uno, una breve descripción.

Parte II – Propuesta Presenta la propuesta en sí. En primer lugar realiza una revisión de distintas estrategias de visualización de grafos. A continuación, describe el diseño de CLOVER. Los siguientes capítulos introducen aplicaciones basadas en CLOVER orientadas a la Hipermedia Adaptativa y otros dominios.

Parte III – Conclusiones y trabajo futuro Contiene una breve discusión, las conclusiones, y un esbozo general de futuras líneas de trabajo.

El apéndice A contiene la presente traducción al español del primer capítulo. El apéndice B está dedicado íntegramente a aspectos de implementación, incluyendo detalles sobre la arquitectura y los formatos de archivo usados en la Parte II.

Appendix B

Implementation notes

This appendix describes CLOVER's architecture in a more technical fashion, first providiong a general overview of the implementation, and then presenting commented APIs for the main classes. It can be used as a reference while reading Chapter 6, which contains as few actual implementation details as possible. A final section includes samples of the main file formats used in CLOVER and WOTED.

B.1 General architecture

Figure B.1 is a UML class diagram of the actual implementation of CLOVER, including only the most relevant classes. This diagram can be compared to fig. 6.2 to explore the implementation of each pipeline stage.

A note on terminology: Since CLOVER is written in JAVA, an object-oriented language, the framework has been modelled as a series of *classes*; in programming languages such as C, a *class* would correspond roughly to a data structure with a well-defined interface and the set of functions that implement that interface.

B.1.1 Model

The *BaseGraph* class represents a "base" graph, generated from some external data source; it derives from JGRAPHT's [98] *DefaultDirectedWeightedGraph*. A base graph can be filtered through one or more *FilterGraph*s, which can be used as base graphs themselves (although they do not need to store the whole graph model). *BaseGraphs* can send *StructureChangeEvent*s (SCEs) to any object that registers itself to receive them. This mechanism is used, for instance, to notify filtered graphs of any change that may occur to the graphs they were built upon.

A *ClusterHierarchy* represents a cluster hierarchy built upon a base graph. Hierarchy construction and update are delegated to a *ClusteringEngine*. The default clustering engine in CLOVER is the *SimpleRuleClusteringEngine*, based on a simplistic (ie.:

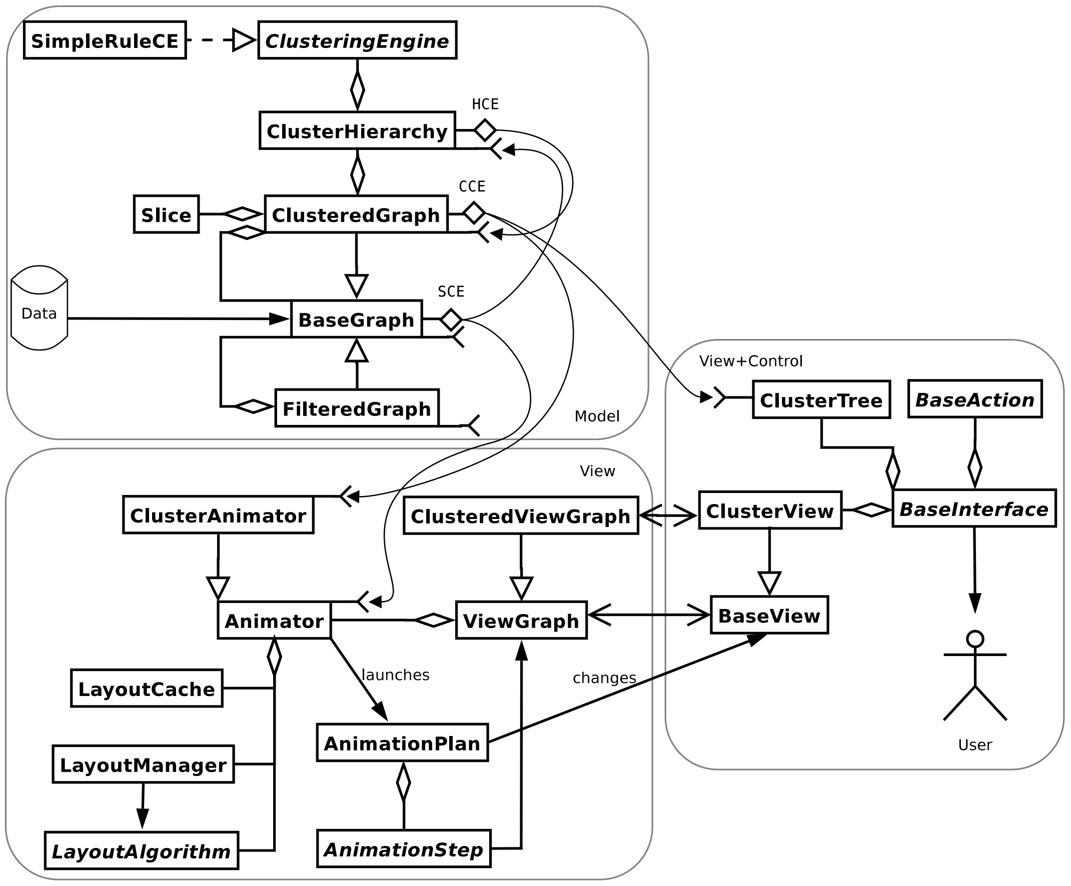


Figure B.1: Architecture of Clover. Empty triangles represent inheritance (*is-a*), diamonds represent aggregation or reference (*has-a*), and curved arrows represent event flow. Events can represent changes to *structure* (SCE), *hierarchy* (HCE) or *cluster* expansion/collapse (CCE).

suboptimal) graph grammar engine. A *ClusterHierarchy* registers itself to be notified of any changes to the base graph; when such a change occurs, the engine is queried for corresponding updates to the hierarchy itself, and a HierarchyChangeEvent (HCE) describing all changes from the prior hierarchy to the current one is generated.

A *ClusteredGraph* represents an abstracted view of a certain *BaseGraph*, and the abstracted representation itself can be queried as another *BaseGraph*. Abstraction is performed by means of a *ClusterHierarchy* superimposed on the base graph that the clustered graph was built upon; since the cluster hierarchy already has a reference to the base graph, the clustered graph uses the reference found in the hierarchy. Clustered graphs make use of a “visible slice” to keep track of what parts of their hierarchy are currently expanded. In the above diagram, this role is fulfilled by the *Slice* class. Whenever a change is performed to the base graph a given clustered graph was built upon, the hierarchy will process the change and the *ClusteredGraph* will receive the corresponding HCE. This will cause an update to visible vertices and/or degree of abstraction that will be immediately reflected in the clustered graph and its slice.

Changes to a *ClusteredGraph*, resulting either from normal navigation or cascaded changes received from its hierarchy, can cause two types of events. Since a *ClusteredGraph* is a *BaseGraph*, any changes to the edges or vertices currently displayed results in the generation and delivery of *StructureChangeEvents* (SCEs) to any interested parties. Additionally, changes corresponding to cluster expansion or collapse will cause a *ClusteringChangeEvent* to be generated and delivered.

B.1.2 View and control

So far, the first steps of the pipeline, corresponding to the model, have been discussed. The external representation of the model, however, requires layout, animation and a component that can display the results in the corresponding window toolkit.

Clover runs on top of the Swing toolkit, using JGraph's [1] Swing-compliant *JGraph* class to actually display graphs. *ViewGraph*s are used instead of *BaseGraph*s for this role. *ViewGraph*s are similar to *BaseGraph*s in the fact that they also extend JGraphT's *DefaultDirectedWeightedGraph*. Unlike *BaseGraph*s, which hold no information on how a graph should be visually represented, the main responsibility of a *ViewGraph* is to determine visual representation of graph elements (vertices or edges) prior to display. Visual aspects include font selection, foreground and background colors, icons to use, labels and tooltips, and many other options. This is possible through the inclusion of a bidirectional JGraphT $\leftrightarrow$ JGraph bridge within the *ViewGraph*, which ensures that any change performed to the *ViewGraph* is represented in the associated *BaseView*, and conversely, any change to the graphical component is immediately reflected in its model.

Although it is possible to use Clover with non-clustered graphs, the framework's main strength is the representation and manipulation of clustered graphs. *ClusterViewGraph* and *ClusterView* extend their corresponding base classes with cluster-oriented capabilities. A *ClusterViewGraph* can decorate and label graphs with cluster vertices or edges; and a *ClusterView* adds the necessary interaction mechanisms to navigate through a clustered graph.

Many of the high-level interface changes have been encapsulated within *actions*, which can be triggered by different mechanisms (such as popup menus, key bindings, or toolbar icons). Actions themselves can affect all components.

Separation between abstract, non-visual model and visual details is one reason to distinguish between abstract graphs and their corresponding *ViewGraph*s. A second reason is the need for animation. Due to the bidirectional nature of the bridge between *ViewGraph* and *BaseView*, changes to the former are immediately visible within the latter. However, some animations may require elements to be presented only after a previous animation finishes execution. Separating the "true" model from the represented model allows views to lag behind as required, enabling such animations.

Clover includes a higher-level component designed for applications where the main interface is a clustered graph; the *BaseInterface* displays toolbars that allow control of this clustered graph, a main window with a set of tabs, each of them a *ClusterView*, and a side pane with a *ClusterTree*. These classes will be explored in greater detail in the control section.

B.1.3 Animation and layout

Animation of a *BaseView* is performed through its associated *Animator*. Animators run *AnimationPlan*s, which are composed of one or more sequential sets of simultaneous *AnimationStep*s. A step may involve altering the representation of one or more edges or vertices directly on the *BaseView*, or require structural changes to the graph, which would be performed on the *ViewGraph* instead (recall that *ViewGraph*s and *BaseView*s are tightly coupled: any change on one is reflected on the other). Animation plans can be generated and launched as a result of changes to a *BaseGraph* or a *ClusteredGraph*, but this need not bee the case. Highlights and other visual effects are very easily accomplished through animations.

The *ClusterAnimator* class extends the default animator with support for *ClusteringChangeEvent*s. These events are animated differently than non-clustering related changes to a graph. Animators make use of a *LayoutManager* to generate the layouts that will be shown at the end of each intermediate step of their animation plans. Layouts themselves are subclasses of *LayoutAlgorithm* (such as *VarLengthFDL*), and are applied sequentially by the layout manager as requested by the animator. In order to avoid recomputation of previous layouts, and to improve the consistency of navigation operations on the graph, a *LayoutCache* is maintained.

B.2 Clover API

This is an abbreviated API of the Clover framework, covering the main classes found in the previous section. Each class description includes a short overview the design decisions involved, but detail is in general kept to a minimum. For instance, utility methods that do not form part of the main interface of a class have not been included. Full details can be found in the project's website, available at `http://tangow.ii.uam.es/clover`.

B.2.1 BaseGraph

By extending JGraphT's *DefaultDirectedWeightedGraph*, methods for efficient vertex adition and removal, edge adition and removal, connectivity queries and so on are gained for free. Use of *generics*, a mechanism similar to C++ templates, allows edges to be created and returned in a typesafe manner as provided by the *EdgeFactory*.

Vertices can be of any type (all Java classes are implicit subclasses of *Object*), but type safety through the use of generics has not been ensured, since in clustered graphs vertices can be either cluster-vertices or actual base graph vertices, and imposing a common subclass for both would not yield any advantage.

The event mechanism offered in JGraphT was deemed insufficient (events did not include enough information); this required the addition *StructureChangedEvent*s and associated management methods. Making the *BaseGraph* a listener to its own events simplifies maintenance: to perform a structural change on a *BaseGraph* or any of its subclasses, the subclasser need only be capable of creating an appropiate change event and deliver it to the intended recipient. The same event can then be recast to all “downstream” listeners.

```
public class BaseGraph extends DefaultDirectedWeightedGraph
    implements StructureChangeListener {

    // build
    public BaseGraph(DefaultEdgeFactory edgeFactory);

    // structure change events
    public void addStructureChangeListener(StructureChangeListener l)
    public void removeStructureChangeListener(StructureChangeListener l)
    public void structureChangePerformed(StructureChangeEvent evt)

    // unique ID, and labels for vertices and edges
    public String getId(Object vertex)
    public String getVertexLabel(Object o);
    public String getEdgeLabel(Edge e);
}
```

B.2.2 StructureChangeEvent

Events of this type encode changes to the vertices or edges of a graph, and their approximate cause. Events are of “Normal” cause if they involve *true* addition or removal of vertices or edges, as opposed to the virtual addition or removal found in “Clustering” events. If nothing is added or removed at all, then the event is considered to be “AttributeOnly”.

Vertices or edges that are marked as “Changed” require update. For example, if the label of a vertex changes, dependent views would need to be notified in order to display an updated label.

```
public class StructureChangeEvent {

    // change types; normal = base graph edges added/removed
    public enum ChangeType { Normal, AttributeOnly, Clustering };

    // construction
    public StructureChangeEvent(BaseGraph source, ChangeType changeType);

    // initialization & query
    public ChangeType getChangeType();
    public ArrayList getAddedVertices();
    public ArrayList<Edge> getAddedEdges();
    public ArrayList getRemovedVertices();
    public ArrayList<Edge> getRemovedEdges();
    public ArrayList getChangedVertices();
    public ArrayList<Edge> getChangedEdges();

    // query
    public BaseGraph getSource();
    public String getDescription();
}
```

B.2.3 FilteredGraph

A filtered graph listens to the events of its *BaseGraph*; however, since it is itself a subclass of *BaseGraph*, it cannot use the default *structureChangePerformed()* method for this task: *updateFromUpstream()* is invoked instead.

The JGraphT [98] library supports filters as `Subgraphs`, with the same semantics, and automates their synchronization when changes are performed on the base graph with a listener model. However, Clover's event system was preferred to JGraphT's implementation, as it was perceived to be more consistent with the rest of the framework.

When the filtered graph is constructed, it is associated with a *Filter* that will be used to decide which edges and vertices "make the cut". The filter is also queried during all subsequent updates.

```
public class FilteredGraph extends BaseGraph {

    // creation and access
    public FilteredGraph(BaseGraph base, Filter filter);
    public BaseGraph getBase();
```

```
        public void setBase(BaseGraph base);
        public Filter getFilter();
        public void setFilter(Filter filter);

        // triggered upon structure changes from upstream
        protected void updateFromUpstream(StructureChangeEvent evt);

        // trigger full refilter
        protected void refilter();
    }
```

B.2.4 Filter

A simple filter interface. The division between normal filters and efficient filters made in JUNG [101] was considered not necessary, although it could be incorporated into further subclasses. CLOVER filters can be plugged into any *FilteredGraph*, implementing the *Strategy* design pattern described in [67].

```
    public interface Filter {

        // save and restore
        public void save(Element e);
        public void restore(Element e);

        // edge and vertex filtering logic
        public boolean isEdgeValid(Edge e);
        public boolean isVertexValid(Object v);
    }
```

B.2.5 Cluster

The *Cluster* class contains clustering information. The implementation has not been optimized for efficiency when querying edge (see performance concerns in section 6.2.3).

Since the *Cluster* class extends SWING's *DefaultMutableTreeNode* class, cluster hierarchies can be easily represented in any SWING component that displays trees. *DefaultMutableTreeNodes* also include a complete API for tree node data and common operations.

Listings of component vertex ids in canonical order are used in later stages when loading and saving cluster hierarchies. Several other utility methods are provided for frequently performed operations, such as hierarchy traversals.

```
public class Cluster extends DefaultMutableTreeNode {

    // creation of leaf and inner cluster vertices
    public Cluster(BaseGraph g, Object v);
    public Cluster();

    // internal vertices wrap a cluster
    public static class Vertex {
        public Cluster getCluster();
    }

    // listing = sorted list of ids of all leaf vertices within the cluster
    public String getListing(BaseGraph base);
    public static ArrayList<String> parseListing(String l);

    // insert and remove clusters into this one
    public void insert(MutableTreeNode o, int );
    public void remove(int );

    // raw query
    public HashSet<Edge> getOutgoing();
    public HashSet<Edge> getIncoming();
    public List<Cluster> localOutgoingNeighbors();
    public static HashSet getVertices(Collection<Cluster> clusters);

    // other queries
    public Edge getLeafEdgeTo(Cluster c);
    public boolean hasEdgesTo(Cluster c);
    public Set<Edge> edgesTo(Cluster c);
    public Object getVertex()
    public Object getFirstLeafVertex();
    public Cluster getLastClusterFor(Object v);
    public Cluster clusterForVertex(Object v);
    public boolean isLeafCluster();
    public Cluster getRootCluster();
    public Cluster getParentCluster();
    public List<Cluster> getAncestors();
    public List<Cluster> getDescendants();
    public static List<Cluster> getAncestors(Collection<Cluster> clusters);
    public static List<Cluster> getDescendants(Collection<Cluster> clusters);
}
```

B.2.6 ClusterHierarchy

Cluster hierarchies delegate actual hierarchy creation and update to *ClusteringEngine*s. Their main responsibilities are to create and manage hierarchy changes, by registering themselves to receive changes to a graph's structure, creating the suitable hierarchy change events, and delivering them to all associated listeners.

The *createChangeEventFor()* method is provided for use by clustering engines that do not provide after-update difference tracking. This method creates the suitable hierarchy change set from the old and new hierarchies.

Save and restore to and from XML snippets is also included; the engine-specific aspects of saving a restoring a given cluster hierarchy are delegated to the actual engine.

```
public class ClusterHierarchy implements
    StructureChangeListener, HierarchyChangeListener {

    // build new, or from saved state
    public ClusterHierarchy(BaseGraph base, Object rootVertex,
        ClusteringEngine engine);
    public ClusterHierarchy(Element e, BaseGraph base,
        ClusteringEngine engine);

    // access
    public ClusteringEngine getEngine();
    public Cluster getRoot();
    public BaseGraph getBase();
    public void setBaseGraph(BaseGraph base, Object rootVertex);

    // hierarchy change event management
    public void addHierarchyChangeListener(HierarchyChangeListener l);
    public void removeHierarchyChangeListener(HierarchyChangeListener l);
    public void hierarchyChangePerformed(HierarchyChangeEvent evt);

    // structure change reception; SCs may trigger hierarchy change events
    public void structureChangePerformed(StructureChangeEvent evt);

    // hierarchy change creation from a newClustering resulting from a SCE
    public void createChangeEventFor(Cluster newClustering,
            StructureChangeEvent structureChange,
            HierarchyChangeEvent hce);

    // save and restore
    public void save(Element e);
    public void restore(Element e);
}
```

Tree Matching

The tree difference algorithm used within *createChangeEventFor()* when locating differences between old and new clusterings is presented in the following pseudocode listing:

```
findTreeChanges(root_o root_n delta_g delta_t) {

  u_o = a new size_queue
  u_n = a new size_queue with { x : x in delta_g.added }

  s_o = { x : x in root_o.leaves AND NOT x in delta_g.removed }
  s_n = { x : x in root_n.leaves }

  delta_t.matched = { ( x y ) : x in root_n AND y in root_o }

  outer: while NOT delta_t.matched.contains ( root_o )

    s_o = collapse_level( s_o )
    s_n = collapse_level( s_n )

    if ( s_o.empty OR s_n.empty ) break outer

    u_o += s_o
    u_n += s_n

    inner: do
      // matches smallest-to-smallest; a is from old, b is from new
      for a in u_o, b in u_n

        if NOT delta_t.matched.contains ( b ) AND matches ( a b )

          // remove unwanted children from old
          for ca in children ( a )
            if u_o.contains ( ca )
              u_o -= ca
              // if unreferenced, remove (and subsume removed children)
              if NOT delta_t.matched.contains_dest ( ca )
                delta_t.removed += ( a ca )
                delta_t.removed -= { ( x y ) : x in descendants ( ca ) }
            else if exist cb in children ( b ) : matches ( cb ca )
              // discard if previously matched to another cluster
              if delta_t.matched ( cb ) != ca
                delta_t.removed += ( a ca )
            else
              delta_t.removed += ( a ca )

          // add new children to old
          for  cb in children ( b )
            if delta_t.matched.contains ( cb )
              ca = delta_t.matched ( cb )
              delta_t.matched -= cb
              // avoid duplicate introduction
              if NOT children ( a ) contains ( ca )
                delta_t.added += ( a ca )
            else
              if u_n.contains ( cb )
                u_n -= cb
              delta_t.added += ( a cb )

          u_o.remove ( a )
          u_n.remove ( b )
          delta_t.matched += ( b a )

          continue inner

      // loop ends when no more matches
      break inner
    end inner
  end outer
```

B.2.7 ClusteringEngine

Clustering engines are required to create a clustering hierarchy from a given root vertex, and to update that hierarchy after a given change in the base graph. The second operation is only possible for certain clustering algorithms; if the chosen algorithm does not support it, the operation can be performed by recreating the hierarchy from scratch, and calling *hce.getSource().createChangeEventFor()* to locate the differences between the old and new hierarchies.

The relation between clustering hierarchies and their clustering engines implements the *Strategy* design pattern described in [67].

```
public interface ClusteringEngine {

    // save and restore settings
    void save(Element e);
    void restore(Element e);

    // create or update hierarchy
    Cluster createHierarchy(BaseGraph base, Object rootVertex);
    Cluster updateHierarchy(Cluster root, BaseGraph base,
        StructureChangeEvent sce, HierarchyChangeEvent hce);
}
```

B.2.8 SimpleRuleClusterer

Implements a graph gramar engine, where rules that are triggered result in new clusters. Rule application can be controlled to ignore certain edge types or reverse their directions. Priority is implicit in the order of rule execution; lower-priority rules are only executed if higher-priority ones fail to trigger. Subclasses can easily tune the rules that are evaluated by overriding the *initRules()* method.

```
public class SimpleRuleClusterer implements ClusteringEngine {

    public SimpleRuleClusterer();

    // hook for subclassers to use different rules
    public void initRules();

    // implementation of ClusteringEngine interface
    public Cluster createHierarchy(BaseGraph base, Object rootVertex);
    public Cluster recreateHierarchy(BaseGraph base, Object rootVertex,
        StructureChangeEvent sce, HierarchyChangeEvent hce);

    // save and restore
    public void save(Element e);
    public void restore(Element e);

    // default set of rules, in order of priority
    public static class SingleParentOfTerminals extends ClusteringRule {}
```

```
        public static class SingleParentOfAlmostTerminals
            extends ClusteringRule {}
        public static class SharedParentOfTerminals extends ClusteringRule {}
        public static class ParentOfSomething extends ClusteringRule {}

        // a clustering rule
        public static abstract class ClusteringRule {
            public abstract ArrayList applyTo(DirectedGraph g, Object v);
            public abstract String getDescription();

            // 'not valid' edges are ignored within rule; edges can be reversed
            public void setValidator(EdgeValidator validator) {
            public void setReversed(boolean reversed);

            // connectivity queries
            public ArrayList outgoingNeighborsOf(DirectedGraph g, Object v);
            public ArrayList incomingNeighborsOf(DirectedGraph g, Object v);
            public int outDegreeOf(DirectedGraph g, Object v);
            public int inDegreeOf(DirectedGraph g, Object v);
        }

        // edge validator used within rules
        public interface EdgeValidator {
            public boolean isEdgeValid(Edge e);
        }
    }
```

B.2.9 HierarchyChangeEvent

A change in a hierarchy. This event encapsulates changes both to graph and to its superimposed cluster hierarchy; a single *HierarchyChangeEvent* should allow a clustered graph built upon that hierarchy to update itself completly.

Since all events are delivered to listeners *after* being performed on their source, removed clusters will no longer be attached to the current hierarchy. In order to match "old" clusters to "new" ones, the *getMatchedClusters()* method is provided.

```
    public class HierarchyChangeEvent {

        // construction
        public HierarchyChangeEvent(ClusterHierarchy source, String description);
        public void insertRemovedCluster(Cluster parent, Cluster removed);
        public void insertAddedCluster(Cluster parent, Cluster added);
        public void augmentChangesWithAddedAndRemoved();

        // root changes (entire hierarchy is overhauled)
        public void setRootChange(Cluster oldRoot);
        public boolean isRootChange();

        // initialization & query
        public HashMap<Cluster,Cluster> getMatchedClusters();
        public HashMap<Cluster,ArrayList<Cluster>> getAddedClusters();
        public HashMap<Cluster,ArrayList<Cluster>> getRemovedClusters();
        public HashSet<Cluster> getChangedClusters();
        public HashSet<Edge> getAddedEdges();
        public HashSet<Edge> getRemovedEdges();

        // query
        public ClusterHierarchy getSource();
        public String getDescription();

        public Object getVisibleRepresentativeFor(Object v, Slice s);
        public Cluster getMatchedVersion(Cluster alien);
    }
```

B.2.10 ClusteredGraph

Clustered graphs include numerous methods to create and manage cluster expansions and collapses triggered by graph navigation. The degree-of-interest algorithm used for implementing a Furnas-style semantical fisheye lens is also located within this class.

Special care must be given to update the graph after a change of hierarchy or graph structure. This logic is encapsulated within the *hierarchyChangePerformed()* method.

```
    public class ClusteredGraph extends BaseGraph
            implements HierarchyChangeListener, ClusteringChangeListener {

        // creation
        public ClusteredGraph(ClusterHierarchy hierarchy);

        // access
        public void setHierarchy(ClusterHierarchy hierarchy);
        public ClusterHierarchy getHierarchy();
        public BaseGraph getBase() {
        public Slice getSlice() {

        // clustering event management / hierarchy event reception
        public void addClusteringChangeListener(ClusteringChangeListener l);
        public void removeClusteringChangeListener(ClusteringChangeListener l);
        public void clusteringChangePerformed(ClusteringChangeEvent evt);
        public void hierarchyChangePerformed(HierarchyChangeEvent hce);

        // query and set PoI; setting the PoI may trigger a clustering event
        public Object getPointOfInterest();
        public void setPointOfInterest(Object v);

        // event creation
        public ClusteringChangeEvent createCollapseEvent(Object v);
        public ClusteringChangeEvent createExpandEvent(Object v);
        public ClusteringChangeEvent createMakeVisibleEvent(Object v);
        public ClusteringChangeEvent createPoIChangeEvent(Object nextPoI,
                Set frozen, int focusSize, int maxClusters);

        // extend behaviours from BaseGraph
        public String getId(Object vertex);
        public String getVertexLabel(Object v);
        public String getEdgeLabel(Edge e);
    }
```

B.2.11 Slice

A slice is the set of clusters which are currently visible within a clustered graph. As such, it should be a cut-set of the graph hierarchy. During application of changes found in a hierarchy change event, however, gaps may occur; the *findHoles()* returns a list of such gaps.

Other important methods include expansion and collapse of clusters within the slice, location of differences between two given slices, and location of the "visible representatives" for any given cluster.

```
public class Slice extends HashSet<Cluster> {

    // creation
    public Slice();
    public Slice(Collection<Cluster> collection);

    // check for malformed slices
    public ArrayList<Cluster> findHoles(Cluster root);

    // expand, collapse
    public ArrayList<Cluster> expand(Cluster c);
    public ArrayList<Cluster> collapse(Cluster c);
    public void collapseAllExcept(Set<Cluster> toPreserve);

    // find difference between two slices of same hierarchy
    public int diff(Slice s2, ArrayList<Cluster>toExpand,
        ArrayList<Cluster>toCollapse);

    // query; mostly for visible ascendants and descendants
    public ArrayList<Cluster> getDescendantsOf(Cluster c);
    public Cluster getRepresentativeFor(Cluster c);
    public boolean containsClusterOrAncestor(Cluster c);
    public boolean isCovered(Cluster c);
    public boolean isUncovered(Cluster c);
    public ArrayList<Cluster> clustersWithEdgesFrom(Cluster c);
}
```

B.2.12 ClusteringChangeEvent

Cluster change events are issued to animators and trees to notify them of one or more expansions and collapses within a slice. Once *commit()* has been called, the individual changes will have been ordered so that their execution one after another is guaranteed to succeed. This is important because, for instance, it is an error to try to expand a cluster before it has been made visible; and this may require expanding another cluster first.

```
public class ClusteringChangeEvent {

    // creation & construction
    public ClusteringChangeEvent(ClusteredGraph source,
        Object initialPoI, Object finalPoI,
        String description);
    public void addCollapsed(Cluster c);
    public void addExpanded(Cluster c);
    // query
    public Object getInitialPoI();
    public Object getFinalPoI();
    public ClusteredGraph getSource();
    public String getDescription();

    // undo support
    public ClusteringChangeEvent getUndoEvent();
    public boolean isUndoEvent();

    // execution on top of clustered graph
    public void commit(ClusteredGraph g);

    // post-execution query; collapses and executions are ordered
    public ArrayList<ClusteringAction> getExpanded();
    public ArrayList<ClusteringAction> getCollapsed();

    // individual expansion or collapse within the main event
    public static class ClusteringAction {
        public Cluster getCluster();
        public StructureChangeEvent getStructureChange();
        public int getLevel();
    }
}
```

B.2.13 ViewGraph

View graphs act as bridges, including within them a *JGraphModelAdapter* which is used by the *BaseView* or *ClusterView* to actually represent the graph.

Multiple methods for decorating vertices are defined; additionally, a “connectivity inspector” is provided, in charge of locating connected components within the graph. This is specially important during layout, since vertices from different components should usually not affect the fine placement of each other; and components themselves should not overlap.

```
public class ViewGraph extends ListenableDirectedWeightedGraph
        implements JGraphModelAdapter.CellFactory {

    // constructor and base graph access
```

```
        public ViewGraph(BaseGraph base);
        public BaseGraph getBase();
        public void setBase(BaseGraph base);

        // bridge to BaseView
        public JGraphModelAdapter getModelAdapter(
                AttributeMap vertexAttribs,
                AttributeMap edgeAttribs);
        public ArrayList getCells();

        // interface vertex and edge creation and decoration
        public void addVertex(Object v, Rectangle2D initialBounds);
        public DefaultEdge createEdgeCell(Object o);
        public DefaultGraphCell createVertexCell(Object o);
        public void decorateVertexCell(DefaultGraphCell c);
        public void decorateEdgeCell(DefaultEdge de);

        // find connected components in displayed graph
        public ConnectivityInspector getConnectivityInspector();

        // labels and tooltips that will be shown in interface
        public String getVertexLabel(Object o);
        public String getEdgeLabel(Edge e);
        public String getVertexToolTip(Object o);
        public String getEdgeToolTip(Edge e);
    }
```

B.2.14 BaseView

This class extends JGraph's *JGraph* class. Additions include a "layout zoom", used to alter the distances between vertices without scaling the vertices or edges themselves, and better support for panning and zooming.

An animator is used to execute animation plans, which can cover layout changes, animations of changes to the graph's structure, or simple rollover highlights.

```
    public class BaseView extends JGraph implements Printable {

        // creation and access
        public BaseView(ViewGraph viewGraph);
        public ViewGraph getViewGraph();
        public BaseGraph getBase();
        public void setBase(BaseGraph base);

        // 'layout zoom' support
        public double getLayoutZoom();
        public void setLayoutZoom(double val);

        // zoom and pan
        public void setRelativeCenter(Point2D p, double zoomDiff);
        public void setCenter(Point2D desiredCenter);

        // vertex and edge labels and tooltips
        public String convertValueToString(Object cell);
        public String getToolTipText(MouseEvent event);

        // animation
        public Animator getAnimator();
        public void setAnimator(Animator animator);
        public AnimationPlan getCurrentPlan();
        public void setCurrentPlan(AnimationPlan currentPlan);

        // printing support
        public int print(Graphics graphics,
            PageFormat pageFormat, int pageIndex);

        // save and restore
        public void save(Element e);
        public void restore(Element e);
    }
```

B.2.15 ClusterView

Cluster views deal with a host of navigation issues encountered when representing clustered graphs, and include a "point of interest" – the vertex around which degree of interest will be calculated.

Besides the point of interest, other factors that affect automatic visibility recalculation include focus size, maximum concurrently displayed clusters, and "frozen" vertices, which will not be expanded or collapsed due to automatic visibility recalculation. Visibility recalculation can be toggled on or off using the *setClusterLock()* method.

Cluster navigation is saved in a stack, allowing navigational actions to be undone or redone. This lowers the negative effects of a “bad” navigational action. To prevent users from taking such actions, the default behaviour when hovering the mouse over a vertex displays an aura over vertices that will be affected by a change of focus to that vertex; this uses the results of calling *getPoIChangeEventFor()*.

```
public class ClusterView extends BaseView
        implements ClusteringChangeListener {

    // creation and basic access, including degree of interest parameters
    public ClusterView(ViewGraph viewGraph);
    public ClusterHierarchy geHierarchy();
    public void setBase(BaseGraph base);
    public Slice getSlice();
    public int getFocusSize();
    public void setFocusSize(int focusSize);
    public int getMaxClusters();
    public void setMaxClusters(int maxClusters);

    // frozen vertices do not participate in cluster expand/collapse
    public boolean isFrozen(Object v);
    public void setFrozen(Object v, boolean b);

    // when cluster-lock is active, no automatic DoI is performed
    public void setClusterLock(boolean clusterLock);
    public boolean isClusterLock();

    // visibility, point of interest and cluster navigation history
    public void makeVertexVisible(Object v);
    public ClusteringChangeEvent getPoIChangeEventFor(Object vertex);
    public Object getCurrentPoI();
    public void setCurrentPoI(Object anotherPoI, boolean recalculateDoI)
    public void nextNavAction();
    public void prevNavAction();
    public void clusteringChangePerformed(ClusteringChangeEvent evt);

    // save and restore
    public void save(Element e);
    public void restore(Element e);
}
```

B.2.16 Animator

Animators are in charge of responding to changes in the graph and animating them in a view; this requires incremental layout, which is performed by calling a layout manager with a specific algorithm.

Finished layouts are kept in a layout cache, allowing fast return to previous views, and consistent results after changes are undone. Manual layout changes are also accepted via the *resync()* methods.

```
public class Animator implements StructureChangeListener {

    // construction and access
    public Animator(BaseView view);
    public void setView(BaseView view);
    public LayoutManager getLayoutManager();
    public LayoutCache getLayoutCache();
    public void setLayoutCache(LayoutCache layoutCache);

    // initial layout (non-animated), incremental animated relayout
    public void start();
    public void doIncrementalLayout();

    // animate a smooth transition to a layout that reflects this change
    public void structureChangePerformed(StructureChangeEvent evt);

    // synchronize with real layout, or with changes contained in event
    public void resync();
    public void resyncFromEvent(GraphModelEvent evt);

    // save and restore
    public void restore(Element e);
    public void save(Element e);

    // first calculates incremental layout, then animates
    protected InterpolatedMovementStep incrementalLayoutStep(
            HashSet freeViews, int n, boolean useOldPositions) { ... }
}
```

B.2.17 ClusterAnimator

Cluster animators extend normal animators with support for *ClusteringChangedEvents*; since any clustering change event also involves addition and removal of vertices and edges, their structure change events are ignored, to be animated as clustering changes instead. Animation is performed using a host of dedicated steps.

```
public class ClusterAnimator extends Animator
    implements ClusteringChangeListener {

    // construction
    public ClusterAnimator(ClusterView view);
    public void setView(BaseView view);

    // structure changes triggered by clustering changes get ignored...
    public void structureChangePerformed(StructureChangeEvent evt);
    // ... since they will be received and processed here
    public void clusteringChangePerformed(ClusteringChangeEvent evt);

    // color hues used to highlight expanding and contracting vertices
    public float getExpandHue();
    public float getCollapseHue();

    // special animation steps used in clustering change animation
    private class SwitchAndUpdateLayoutStep
        extends TwoPhaseStep { ... }
    private class WaitAndHighlightStep
        extends AbstractStep { ... }
    private class ShiftFocusStep
        implements AnimationStep { ... }
    private class WaitAndCollapseStep
        extends InterpolatedMovementStep { ...}
}
```

B.2.18 AnimationPlan

Animation plans are associated with actual views (subclasses of *BaseView*). At most one plan can be in execution at any given moment, although it is possible to append an additional plan to another one already in execution. A priority mechanism is used to resolve ties: higher-priority plans replace lower-priority plans, while equal-priority plans are appended.

Plans execute individual steps, which can be performed either in sequence or in parallel. Steps that are expected to execute in parallel are said to be *merged*, and the duration of a merged step is that of its longest-running step.

```
public class AnimationPlan {

    // priories for real changes, layout changes, and simple highlights
    public static final int STRUCTURE_PRIORITY = 10;
    public static final int RELAYOUT_PRIORITY = 2;
    public static final int ROLLOVER_PRIORITY = 1;

    // creation and initalization
    public AnimationPlan(BaseView view, int priority);
    public void addStep(AnimationStep step);
    public void mergeStep(AnimationStep step);

    // query and access
    public boolean isRunning();
    public BaseView getView();
    public void setView(BaseView view);
    public int getPriority();
    public void setPriority(int priority);
    public String getDescription();
    public ArrayList<ArrayList<AnimationStep>> getMoves();

    // (try to) run; plans that must be stoped are terminated() first
    public void run();
    public void terminate();

    // when running, timer calls actionPerformed, advancing through moves
    private class PlanRunner implements Runnable, ActionListener {
        public boolean isRunning();
        public void finish();
        public void run();
        public void next();
        public void actionPerformed(ActionEvent evt);
    }
}
```

B.2.19 LayoutManager

A layout manager acts as a bridge between the actual layout algorithms and the view which they are in charge of laying out. Layout is performed on arrays of *Node*s, which are more efficient to access than the views themselves. Use of these arrays also decouples layout from the view itself, allowing layout algorithms to run on "virtual" positions. Recall that layout animation depends on interpolating between these final positions and the current vertex positions.

Layout managers can also save and restore a layout to and from a layout cache, enabling quick reuse of previous layouts when a match is found.

```
public class LayoutManager extends Observable implements Runnable {

    // initialize with a set of vertices, or load them from a view
    public LayoutManager(Node[] N);
    public LayoutManager(BaseView view);

    // run until finished or setCanContinue flag raised; then call commit
    public synchronized void run();
    public void setCanContinue(boolean b);
    public void commit(String s);

    // apply layout changes to view
    public void applyChanges(BaseView view);

    // set vertices to layout, time constraints and algorithms
    public void setNodes(Node[] nodes);
    public void setNodes(BaseView view, boolean useOldPositions);
    public int getMaxTime();
    public void setMaxTime(int maxTime);
    public LayoutAlgorithm getAlgorithm();
    public void setAlgorithm(LayoutAlgorithm algorithm);

    // use cache to restore or save vertex positions
    public double setNodesFromCache(LayoutCache cache,
        BaseView view, double minScore);
    public void addNodesToCache(LayoutCache cache, BaseView view);

    // direct access to vertex layout
    public Node[] getNodes();
}
```

B.2.20 LayoutAlgorithm

This abstract class is simple and flexible enough to implement any layout algorithm. The multiple *move()* methods apply the displacents of each node to the node itself. For certain algorithms, the total amount of displacement is important to determine termination; in others, displacement must be bounded to avoid excessive jitter.

```
public abstract class LayoutAlgorithm {

    // initialization, layout until layoutFinished or forced termination
    public void init(Node[] N);
    public abstract void layout();;
    public boolean layoutFinished();
    public void end();

    // bounded movement, free movement, dry run (return total displacement)
    public float move(double bound);
    public float move();
    public float simulateMove();
}
```

B.2.21 Node

This class is used to hold an alternative representation of the position and size of a visible vertex. Use of this alternative representation decouples visible layout from "layout-in-progress". Additionally, JGraph does not provide easy access to bounding boxes (arrays should be much more efficient), many algorithms are designed to operate on "point" vertices where width and height is ignored, and it is often necessary to store vertex displacements prior to actually performing them.

Multiple utility methods to synchronize layout parameters with the view or the view with the layout parameters have been implemented.

```
public class Node {

    // vertex data; includes connected component ID and interface vertex 'peer'
    public float x, y, x0, y0, w, h, dx, dy;
    public int component;
    public int[] edges;
    public float[] strengths;
    public boolean frozen;
    public Object peer;

    // load values from a string or rectangle
    public void sync(String s);
    public void sync(Rectangle2D r, int comp, boolean frozen,
        double layoutZoom);

    // build
    public Node(Object o, Rectangle2D bounds, int
        component, boolean frozen, double layoutZoom);
    public static Node[] loadNodes(Node[] N, Graph g,
            BaseView view,
            ConnectivityInspector ci,
            Node[] oldBounds);

    // query (but note public access to data)
    public Rectangle2D getBounds()
    public Rectangle2D getBounds(Rectangle2D r)

    // get bounds
    public static Point2D getCenterCoords(Map<Object,Rectangle2D> map,
        Collection vertices);
    public static Rectangle2D getBounds(Node[] N, double layoutZoom);

    // get/set vertex positions
    public static HashMap<Object,Rectangle2D> getPositions(
        Node[] N, BaseView view, boolean copyBounds) {
    public static Map getChangeMap(Node[] N, int x0, int y0,
        double layoutZoom)
    public static void setPositions(Map<Object,Rectangle2D> map,
        Node[] N, BaseView view);
}
```

B.2.22 LayoutCache

This is a simple cache that allows retrieval of frequently-used layouts. Each layout can be identified by a layout key. “Layout keys” are simple string representations of a graph.

Matching is sloppy; if no exact match can be found, all layouts are explored and the next-best match is used. Although the time required to find a sloppy match may be high, it will always be low compared to performing the actual layout.

```
public class LayoutCache {

    // create, change size, clear
    public LayoutCache(int maxSize);
    public void setSize(int maxSize);
    public void clear();

    // representation of a hit: a score, and a set of rectangles
    public static class CacheHit {
        public HashMap<Object,Rectangle2D> getData();
        public double getScore();
    }

    // lookup and add
    public CacheHit get(CacheKey key, double minScore);
    public void put(CacheKey key, HashMap<Object,Rectangle2D> layout);

    // save and restore
    public void save(Element e, BaseGraph g);
    public void restore(Element e, BaseGraph g);
}
```

B.3 File formats

XML has been used for all saved files, due to the widespread availability of parsers and manipulation tools, and its cross-platform nature. Redacted examples have been included to illustrate file formats; these examples are easier to read than the actual XML Schemas used for file format specification.

B.3.1 Clover save file

This format is used to save Clover-specific information on graph views, filters, clustering hierarchies and layout cache. Cluster hierarchies and/or filters can be shared between different views. The following example save-file contains a cluster hierarchy, a filter, and a view definition with a single layout cache entry.

```
<?xml version="1.0" encoding="UTF-8"?>
<clover version="1.0c" requiresVersion="1.0b"
        date="Tue Apr 01 13:56:23 CEST 2007">
```

```
    <shared>
        <hierarchy id="1">
            <cluster>
                <vertex id="T_Traffic" />
                <cluster>
                    <!-- ... rest of the cluster hierarchy -->
                </cluster>
            </cluster>
            <engine engineClass="eps.woted.graph.WtRuleClusterer" />
        </hierarchy>
        <filter id="2" filterClass="eps.woted.graph.WtProfileFilter">
            <!-- ...filter constraints -->
        </filter>
        <!-- ... more hierarchies or filters -->
    </shared>
    <view zoom="1.0"
          topCorner="0.0,0.0"
          poiVertex="T_EduVial" focusSize="1" maxClusters="16"
          isClusterLock="false"
          frozenVertices="" visibleSlice= [...]
          hierarchyId="1">

        <layoutCache maxSize="30">
            <entry key= [...] >
                <box id="T_EduVial"
                     x="227.69" y="235.15"
                     w="120.0" h="32.0" />
                <!-- ... one box per visible vertex -->
            </entry>
            <!-- ... the rest of the layout cache entries -->
        </layoutCache>
        <animatorProps
            maxInterpolationTime="2000" initialLayoutArea="300"
            initialLayoutTime="4000" cacheSloppynessLimit="0.9"
            incrementalRefinementPasses="10" />
    </view>
    <!-- ... more views-->
</clover>
```

B.3.2 Wotan course description

This format is used to store the description of a Wotan course structure. No actual course contents are included; the contents themselves are only referenced by versions within fragments. The course description file includes a series of *tasks*, which may include *rules* that reference other tasks, and references to *fragments*; a series of fragments, which include one or more *versions*; and a series of *features*, used to prime the user models of new users with course-specific characteristics.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<twCourse maintask="EduVial" name="traffic" xmlns="http://tangow.ii.uam.es/tangowml">
    <description lang="en">Traffic</description>
    <description lang="es">Curso de trafico</description>

    <task fragments="ea1 ea2 ea3 ea4 ea5" id="S_Ag_Exer" name="S_Ag_Exer" type="P">
        <description lang="en">Exercises about signals by traffic agents</description>
        <description lang="es">Ejercicios de señales de agentes</description>
        <condition>map.task.S_Ag_Exer.grade &gt;= .5</condition>
    </task>
    <task id="S_Prio" name="S_Prio" type="T">
        <description lang="en">Sign priority</description>
        <description lang="es">Prioridad de las señales</description>
        <rule name="RP" seq="A" subtasks="S_Prio_Theo S_Prio_Exam">
            <condition>map.course.age == "old"</condition>
        </rule>
        <rule name="RPPP" seq="A" subtasks="S_Prio_Theo S_Prio_Exam">
            <condition>map.course.experience == "novice"</condition>
        </rule>
        <condition>true</condition>
    </task>
    <task id="S_Circ" name="S_Circ" type="T">
        <description lang="en">Circumstantial signs</description>
        <description lang="es">Señalizacion circunstancial</description>
        <rule name="R1" seq="A" subtasks="S_Circ_Theo S_Circ_Exer">
            <condition>(map.task.S_Vertical.grade &gt;= .5)</condition>
        </rule>
        <condition>true</condition>
    </task>
    <!-- ... many other task descriptions here -->

    <fragment id="ea5">
        <version url="ea5/spanish-old-novice.txt">
            <condition>0.5 * (map.course.language == "spanish")
                  + 0.25 * (map.course.age == "old")
                  + 0.125 * (map.course.experience == "novice")</condition>
        </version>
        <version url="ea5/english-old-novice.txt">
            <condition>0.5 * (map.course.language == "english")
                  + 0.25 * (map.course.age == "old")
                  + 0.125 * (map.course.experience == "novice")</condition>
        </version>
    </fragment>
    <!-- ... many other fragment descriptions here -->

    <feature name="language" type="enum">
        <description lang="en">Course language</description>
        <description lang="es">Idioma del curso</description>
        <description lang="de">Kurssprache</description>
        <description lang="it">Lingua di corso</description>
        <range values="english,spanish,german,italian">
            <description lang="en">English,Spanish,German,Italian</description>
            <description lang="es">Inglés,Español,Alemán,Italiano</description>
            <description lang="de">Englisch,Spanisch,Deutsch,Italienish</description>
            <description lang="it">Inglese,Spagnolo,Tedesco,Italiano</description>
        </range>
    </feature>
    <!-- ... many other course features here -->
</twCourse>
```

B.3.3 Wotan user model

This format is used to store a user model for a Wotan adaptive course, assuming the use of the default Tangow-based adaptation engine. User models are divided into two different files, the course independent or *global* UM, and course overlays. Overlays can be used to restore a

Course-independent user model

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<user xmlns="http://tangow.ii.uam.es/userml">
    <fullName>
        <name>Fred</name>
        <surname>Flintstone</surname>
    </fullName>
    <pAddress>
        <address>C/Antonio Sancha, 66.</address>
        <postalId>28042</postalId>
        <region>Madrid</region>
        <country>España</country>
    </pAddress>
    <eAddress>
        <telephone>915554308</telephone>
        <telephone>655579659</telephone>
        <email>fred.flintstone@gmail.com</email>
        <fax/>
        <web>http://www.fredssite.com</web>
    </eAddress>
    <age>25</age>
    <language>en</language>
    <courses>
        <course id="Traffic" location="file://./users/fred"/>
        <!-- ... other individual course descriptions here -->
    </courses>
    <userAtt key="name" value="Fred"/>
    <userAtt key="age" value="25"/>
    <userAtt key="language" value="en"/>
</user>
```

Course Overlay

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<course course="Traffic"
 created="2005-06-09T15:41:04.731+02:00" updated="2005-06-09T16:03:34.846+02:00"
 xmlns="http://tangow.ii.uam.es/courseml">
   <profile>
       <feature name="age" value="old"/>
       <feature name="language" value="spanish"/>
       <feature name="experience" value="novice"/>
   </profile>
   <task name="EduVial">
       <rule name="R00">
           <attribute name="active" value="true"/>
       </rule>
       <attribute name="rule" value="R00"/>
       <attribute name="current" value="false"/>
       <attribute name="available" value="true"/>
       <attribute name="grade" value="0"/>
       <attribute name="complete" value="0.98"/>
       <attribute name="interest" value="0.1"/>
       <attribute name="visits" value="1"/>
   </task>
   <!-- ... many other task descriptions here -->
   <task name="S_Vert_M_Exer">
       <attribute name="current" value="false"/>
       <attribute name="available" value="true"/>
       <attribute name="grade" value="0.6"/>
       <attribute name="complete" value="1.0"/>
       <attribute name="interest" value="0.1"/>
       <attribute name="visits" value="1"/>
       <attribute name="parent" value="S_Vert_Mean"/>
   </task>
   <fragment name="emergencias">
       <version name="emergencias/spanish-old-novice.html"/>
       <version name="emergencias/english-old-novice.html"/>
   </fragment>
   <fragment name="evs1">
       <version name="evs1/spanish-old-novice.txt"/>
       <version name="evs1/english-old-novice.txt"/>
       <attribute name="grade" value="1.0"/>
   </fragment>
   <!-- ... many other fragments here -->
   <fragment name="vrapida">
       <version name="vrapida/spanish-old-novice.html"/>
       <version name="vrapida/english-old-novice.html"/>
   </fragment>
   <log>
       <entry message="First go at course"
              timestamp="2005-06-09T15:41:04.732+02:00" type="SESSION-START"/>
       <entry message="Leaving session"
              timestamp="2005-06-09T16:03:34.936+02:00" type="SESSION-END"/>
       <!-- ... many other log entries here -->
   </log>
</course>
```

Bibliography

[1] JGraph and JGraph Layout Pro user manual.
http://www.jgraph.com/pub/jgraphmanual.pdf. Last visited, Dec. 2006. 70, 75, 91, 167

[2] Ariadne Foundation website.
http://www.ariadne-eu.org/. Last visited, Jan. 2007. 46

[3] Google Personalized Search.
http://www.google.com/psearch. Last visited, Jan. 2007. 46

[4] IEEE Standard for Learning Object Metadata.
http://ltsc.ieee.org/wg12/files/LOM_1484_12_1_v1_Final_Draft.pdf. Last visited, Jan. 2007. 46

[5] Instructional Management System (IMS) website.
http://www.imsglobal.org/. Last visited, Jan. 2007. 46

[6] JGraphPad user manual.
http://www.jgraph.com/pub/jgraphpadmanual.pdf. Last visited, Apr. 2007. 74

[7] MERLOT website.
http://www.merlot.org/merlot/index.htm. Last visited, Jan. 2007. 46

[8] Prefuse website.
http://prefuse.org. Last visited, Apr. 2007. 76

[9] RSS 2.0 Specification.
http://www.rssboard.org/rss-specification. Last visited, Jan. 2007. 46

[10] Sharable Content Object Reference Model (SCORM) specification, v3.
http://www.adlnet.gov/scorm/20043ED/Documentation.cfm. Last visited, Jan. 2007. 47

[11] TouchGraph, commercial website.
http://touchgraph.com. Last visited, Apr. 2007. 72, 73

[12] TouchGraph, open source website. http://sourceforge.net/projects/touchgraph. Last visited, Apr. 2007. 72

[13] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The lorel query language for semistructured data. *International Journal on Digital Libraries*, 1:68–88, Apr. 1997. 21

[14] E. Alfonseca, D. Perez, and P. Rodrıguez. Welkin: automatic generation of adaptive hypermedia sites with NLP techniques. *Lecture Notes in Computer Science*, 3140:617–618, 2004. 45

[15] L. A. N. Amaral, A. Scala, M. Barthélémy, and H. E. Stanley. Classes of small-world networks. *Proceedings of the National Academy of Sciences*, 97(21):11149–11152, 2000. 15, 16, 17

[16] L. Aroyo, D. Dicheva, and A. Cristea. Ontological Support for Web Courseware Authoring. In *Proc. Int. Conf. On Intelligent Tutoring Systems (ITS'02)*, pages 270–280. Springer, 2002. 48, 72

[17] D. . Auber, Y. Chiricota, F. Jourdan, and G. Melancon. Multiscale visualization of small world networks. In *Proceedings of the IEEE Symposium on Information Visualization, InfoVis'03*, pages 75–81, 2003. 15, 17, 20, 22, 73

[18] D. Auber, M. Delest, and Y. Chiricota. Strahler based graph clustering using convolution. In *Proceedings of the Eighth International Conference on Information Visualisation, InfoVis'04*, pages 44–51, 2004. 73

[19] D. Auber and F. Jourdan. Interactive Refinement of Multi-scale Network Clusterings. In *Proceedings of the Ninth International Conference on Information Visualisation, InfoVis'05*, pages 703–709, 2005. 65, 73

[20] P. Auillans and O. Baudon. Graph clustering for very large topic maps. In P. Gennusa, editor, *Proceedings of XML-Europe 2001*, May 2001 2001. 17

[21] C. Bachmaier, F. J. Brandenburg, M. Forster, P. Holleis, and M. Raitner. Gravisto: Graph visualization toolkit. In J. Pach, editor, *Proc. Graph Drawing, GD 2004*, volume 3383 of *Lecture Notes in Computer Science*, pages 502–503. Springer, 2005. 71, 73

[22] R. A. Baeza-Yates and B. A. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press / Addison-Wesley, 1999. 25, 47

[23] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286:509–512, 1999. 15, 16, 17

[24] V. Batagelj and A. Mrvar. Pajek-Program for Large Network Analysis. *Connections*, 21(2):47–57, 1998. 73

[25] G. D. Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs.* Prentice-Hall, Upper Saddle River, New Jersey 07458, U.S.A, July 1998. 27

[26] B. B. Bederson, J. Grosjean, and J. Meyer. Toolkit design for interactive structured graphics. *IEEE Trans. Software Eng*, 30(8):535–546, 2004. 64, 74, 77

[27] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic Web. *Scientific American*, 284(5):28–37, 2001. 48

[28] S. Boccaletti, V. Latora, Y. Moreno, M. Chavez, and D. Hwang. Complex networks: Structure and dynamics. *Physics Reports*, 424(4-5):175–308, 2006. 13, 16, 17

[29] K. Boerner, C. Chen, and K. W. Boyack. Visualizing knowledge domains. *Annual Review of Information Science and Technology*, 37(1):179–255, 2003. 33

[30] R. A. Botafogo. Cluster analysis for hypertext systems. In *Proceedings of the 16th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 116–125. ACM Press, 1993. 19

[31] J. Bravo and A. Ortigosa. Validating the evaluation of adaptive systems by user profile simulation. *Proceedings of Workshop Held at the Fourth International Conference on Adaptive Hypermedia and Adaptive Web-Based Systems (AH2006)*, pages 479–483, June 2006. 152

[32] T. Bray, M. Pilgrim, S. Ruby, and et al. The atom syndication format. *IETF Request for Comments*, 4287:1–43, Dec. 2005. 46

[33] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the Web. *Computer Networks*, 33(1-6):309–320, 2000. 15, 16

[34] P. Brusilovsky. Methods and techniques of adaptive hypermedia. *User Modeling and User-Adapted Interaction*, 6(2-3):87–129, 1996. 42

[35] P. Brusilovsky. Adaptive Hypermedia. In *User Modeling and User-Adapted Interaction*, volume 11, pages 87–110. Kluwer Academic Publishers, The Netherlands, 2001. 42, 44, 45

[36] P. Brusilovsky. Developing adaptive educational hypermedia systems: From design models to authoring tools. In *Authoring Tools for Advanced Technology Learning Environment. Dordrecht: Kluwer Academic Publishers*, pages 377–409, 2003. 51, 53

[37] P. Brusilovsky, E. W. Schwarz, and G. Weber. ELM-ART: An intelligent tutoring system on world wide web. In *ITS '96: Proceedings of the Third International Conference on Intelligent Tutoring Systems*, pages 261–269, London, UK, 1996. Springer-Verlag. 45

[38] A. L. Buchsbaum and J. R. Westbrook. Maintaining hierarchical graph views. In *Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*, pages 566–575. Society for Industrial and Applied Mathematics, 2000. 22, 66, 67, 68

[39] S. K. Card, J. D. Mackinlay, and B. Shneiderman, editors. *Readings in Information Visualization — Using Vision to Think*. Morgan Kaufmann, 1999. 25, 62

[40] R. M. Carro. *Un mecanismo basado en tareas y reglas para la creación de sistemas adaptativos: aplicación a la educación a través de Internet*. PhD thesis, Escuela Politécnica Superior de la Universidad Autónoma de Madrid, Sept. 2001. 109

[41] R. M. Carro, E. Pulido, and P. Rodriguez. TANGOW: a Model for Internet Based Learning. *International Journal on Continuing Education and Life-Long Learning*, 11(1–2):25–34, 2001. 45, 54, 109

[42] C. Chen. Editorial. *Information Visualization*, 1:1–4, 2002. 25

[43] L. Chittaro. Information visualization and its application to medicine. *Artificial Intelligence in Medicine*, 22(2):81–88, 2001. 26

[44] M. Consens and A. Mendelzon. Hy+: a hygraph-based query and visualization system. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 511–516, New York, NY, USA, 1993. ACM Press. 21

[45] P. de Bra, A. Aerts, D. Smits, and N. Stash. AHA! Version 2.0, More Adaptation Flexibility for Authors. In *Proceedings of the AACE ELearn'2002 conference*, pages 240–246, Oct. 2002. 54

[46] P. de Bra and L. Calvi. Aha! an open adaptive hypermedia architecture. *New Review of Hypermedia and Multimedia*, 4:115–140, 1998. 43, 45, 50

[47] C. Dichev, D. Dicheva, and L. Aroyo. Using Topic Maps for Web-based Education. *Advanced Technology for Learning*, 1(1):1–7, 2004. 72, 73

[48] D. Dicheva and C. Dichev. Authoring educational topic maps: Can we make it easier? In *ICALT*, pages 216–218. IEEE Computer Society, 2005. 48, 72

[49] H. A. do Nascimento and P. Eades. User Hints for Directed Graph Drawing. *Lecture Notes in Computer Science*, 2265:205, Jan. 2002. 71

[50] Dwyer and Eckersley. Wilmascope – an interactive 3D graph visualisation system. In *GDRAWING: Conference on Graph Drawing (GD)*, 2001. 32, 150

[51] P. Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984. 28

[52] P. Eades and M. L. Huang. Navigating clustered graphs using force-directed methods. *JGAA: Special Issue on Selected Papers from 1998 Symp. Graph Drawing*, 4(3):157–181, 2000. 22, 37, 71

[53] J. Eklund and P. Brusilovsky. InterBook: An adaptive tutoring system. *UniServe Science News*, 12:8–13, 1999. 45

[54] J. Ellson, E. Gansner, L. Koutsofios, S. North, and G. Woodhull. Graphviz-open source graph drawing tools. *Graph Drawing*, 2265:483–485, 2001. 83

[55] R. Felder and L. Silverman. Learning and Teaching Styles in Engineering Education. *Engineering Education*, 78(7):674–681, 1988. 49

[56] M. Freire. Visualization of hypermedia course structures. Master's thesis, Escuela Politécnica Superior, Universidad Autónoma de Madrid, Sept. 2003. 130, 132

[57] M. Freire. WOTAN documentation.
http://tangow.ii.uam.es/wotan. Last visited, jan 2007. 50, 109

[58] M. Freire, M. Cebrian, and E. del Rosal. Ac: An integrated source code plagiarism detection environment. Pre-print manuscript, available at
http://www.citebase.org/abstract?id=oai:arXiv.org:cs/0703136, May 2007. 136, 138

[59] M. Freire and P. Rodríguez. Comparing graphs and trees for adaptive hypermedia authoring. In *3rd International Workshop on Authoring of Adaptive and Adaptable Educational Hypermedia (A3EH) at the12th International Conference on Artificial Intelligence in Education (AIED)*, pages 4–12. AIED'05, 18 July 2005. 121

[60] M. Freire and P. Rodríguez. Preserving the mental map in interactive graph interfaces. In *Proceedings of Advanced Visual Interfaces (AVI)*, pages 270–273, New York, NY, USA, May 2006. ACM Press. 37

[61] M. Freire and P. Rodríguez. Graphs versus trees in adaptive hypermedia authoring. Submitted to the Journal of Digital Information, April 2007. 121

[62] C. Friedrich and P. Eades. Graph drawing in motion. *Journal of Graph Algorithms and Applications*, 6(3):353–370, 2002. 38, 76, 77, 100

[63] C. Friedrich and P. Eades. Navigating clustered graphs using force-directed methods. In *journal of graph algorithms and applications*, volume 6, pages 353–370, 2002. 37

[64] T. Fruchterman and E. Reingold. Graph Drawing by Force-directed Placement. *Software- Practice and Experience*, 21(11):1129–1164, 1991. 29, 94

[65] G. W. Furnas. Generalized fisheye views. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 16–23. ACM Press, 1986. 35, 88

[66] G. W. Furnas. Effective view navigation. In *CHI '97: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 367–374, New York, NY, USA, 1997. ACM Press. 23, 87

[67] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, Massachusetts, 1994. 62, 75, 89, 93, 171, 176

[68] Garey and Johnson. Crossing number is NP-complete. *SIJADM: SIAM Journal on Algebraic and Discrete Methods*, 4:312–316, 1983. 28

[69] S. Garlatti and S. Iksal. A Semantic Web Approach for Adaptive Hypermedia. In *Workshop on Adaptive Hypermedia and Adaptive Web-Based Systems. AH2003. Twelfth International World Wide Web Conference*, 2003. 45

[70] N. Gershon, S. G. Eick, and S. Card. Design: Information visualization. *interactions*, 5(2):9–15, 1998. 25

[71] R. Grimaldi. *Discrete and combinatorial mathematics.* Addison-Wesley Reading, Mass, 1994. 10

[72] A. Gutierrez, P. Pucheral, H. Steffen, and J. Thévenin. Database Graph Views: A Practical Model to Manage Persistent Graphs. In *20th International Conference on Very Large Data Bases*, pages 391–402. 21

[73] F. V. Harmelen, J. Broekstra, C. Fluit, H. ter Horst, A. Kampman, J. van der Meer, and M. Sabou. Ontology-based information visualisation. In *5th International Conference on Information Visualization (IV '01)*, pages 546–554, Washington - Brussels - Tokyo, July 2001. IEEE. 21

[74] E. Hartuv and R. Shamir. A clustering algorithm based on graph connectivity. *Inf. Process. Lett.*, 76(4-6):175–181, 2000. 18, 19

[75] P. A. Haya and G. Montoro. *A spoken interface based on the contextual modelling of smart homes*, pages 147–154. Springer Verlag, January 2006. 140

[76] P. A. Haya, G. Montoro, and X. Alamán. A prototype of a context-based architecture for intelligent home environments. In R. Meersman and Z. Tari, editors, *CoopIS/DOA/ODBASE*, volume 3290 of *Lecture Notes in Computer Science*, pages 477–491. Springer, 2004. 140

[77] J. Heer and S. K. Card. DOITrees revisited: scalable, space-constrained visualization of hierarchical data. In M. F. Costabile, editor, *AVI*, pages 421–424. ACM Press, 2004. 76

[78] N. Henze and W. Nejdl. Adaptivity in the KBS Hyperbook System. In *Proceedings of the 2nd Workshop on Adaptive Systems and User Modeling on the WWW*, 1999. 45, 50

[79] I. Herman, G. Melançon, and M. S. Marshall. Graph Visualization and Navigation in Information Visualization: A Survey. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):24–43, 2000. 27, 30, 32

[80] P. Holleis and F.-J. Brandenburg. QUOGGLES: Query on graphs - A graphical largely extensible system. In J. Pach, editor, *Graph Drawing, 12th International Symposium, GD ' 04*, volume 3383 of *Lecture Notes in Computer Science*, pages 465–470. Springer, Sept. 2004. 21, 73

[81] K. Höök, J. Karlgren, A. Wærn, N. Dahlbäck, C. Jansson, K. Karlgren, and B. Lemaire. A glass box approach to adaptive hypermedia. *User Modeling and User-Adapted Interaction*, 6(2):157–184, 1996. 42

[82] X. Huang, P. Eades, and W. Lai. A framework of filtering, clustering and dynamic layout graphs for visualization. In V. Estivill-Castro, editor, *Proceedings of the Twenty-Eighth Australasian Computer Science Conference (ACSC2005)*, volume 38 of *CRPIT*, pages 87–96. Australian Computer Society, January 2005. 22, 62, 65, 71

[83] X. Huang and W. Lai. Force-transfer: a new approach to removing overlapping nodes in graph layout. In *CRIPTS '03: Proceedings of the twenty-sixth Australasian computer science conference on Conference in research and practice in information technology*, pages 349–358, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc. 95, 152

[84] S. Iksal and S. Garlatti. Revisiting and Versioning in Virtual Special Reports. *Third Workshop on Adaptive Hypertext and Hypermedia, 12th ACM Conference on Hypertext and Hypermedia, Arhus, Denmark, August*, 2001. 134

[85] T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Inf. Process. Lett.*, 31(1):7–15, 1989. 28

[86] J. Kay. Learner control. *User Modeling and User-Adapted Interaction*, 11(1-2):111–127, 2001. 42

[87] N. Kiesel, A. Schurr, and B. Westfechtel. GRAS, a graph-oriented (software) engineering database system. *Information Systems*, 20(1):21–52, 1995. 21

[88] R. Kincaid and H. Lam. Line graph explorer: scalable display of line graphs using focus+context. In A. Celentano, editor, *AVI*, pages 404–411. ACM Press, 2006. 138

[89] J. Lamping, R. Rao, and P. Pirolli. A focus+context technique based on hyperbolic geometry for visualizing large hierarchies. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 401–408. ACM Press/Addison-Wesley Publishing Co., 1995. 35, 36

[90] B. Lee, C. Plaisant, C. S. Parr, J.-D. Fekete, and N. Henry. Task taxonomy for graph visualization. In *BELIV '06: Proceedings of the 2006 AVI workshop on BEyond time and errors*, pages 1–5, New York, NY, USA, 2006. ACM Press. 31

[91] Y. K. Leung and M. D. Apperley. A review and taxonomy of distortion-oriented presentation techniques. *ACM Trans. Comput.-Hum. Interact.*, 1(2):126–160, 1994. 33

[92] W. Li, Y. Hara, R. Ito, Y. Kimura, K. Shimazu, Y. Saito, Q. Vu, E. Chang, D. Agrawal, K. Hirata, et al. PowerBookmarks: a system for personalizable Web information organization, sharing, and management. In *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, pages 565–567. ACM Press New York, NY, USA, 1999. 46

[93] J. A. Macías and P. Castells. *Interactive Design of Adaptive Courses*, pages 235–242. Kluwer Academic Publishers, 2001. 53

[94] J. Marks, editor. *The Marey Graph Animation Tool Demo*, volume 1984 of *Lecture Notes in Computer Science*. Springer, 2000. 73

[95] M. S. Marshall, I. Herman, and G. Melançon. An object-oriented design for graph visualization. *Software, Practice and Experience*, 31(8):739–756, 2001. 62

[96] S. Milgram. The small world problem. *Psychology Today*, 1:61, 1967. 13

[97] K. Misue, P. Eades, W. Lai, and K. Sugiyama. Layout adjustment and the mental map. *Journal of Visisual Languages and Computing*, 6(2):183–210, 1995. 35

[98] B. Naveh and contributors. JGraphT project website. `http://jgrapht.org/`. Last visited, Jan. 2007. 64, 73, 81, 165, 170

[99] A. Newell. *Unified theories of cognition*. Harvard University Press, Cambridge, MA, USA, 1990. 25

[100] N. Noy, M. Sintek, S. Decker, M. Crubezy, R. Fergerson, and M. Musen. Creating Semantic Web contents with Protege-2000. *Intelligent Systems, IEEE [see also IEEE Intelligent Systems and Their Applications]*, 16(2):60–71, 2001. 48, 74

[101] J. O'Madadhain, D. Fisher, S. White, and Y. Boey. The JUNG (Java Universal Network/Graph) Framework. Technical report, Technical Report UCI-ICS 03-17. School of Information and Computer Science, UC Irvine. Irvine, California, 2003. 64, 69, 73, 171

[102] R. Oppermann, R. Rashev, and Kinshuk. Adaptability and adaptivity in learning systems. In A. Behrooz, editor, *Knowledge Transfer (volume II)*, pages 173–179, 1997. 41

[103] F. Pfeiffer. Implementation eines editors für compound graphen. Diplomarbeit, University of Passau, 2005. 71, 72

[104] F. Pfeiffer and M. Pröpster. VisnaCom. `http://www.infosun.fim.uni-passau.de/VisnaCom/`. Last visited, Apr. 2007. 71, 73, 77

[105] S. Pook, E. Lecolinet, G. Vaysseix, and E. Barillot. Context and interaction in zoomable user interfaces. In *AVI '00: Proceedings of the working conference on Advanced visual interfaces*, pages 227–231, New York, NY, USA, 2000. ACM Press. 34

[106] M. Pröpster. Visuelle navigation in compound graphen. Diplomarbeit, University of Passau, 2005. 71, 72

[107] M. Raitner. HGV: A library for hierarchies, graphs, and views. In S. G. Kobourov and M. T. Goodrich, editors, *Graph Drawing*, volume 2528 of *Lecture Notes in Computer Science*, pages 236–243. Springer, 2002. 71, 73

[108] M. Raitner. Maintaining hierarchical graph views for dynamic graphs. Technical report, University of Passau, Feb. 13 2004. 71

[109] M. Raitner. *Efficient Visual Navigation of Hierarchically Structured Graphs.* PhD thesis, Fakultät für Mathematik und Informatik, Universität Passau, Feb 2006. 22, 65, 66, 68, 71, 85, 151

[110] K. Rosen and J. Michaels. *Handbook of discrete and combinatorial mathematics.* CRC Press Boca Raton, Fla, 2000. 10

[111] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations.* World Scientific, 1997. 20

[112] M. Sarkar, S. S. Snibbe, O. J. Tversky, and S. P. Reiss. Stretching the rubber sheet: A metaphor for viewing large layouts on small screens. In *Proceedings of the 6th Annual Symposium on User Interface Software and Technology*, pages 81–92, New York, NY, USA, Nov. 1993. ACM Press. 35

[113] D. Schaffer, Z. Zuo, S. Greenberg, L. Bartram, J. Dill, S. Dubs, and M. Roseman. Navigating hierarchically clustered networks through fisheye and full-zoom methods. *ACM Trans. Comput.-Hum. Interact.*, 3(2):162–188, 1996. 35

[114] B. Schneiderman, G. Fischer, M. Czerwinski, B. Myers, and M. Resnick, editors. *Creativity Support Tools Workshop.* National Science Foundation, sep 2005. 38

[115] B. Shneiderman. The eyes have it: a task by data type taxonomy for invormation visualization. In *Proceedings of the IEEE Workshop on Visual Language*, pages 336–343, 1996. 26

[116] Song, Havlin, and Makse. Self-similarity of complex networks. *NATURE: Nature*, 433:392–395, 2005. 17, 18, 20

[117] T. Stafford and M. Webb. *Mind Hacks: Tips & Tools for Using Your Brain.* O'Reilly Media, Nov. 2004. 25, 76

[118] N. Stash, A. Cristea, and P. De Bra. Explicit Intelligence in Adaptive Hypermedia: Generic Adaptation Languages for Learning Preferences and Styles. In *International Workshop on Combining Intelligent and Adaptive Hypermedia Methods/Techniques in Web-Based Education Systems, HT*, volume 5, pages 6–9, 2005. 50

[119] M. Storey, M. Musen, J. Silva, C. Best, N. Ernst, R. Fergerson, and N. Noy. Jambalaya: Interactive visualization to enhance ontology authoring and knowledge acquisition in Protege. *Workshop on Interactive Tools for Knowledge Capture, K-CAP-2001*, 2001. 48

[120] M.-A. D. Storey, F. D. Fracchia, and H. A. Müller. Customizing a Fisheye View Algorithm to Preserve the Mental Map. *Journal of Visual Languages & Computing*, 10(3):245–267, 1999. 34

[121] M.-A. D. Storey, N. F. Noy, M. A. Musen, C. Best, R. W. Fergerson, and N. Ernst. Jambalaya: an interactive environment for exploring ontologies. In *IUI*, pages 239–239, 2002. 74, 134, 150

[122] Storey, Margaret-Anne, C. Best, J. Michaud, D. Rayside, M. Litoiu, and M. Musen. SHriMP views: an interactive environment for information visualization and navigation. In *Proceedings of ACM CHI 2002 Conference on Human Factors in Computing Systems*, volume 2 of *Demonstrations*, pages 520–521, 2002. 64, 73, 74, 106

[123] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions On Systems, Man, And Cybernetics*, SMC-11(2):109–125, Feb. 1981. 28

[124] G. Teege. Reuse of teaching materials in Targeteam. In *International Workshop on Interactive Computer aided Learning ICL 2002*, 2002. 53, 130

[125] G. Teege. Targeteam website.
http://www.targeteam.net/. Last visited, May 2007. 130, 131

[126] The Object Management Group. Ontology Definition Metamodel,.
http://www.omg.org/docs/ad/05-08-01.pdf Last visited, june 2005. 48

[127] E. R. Tufte. *The Visual Display of Quantitative Information.* Graphics Press, Cheshire, CT, USA, 1983. 25

[128] E. R. Tufte. *Envisioning Information.* Graphics Press, 1990. 25

[129] E. R. Tufte. *Visual Explanations.* Graphics Press, 1997. 25

[130] D. Tunkelang. JIGGLE: Java interactive graph layout environment. In S. Whitesides, editor, *Graph Drawing*, volume 1547 of *Lecture Notes in Computer Science*, pages 412–422. Springer, 1998. 73

[131] S. Van Dongen. Graph clustering by flow simulation. Master's thesis, Center for Mathematics and Computer Science (CWI), 2000. 18, 19

[132] F. van Ham and J. J. van Wijk. Interactive visualization of small world graphs. In *Proceedings of the IEEE Symposium on Information Visualization (INFOVIS'04)*, pages 199–206, Washington, DC, USA, 2004. IEEE Computer Society. 33, 35, 36, 65, 70, 71

[133] J. Vassileva. DCG+GTE: Dynamic Courseware Generation with teaching expertise. *Instructional Science*, 26(3–4):317–332, July 1998. 45

[134] C. Walshaw. A multilevel algorithm for force-directed graph drawing. In J. Marks, editor, *Proc. 8th Int. Symp. Graph Drawing, GD*, volume 1984 of *Lecture Notes in Computer Science, LNCS*, pages 171–182. Springer-Verlag, 20–23 Sept. 2000. 30, 152

[135] C. Ware. *Information Visualization: Perception for Design.* Morgan Kaufmann Publishers, San Francisco, 2000. 25

[136] D. J. Watts. Networks, dynamics, and the small-world phenomenon. *The American Journal of Sociology*, 105:493, Sept. 1999. 14

[137] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393:440–442, 4 June 1998. 2, 13, 14, 15, 16, 159

[138] S. Weibel, J. Kunze, C. Lagoze, and M. Wolf. Dublin Core Metadata for Resource Discovery. *IETF Request for Comments*, 2413:1–8, sep 1998. 47

[139] G. Whale. Identification of Program Similarity in Large Populations. *The Computer Journal*, 33(2):140, 1990. 138

[140] R. Wiese, M. Eiglsperger, and M. Kaufmann. yfiles: Visualization and automatic layout of graphs. In *Proceedings of the 9th International Symposium on Graph Drawing (GD'01)*, pages 453–454. Springer, 2001. 70, 73

[141] K. Zhang, R. Statman, and D. Sasha. On the editing distance between unordered labeled trees. *Inf. Proc. Lett.*, 42:133–139, May 1992. 66

Index