Software Design For a Fuzzy Cognitive Map Modeling Tool

Stephen T. Mohr
66.698 Master’s Project
Fall 1997
Rensselaer Polytechnic Institute

8 September 1997

Table of Contents

Overview of Fuzzy Cognitive Maps

Fuzzy cognitive maps (FCM) are soft computing tools which combine elements of fuzzy logic and neural networks. Strictly speaking, an FCM is a digraph in which qualitative concepts are the nodes, and causal influences are the edges. Concept nodes possess a numeric state which denotes a qualitative measure of the concepts presence in the conceptual domain. A high numeric value indicates the concept is strongly present. A negative or zero value – implementations vary – indicates the concept is not currently active or relevant to the conceptual domain. An FCM works in discrete steps. When a strong correlation exists between a concept's state and another concept's state in the preceding step, we say the former concept positively influences the latter concept and we draw a positively weighted edge from the causing concept to the influenced concept. When a strong negative correlation exists, there is a negative causal influence, and we draw an edge with a negative weight. Two conceptual nodes without a direct link are independent.

FCMs are preferable to quantitative tools in domains involving complex webs of causal relationships, particularly feedback, and where hard quantitative measures of influences are not available. They are easy to construct, allow users to rapidly compare their mental model of a system with the real world, and, because of their fuzzy logic elements, extremely forgiving of uncertain information. FCMs are excellent informal tools for knowledge workers as well as a simple and clear way to visually represent causal relationships.

Consider the FCM depicted in Figure 1. This simple FCM simulates the relationships between capital investment in a company, manufacturing productivity, profit, and other measures of a company's performance. Positive casual influences are shown as black edges, while negative casual influences are in red. Thus, when quality is "up", support costs will go down. When investment is occurring, research spending goes up and productivity increases. Because this model is not quantitative, it is easily modified to reflect changing beliefs and domain understanding. It enables a user to quickly develop a model and test investment strategies before putting substantial effort into quantitative modeling. Because it is graphical, it quickly exposes the author's assumptions to his readers and permits them to study what happens to the author's arguments when those assumptions are changed.

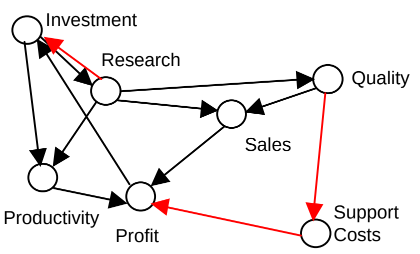

Figure 1: Corporate Investment FCM

The state of conceptual node A at time step n is computed by taking the sum of the inputs, i.e., the state values at step $n-1$ of nodes with edges coming into A multiplied by the corresponding edge weights. Because this is a qualitative model, we can maintain stability by normalizing the state value following summation. Many threshold functions are available for this normalization. For the purposes of this project, we make the following assumptions. Edge values are on the range [-1..1]. Three threshold functions will be implemented and offered to the user. The first, the bivalent threshold function, is

$$S_i(x_i) = 0, x_i \le 0$$
$$S_i(x_i) = 1, x_i > 0$$

where x_i is the summation of the inputs prior to normalization. The bivalent threshold function is the simplest and by far the most commonly used threshold function in FCM models. The next threshold function, trivalent, extends the range of concept state values to include negative activation according to

$$S_i(x_i) = -1, x_i \le -0.5$$
$$S_i(x_i) = 0, -0.5 < x_i < 0.5$$
$$S_i(x_i) = 1, x_i \ge 0.5$$

Our final threshold function, the logistic signal function, is a continuous function and provides true fuzzy conceptual node states. The function is

$$S_i(x_i) = \frac{1}{1 + e^{-cx_i}}$$

The constant, c, is critical in determining the degree of fuzzification of the function. At large values, the logistic signal function approaches discrete threshold functions. We have chosen c = 5 as a trade-off which favors the center of the range. Plots of this threshold function for various values of the constant are shown in Figure 2.

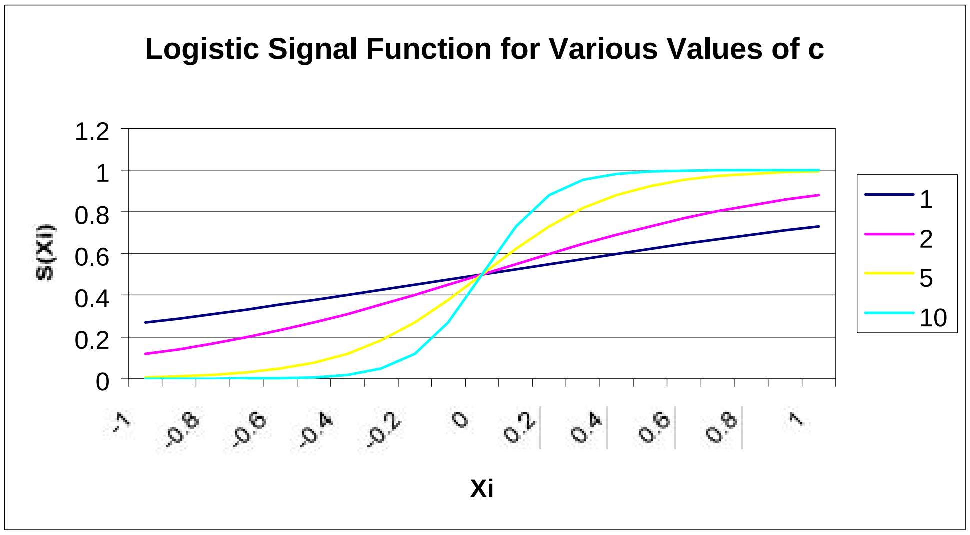

Figure 2: Comparisons of logistic signal functions

FCMs using either of the discrete threshold functions will either reach an equilibrium state or converge to a limit cycle. The threshold functions force fuzzy state vectors to non-fuzzy values. FCMs using the logistic signal threshold function may become nonlinear under some conditions of feedback. In this case, chaotic attractors may exist. Since the state vector of the map at time `n` is completely determined by the state vector at time `n-1`, equilibrium states may be easily detected during FCM simulation by comparing two successive state vectors. If they are identical, the map has reached equilibrium. The problem of predicting limit cycles and chaotic attractors is left as a research problem for the last stage of this project.

Goals and Requirements

This design is concerned with the production of a set of software revolving around, but not limited to, an application that permits the construction, maintenance, and operation of FCMs. The following items are the specific requirements of this software:

- Tool must be capable of building, saving, and loading FCM models
- Tool must be capable of calculating FCM states in continuous and single state modes
- FCM states should be visually represented to the user in a form which clearly distinguishes positive and negative casual influences and active and inactive concept values
- Simple user interface
- Capable of deployment in standalone and HTML applet versions from a common codebase
- Classes should be developed which enable reuse in the development of general digraph related software
- The tool should permit inference of FCMs based on observed concepts and successive state vectors

When the project was proposed, the desire was to create a tool for investigating FCMs. As the plan was refined, it became apparent that the software should support dual deployment: a standalone application for creating and using FCMs in the richest environment, and as an embedded applet for publishing completed models and allowing casual users to experience FCMs. Coincidentally, experience in consulting practice led the author to realize there is a need to create and edit digraphs in a number of problem domains. This is the motivation for setting a high standard for software reuse.

Consideration of the uses of FCMs suggests a need for machine learning and analytical capabilities. Machine learning would help a user detect changes in an established model by using inference against observed performance whenever a model's predictions deviate from observed behavior. By comparing the old model with the newly learned model, a user should detect shifts in casual relationships. An algorithm presented in the literature will be implemented to support this (see also *Machine Learning*, below). Similarly, it would be convenient if the software could analyze a FCM to determine whether it converges to an equilibrium state, a limit cycle, or whether it is, in fact, chaotic. In the latter cases, it would be useful to analytically determine the limit cycle or strange attractors. Unfortunately, we know FCMs are not amenable to a general Lyapunov analysis [1]. Additional research will be conducted to see if a limit cycle analysis is possible. If this proves possible, it may be incorporated into the final release of the software.

Enabling Technologies

The Java programming language is selected for development of this software in order to increase our familiarity with this language and to permit deployment as either a standalone application or an embedded applet in an HTML page. The availability of development tools and WWW browser capabilities drive the choice of development kit (JDK) version. JDK 1.1 is newly released. The JDK itself includes a compiler, but no integrated development environment supporting such productivity features such as class browsers and graphical dialog box layout. WWW browsers are only beginning to be released with support for the new JDK. Consequently, any software developed with JDK 1.1, specifically JavaBeans, will be unable to run as an HTML applet within commonly available WWW browsers. Consequently, we have chosen JDK 1.0 for the development kit version and Microsoft Visual J++ 1.1 as the development environment.

Although the primary platform for this application is Microsoft Windows NT 4.0 and Windows 95, the software will not make use of any platform specific extensions. Any JVM compliant with the JDK 1.0 specification should be able to run the finished software. Nevertheless, we will attempt in the later phases of the project to demonstrate integration of the FCM simulator with other applications using the Component Object Model (COM) as the enabling object system. This takes advantage of the fact that Microsoft's JVM exposes Java classes as ActiveX components to the operating system.

User Interface

The user interface owes much to a Java application for demonstrating Dijkstra's shortest path algorithm written by Carla Laffra at Pace University [2]. This design makes some key alterations, however, to support FCMs.

The heart of the modeling tool is the class FCMApplication, a subclass of the Java class Applet. It consists of two visual components as shown in Figure 3. The display area for the graph permits users to create and modify FCMs with a variety of mouse commands (see Table 1). Concept nodes are created with default values of one for the current and initial values, zero for the initial and current values of the external input, and "N" for the name. These may be changed using the concept configuration dialog box depicted in Figure 4.

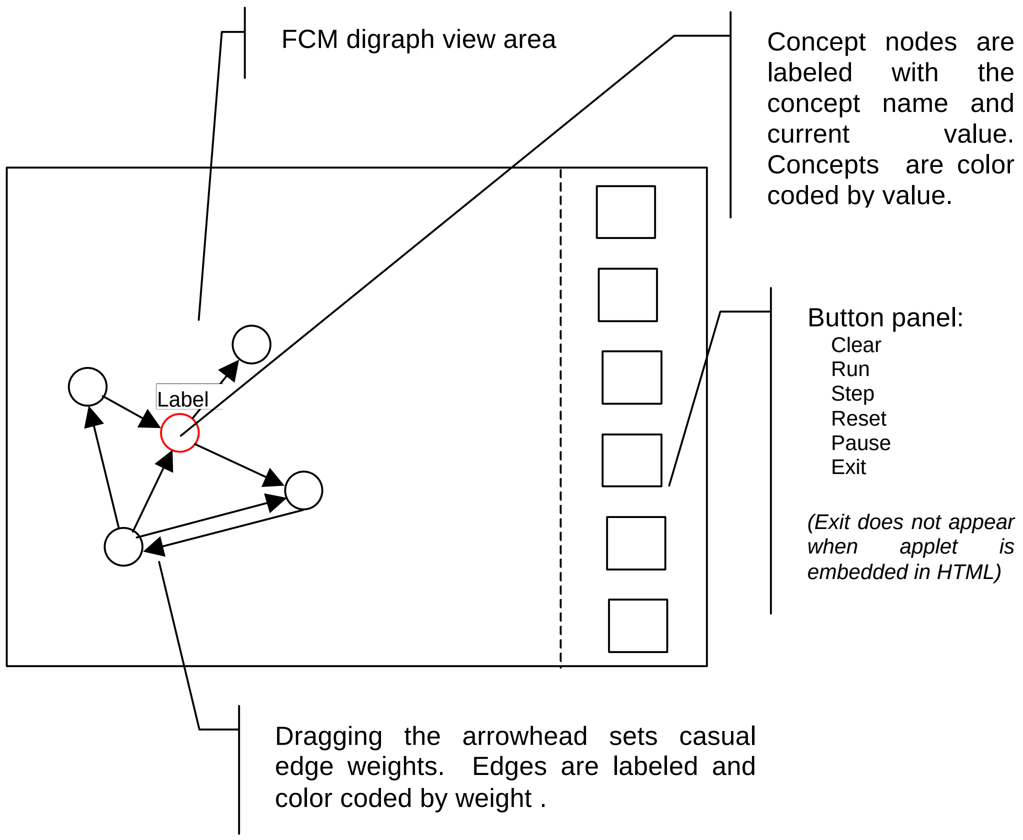
FCM digraph view area
Concept nodes are labeled with the concept name and current value. Concepts are color coded by value.
Label
Button panel:
Clear
Run
Step
Reset
Pause
Exit
(Exit does not appear when applet is embedded in HTML)
Dragging the arrowhead sets casual edge weights. Edges are labeled and color coded by weight .

Figure 3: User interface thumbnail diagram for FCM applet

Target	Gesture	Effect
None	Click	New concept node
Concept node[i]	Drag	New casual node
Concept node	Ctrl + click	Delete concept node
Concept node	Dbl + click	Invoke configuration dialog box
Concept node	Shift + drag	Move concept node
Casual edge[ii]	Shift + click	Delete casual edge
Casual edge	Drag	Change edge weight

Table 1: User interface specification for digraph view

[i] Drag gesture must terminate at another concept node
[ii] Edges are always manipulated by their arrowheads. This holds for weight changes and deletion.

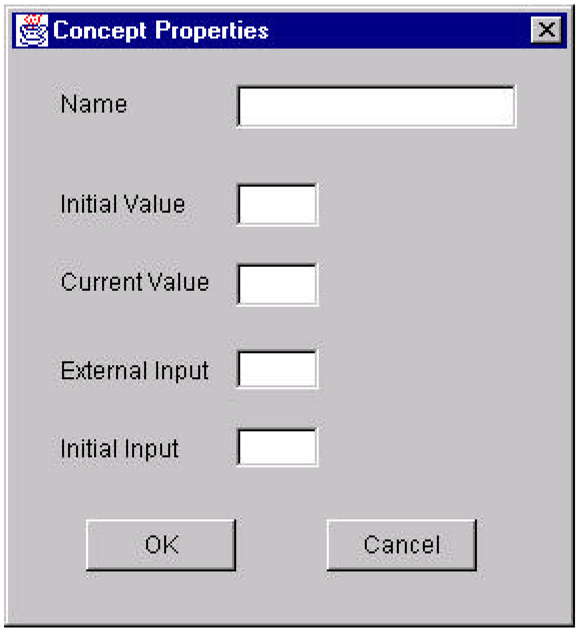

Figure 4: Concept configuration dialog box

The interface specified so far is sufficient to create and modify an FCM. More is needed to operate the state simulation. The button panel located to the right ("East" in Java parlance) provides the minimal set of controls for the simulator. The interface specification for the buttons is given in Table 2.

Button	Effect
Clear	Empty graph
Run	Executes the state simulator in continuous mode until user intervenes or an equilibrium state is reached. Simulator pauses for two seconds between states.
Step	Calculates and displays the next state of the FCM.
Reset	Resets existing FCM to its initial state
Pause	Suspends an FCM run without losing state
Exit[iii]	Terminates the application

Table 2: User interface specification for button panel

A richer user interface is required for a standalone application. For this, we embed the applet in a frame window as depicted in Figure 5. This adds the platform specific capabilities of the title bar, e.g., system control boxes, as well as a menu bar exposing the full features of the FCM modeling tool. The menu specification is given in Table 3.

[iii] The Exit button does not appear when the applet is embedded in an HTML page.

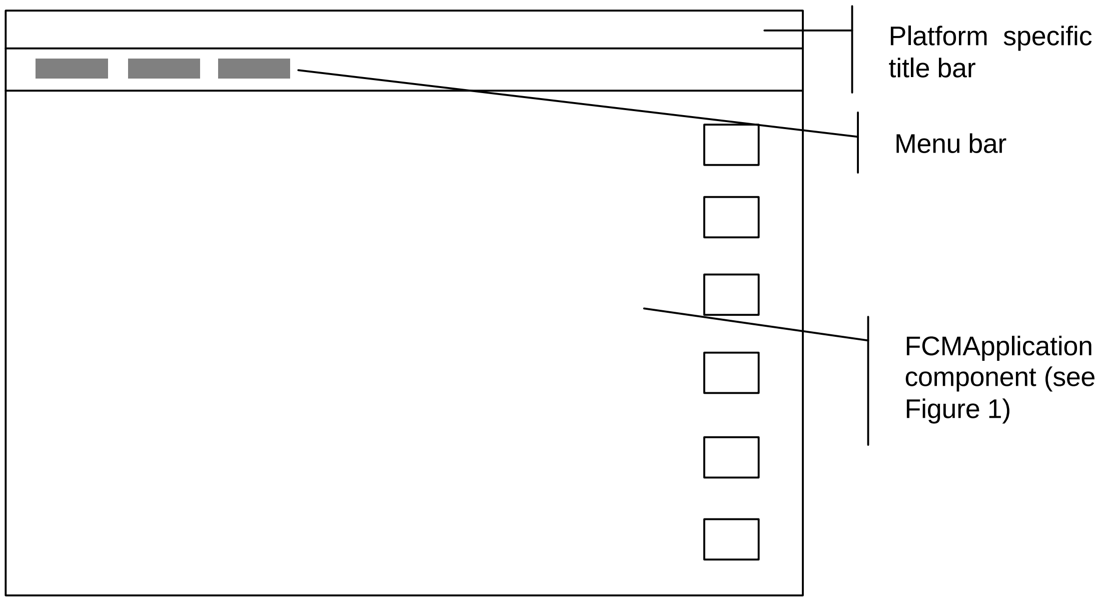
Platform specific title bar
Menu bar
FCMApplication component (see Figure 1)

Figure 5: Standalone application thumbnail diagram

Menu Item	Effect
File	
Open…	Invokes system standard file dialog box to open a previously saved FCM.
Learn…	Invokes a system standard file dialog box to open a data file of concept names and state vectors.
Save	Saves the current FCM under the existing name. If no name has been specified, Save As… is invoked.
Save As…	Invokes a system standard file dialog box to save the persistent properties of the current FCM model.
Exit	Terminates the application
Threshold[iv]	
Bivalent	Selects the bivalent threshold function.
Trivalent	Selects the trivalent threshold function
Logistic	Selects the logistic threshold function
Help	
About…	Invokes a dialog box with information descriptive of the application

Table 3: Application menu specification

[iv] Items in this menu are checked to denote the threshold function in use.

Logical Structure

Java is an object-oriented language, necessitating a class-based design to the application. In addition, the application is divided into components[v] as suggested in *User Interface*, above. Although not as strongly component-based as JavaBean or ActiveX applications, the requirement to provide HTML and standalone versions resulted in the creation of a single component for this project. The application and its constituent component were designed using the Object Modeling Technique (OMT), a formal method for object-oriented design.

Reuse Granularity

The author's experience has been that source code reuse occurs only when it is a design requirement, and then only when careful consideration is given to the wider reuse domain. If the designer considers the immediate application first and reuse second, classes tend to be so focussed on the immediate project that the code is never used elsewhere. Consequently, we need to consider what, if anything, in an FCM can be reused.

Laffra's code came to the author's attention through another program that made use of it, an implementation of Bayesian inference networks. Moreover, two other projects in the author's consulting practice have elements that either require or would benefit from directed graphs. Clearly, then, we should develop a class library permitting the construction of digraphs[vi]. In particular, we need classes implementing weighted edges, nodes, and a class implementing digraphs using those classes. The classes need to participate in the construction of digraphs by handling the addition and deletion of nodes and edges. In particular, nodes must cooperate in communicating their status to connected nodes to permit edge updates when a "downstream" node is deleted. All classes must be capable of persistent storage. Most problems involving digraphs involve the calculation of some algorithm on the digraph. Many such algorithms involve multiple steps. Consequently, we design a digraph representation that involves a numeric edge weight, a numeric state for the node, and a multiple step algorithm. Implementation of a digraph using our classes involves specializing the graph class to implement the specific algorithm.

Model – View Implementation

A common technique for facilitating reuse is the model-view-controller paradigm. Since the controller is typically driven by the operating system, this is usually known as the model-view paradigm in actual practice. Popular class frameworks, including Borland's Object Windows Library (OWL) and Microsoft's Foundation Classes (MFC) make use of this paradigm. In this approach, a single object, the model, encapsulates the state and logic of the domain object. One or more visual representations, or views, are associated

[v] We use the term *component* in the sense of component software – software developed from multiple "black box" components instantiated at runtime.

[vi] Of course, this generalization could be extended to undirected graphs, as well, but a balance was struck between code complexity resulting from generalization and the possibility of reuse. We simply did not encounter any problem involving undirected graphs.

with a single model object. This potentially permits multiple views, e.g., chart and table, of the same underlying data.

A key problem in the model-view approach is constructing the association between a given model and its views. MFC uses an entirely separate class to perform this association. However, this is not warranted by the scope of this project. We are, after all, building a single application. Reuse is a desired outcome. Construction of yet another class framework is a poor use of programming effort[vii]. Early design efforts included complicated structures and processing for forwarding user events originating in a view to the appropriate model. However, it quickly became apparent that this project would only have a single view. Moreover, it seemed that general digraph construction would only require a single view. Runtime use of digraphs would either use this view (as is the case with this application), or no view at all, as in the case of an application which uses the algorithm calculations in support of some other object. As an example, an interactive graph applet written in Java [3], is strictly similar to our design in its visual representation, except that nodes are square instead of round. This could easily be accomplished via inheritance.

We quickly realized that a slight variation on the model – view approach would suffice to limit the complexity of our classes while reaping some of the benefits of model – view architecture. At each level of complexity – edge, node, and graph, we start with a nonvisual class implementing the model for that level. Next, we derive a class implementing the view of that level. Since we delegate the rendering of the graph to each level, changes in the way graphs are drawn may be accomplished by overloading the drawing routines within each class.

Graph Representation

Graphs in general, and digraphs in particular, are commonly represented in one of two ways[4]. The *adjacency matrix* approach for a graph with n nodes uses an $n \times n$ matrix in which an entry in the (i, j) element denotes an edge between nodes i and j. In a digraph, this denotes a weighted edge from node i to node j with an edge weight equivalent to the value of the matrix element. This is an effective representation for many algorithms. For FCMs, this is computationally efficient as the state of the map is defined as

$$C_{n+1} = S(C_n E)$$

where C is a state vector, E is the adjacency matrix, and S is the threshold function[5]. Given this representation, each step of the FCM calculation becomes a matter of matrix multiplication, followed by application of the threshold function.

An adjacency matrix is not resource efficient for sparse matrices, however, nor is it well suited to the problem of interactively building and editing a graph. While many FCMs are strongly interconnected, construction of an FCM using an adjacency matrix would

[vii] Another framework is also likely unwelcome, given the recent furor surrounding Netscape's Internet Foundation Classes and Microsoft's Application Foundation Classes for Java. Less truly is more in today's Java market.

involve frequent reallocation of memory. An *adjacency structure* approach is a better approach to this problem. In our case, each node contains a dynamically sized array[viii] of edges. Strictly speaking, this is sufficient to represent the graph. In the interests of runtime efficiency, however, each node also maintains an array of nodes which have edges coming into the node in question. This greatly simplifies the problem of navigating backwards through the graph, e.g., when nodes connected to a deleted node need to be notified of the deletion.

Our graph class, and the FCM related specialization derived from it, uses the adjacency structure representation. When the user wishes to compute FCM states, an adjacency maxtrix representation is created and encapsulated within another class. A state vector is passed to the new class at each step to permit external inputs to be varied at any given point in the process. Any node or edge deletions or additions causes the matrix representation to be destroyed, forcing recreation the next time a state is calculated. Since a user rarely changes casual relationships in a map during calculations, this has little performance impact.

Persistence

A common practice in the implementation of persistence in class libraries is to use a network database schema. This is the practice in OWL and MFC. Objects are initially streamed with some kind of header denoting the class of the object. When instantiated, the default constructor is used to create an object instance. The framework then calls some required method which initializes the instance from streamed data. Subsequent references to an object are written and read as an index denoting a previously written or read object instance. The network database schema comes into play when objects contain other objects. Examining the raw stream for such a case reveals the persistent form of the contained object embedded within the streamed form of the containing object, as one might expect. The navigational metaphor of the network database is a good model of the containment relationship between objects.

Early design efforts anticipated the creation of a custom approach along these lines. A review of current literature, however, has turned up a pair of classes, PersistentInputStream and PersistentOutputStream, included in a recent Java language book [6]. Although these classes possess some drawbacks[ix], the copyright terms of the book permit their use in noncommercial applications, and their use will accelerate development, permitting us to focus on the study of FCMs. Several changes should be made, both to improve efficiency and to diminish proprietary source code. These changes are:

[viii] Specifically, an instance of the Java class Vector, which preallocates memory in configurable chunks and handles resource reclamation.

[ix] For pedagogical purposes, all output is written as text rather than binary format. Moreover, early experience with these classes reveal some inconsistencies in how data is read and pushed back. The structure tracking objects previously read and written is an instance of the Vector class, which limits its efficiency when searching large numbers of objects. Finally, there is a problem with circular references, e.g., a node containing an edge with a reference back to the containing node as the source of the edge.

- Change from text format to binary for non-string data
- Eliminate known deficiencies in the existing source code
- Manage object indices through a hash table rather than an array
- Offer performance optimizations

The last point stems from our observation that some persistent objects are referenced only once and will never, therefore, be found in the structure tracking previously streamed objects. Offering programmers the option of writing such objects directly to and from the stream, bypassing the overhead of tracking, will result in some performance benefit. In the present project, edges are referenced only by the source node and can therefore benefit from such an approach. An order of magnitude analysis of the impact of this improvement is as follows. In a fully connected FCM comprised of N concept nodes, there will be N(N-1) edges. Each edge in our design has two node references: one to the source, and one to the target. This adds 2N(N-1) references requiring tracking. An additional N-1 node references are added because each node tracks the nodes providing inbound edges. In sum, we have $O(N+2N(N-1)+N-1) = O(2N^2-1)$ performance for the execution of persistence requiring tracking, and $O(N^2-N)$ performance for the execution of persistence directly to the stream. For maps with large numbers of concept nodes, this optimization offers the potential of significant but not compelling performance gains.

Some of the listed changes will be made immediately, while others may have to wait completion of the core FCM functionality. Priority is in descending order as listed.

Classes

The Object modeling Technique (OMT) formal method was used to complete an object-oriented design satisfying the requirements set forth in *Goals and Requirements*. The resulting class diagram appears in Figure 6. Classes distributed with the JDK, excepting Vector, are not shown for clarity. The class FCMFrame is a thin class providing the enhanced user interface when the program executes as a standalone application. The FCMGraphView is the core component common to both standalone and embedded applet modes. The principal classes, and those which embody the reuse strategy described above, are Node, Edge, Graph, FCMGraph, and their associated classes NodeView, EdgeView, GraphView, and FCMGraphView. We will begin our discussion of the class design with these classes.

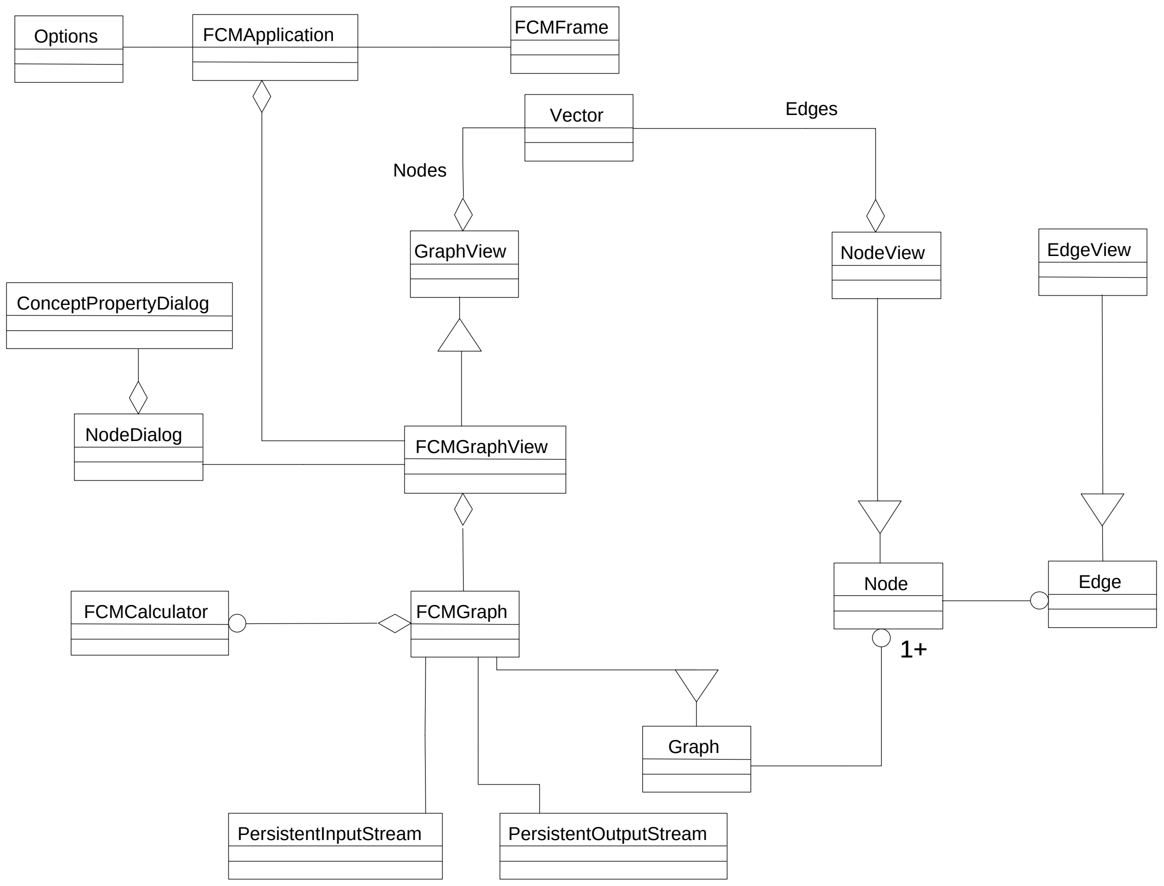

Figure 6: Master class diagram for the FCM Modeler application

Nodes and Edges

The class Edge is very simple. It maintains properties for its source and target nodes as well as the edge weight. Some simple accessor methods are provided to control public access to these properties. EdgeView is chiefly concerned with maintaining the visual state of the edge. An edge color property is added, as are properties for the screen coordinates and some values useful in improving rendering performance. EdgeView is able to calculate a new edge weight value from a given arrowhead location as well as the inverse operation. A draw method accepts a Graphics instance and draws the edge on it when invoked. The FCM domain has no further requirements for edges, so there is no reason to further specialize these two classes.

Nodes are more complicated. They contain a Vector instance whose elements are the edges originating with the node. This is sufficient, but a Vector object containing references to the nodes which provide inputs to the host node is added to simplify navigation backwards through the graph, e.g., to notify influencing nodes of the impending deletion of the host node. Initial and current state values are properties. Initial and current external influence properties are added to implement a fixed external influence on the node, i.e., when "clamping" a concept to represent a fixed policy originating outside the map's domain. The class NodeView performs visual services analogous to those offered by EdgeView to Edge. Again, with the possible exception of the external influence properties, FCMs do not require any unique methods or properties, so Node and NodeView are not further specialized.

Graphs

The Graph class contains a Vector instance of Nodes and coordinates common digraph operations such as adding and deleting edges and nodes. It also handles operations relating to persistent storage. One of our assumptions is that a digraph has some algorithm associated with it. Since we cannot specify this algorithm and maintain generality, Graph is an abstract class. We specify the operational framework of executing a multi-step algorithm in a multithreaded environment (to permit the algorithm to be halted during a run), but the methods of this framework are virtual. Consequently, any programmer reusing our classes must derive a model class from Graph which implements the specific algorithm in question. This is the purpose of FCMGraph. FCMGraph inherits the implementations of Graph, but adds the ability to calculate FCM state vectors. For reasons discussed in *Graph Representation*, FCM Graph drives the overall calculation process but delegates the individual state calculations to a contained instance of FCMCalculator.

GraphView continues our practice of separating visual representations from domain models. Typical GUI actions such as dragging and clicking are handled here. When the graph must be rendered on the screen, it iterates through the contained NodeView instances, directing them to draw themselves on the shared Graphics instance. The NodeViews, in turn, direct their contained EdgeView instances to render. Consequently, GraphView contains little rendering code of its own, serving instead as a coordinator for its components. FCMs require overloaded functionality in two areas. FCM calculation is sensitive to changes in the composition of the graph, the threshold function in effect, and changes in external inputs[x], so interface handlers which result in such changes must call the WarnEngine method of FCMGraph. For this reason, we derive FCMGraphView from GraphView to implement this. In addition, FCMGraphView must notify the frame window of the threshold function in effect when it loads an FCM from persistent storage or when clearing an FCM so that the appropriate menu check marks may be updated.

Following the sequence of events in the following message trace diagrams illuminates the containment relationships inherent in graphs. These diagrams show the sequence of messages needed to implement some common graph building operations.

[x] We commonly wish to change which concepts are clamped and when to model changes in external policy. In the example FCM given in the overview, we might wish to clamp capital investment and research for several state iterations to study a program of start-up investment, then unclamp the concepts to see how the FCM performs in normal operation.

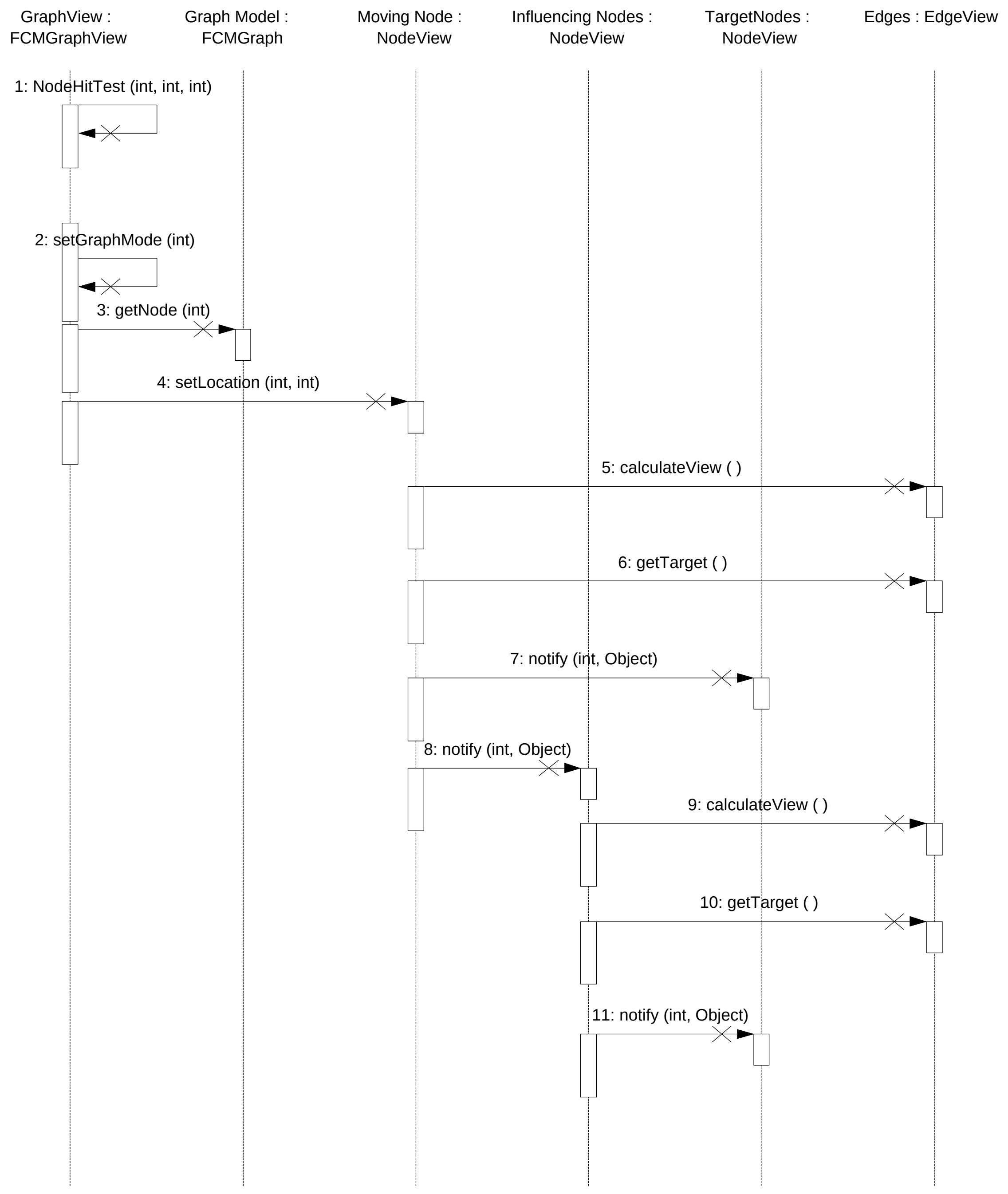

Figure 7: Message trace diagram for moving a node

Figure 7 depicts the application's response to a user mouse drag gesture. The instance of FCMGraphView performs hit testing to see if the initial mouse down gesture occurred over a node. If it is, it sends itself a message to set its global state to indicate a drag operation in progress. This allows the view to preserve state over a related set of discrete events, i.e., mouse down, mouse move, mouse up. A reference to the appropriate node object is obtained and used to instruct the object (known to be an instance of NodeView), to set its location. This is all that is of interest at the level of the graph. Nodes, however, must notify their contained edges as well as their influencing nodes to ensure all edges originating or terminating at the affected node are updated. The node just moved handles this by first sending calculateView messages to its contained edges, allowing them to recalculate their view data. Next, the node notifies its peer influencing nodes, which respond by sending calculateView messages to their contained edges. Thus, a simple operation at the graph view level results in a cascade of messages at lower levels.

Figure 8 demonstrates the ripple effect caused by deleting an edge. Once again, hit testing is performed. This time, node hit testing fails, so edge hit testing occurs. When it

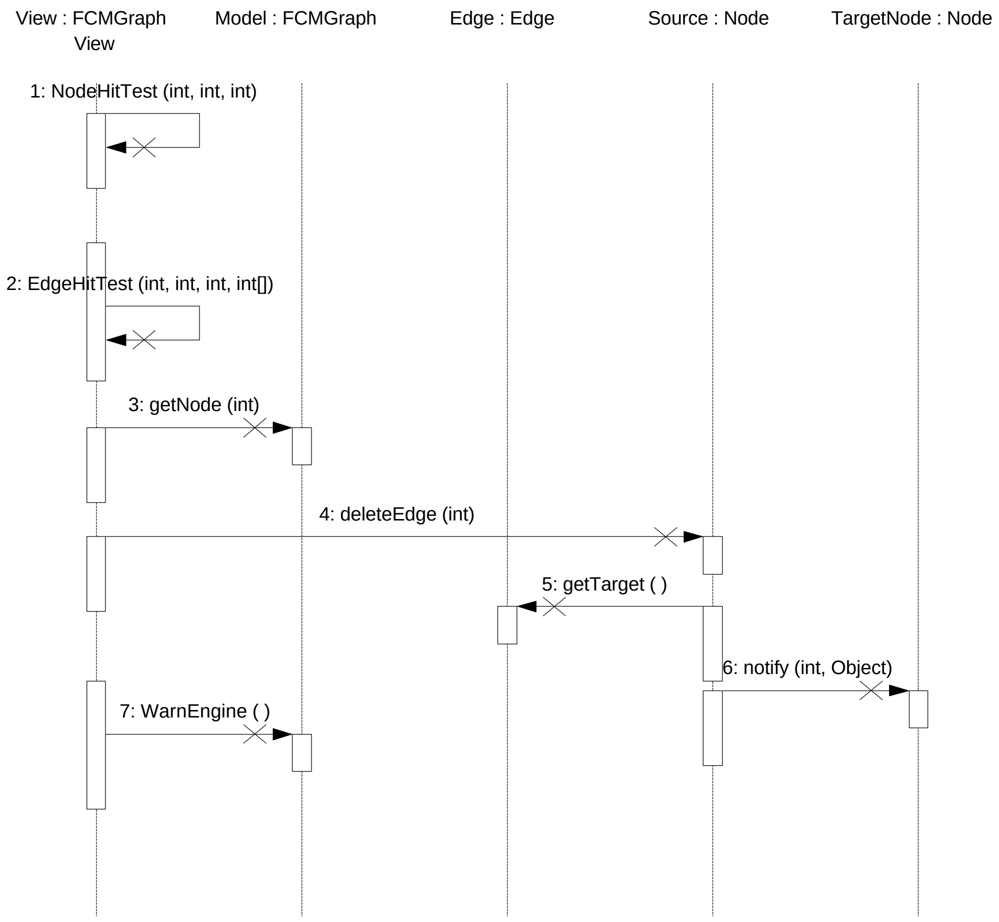

Figure 8: Message trace diagram for deleting an edge

succeeds, the view uses the indices thus obtained[xi] to obtain a reference to the node. It instructs this node to delete the edge. Before removing the edge from its collection, the node obtains a reference to the edge's target node and notifies that node of the impending

[xi] As hit test indices directly relate to dynamic structures, the validity of such indices is not generally guaranteed beyond two immediately sequential method calls. However, since related mouse events preclude operations which change the contents of these structures, indices are guaranteed for the duration of these related gestures.

deletion. When this activity is complete, the graph view is able to warn its model that an activity has taken place which affects state calculations.

The process of deleting a node, illustrated in Figure 9, is somewhat simpler. The hit test – obtain node reference sequence allows the view to send the selected node a KillNode message. Since nodes implicitly understand that they are connected to other nodes, the node notifies its influences of the change before the node is deleted. Edges are automatically garbage collected by the Java virtual machine when the node is deleted. Finally, the view sends its model a warning regarding this change.

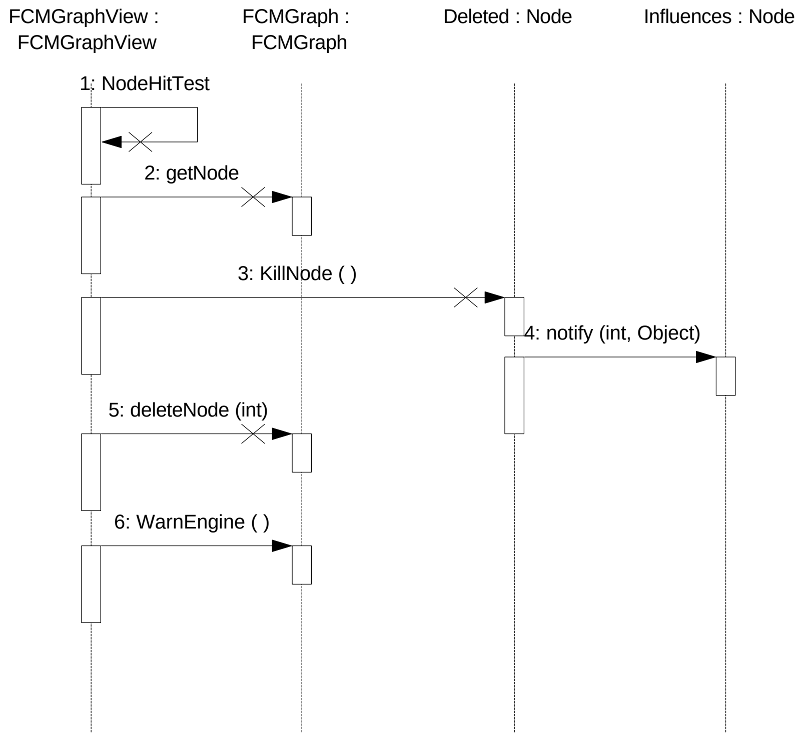

Figure 9: Message trace diagram for deleting a node

A final message trace diagram of interest is that regarding the calculation of FCM states. As noted above, we use to instance of FCMCalculator contained in the FCMGraph instance to calculate a single step at a time. This permits both single step operation and pausing the algorithm to change external influence settings. As illustrated in Figure 10, the graph constructs a vector of external inputs by querying its nodes. This vector is submitted together with a state vector (similarly obtained) to the FCM calculator object. Following calculation of the next step, assuming an equilibrium state has not been reached, the view updates its nodes by sending them messages with the concept state values returned by the calculator. The graph notifies its view that significant changes have been made. In response, the view repaints itself, then suspends its thread briefly to permit the user to observe the change. If additional state calculations are needed, the view object loops through these steps.

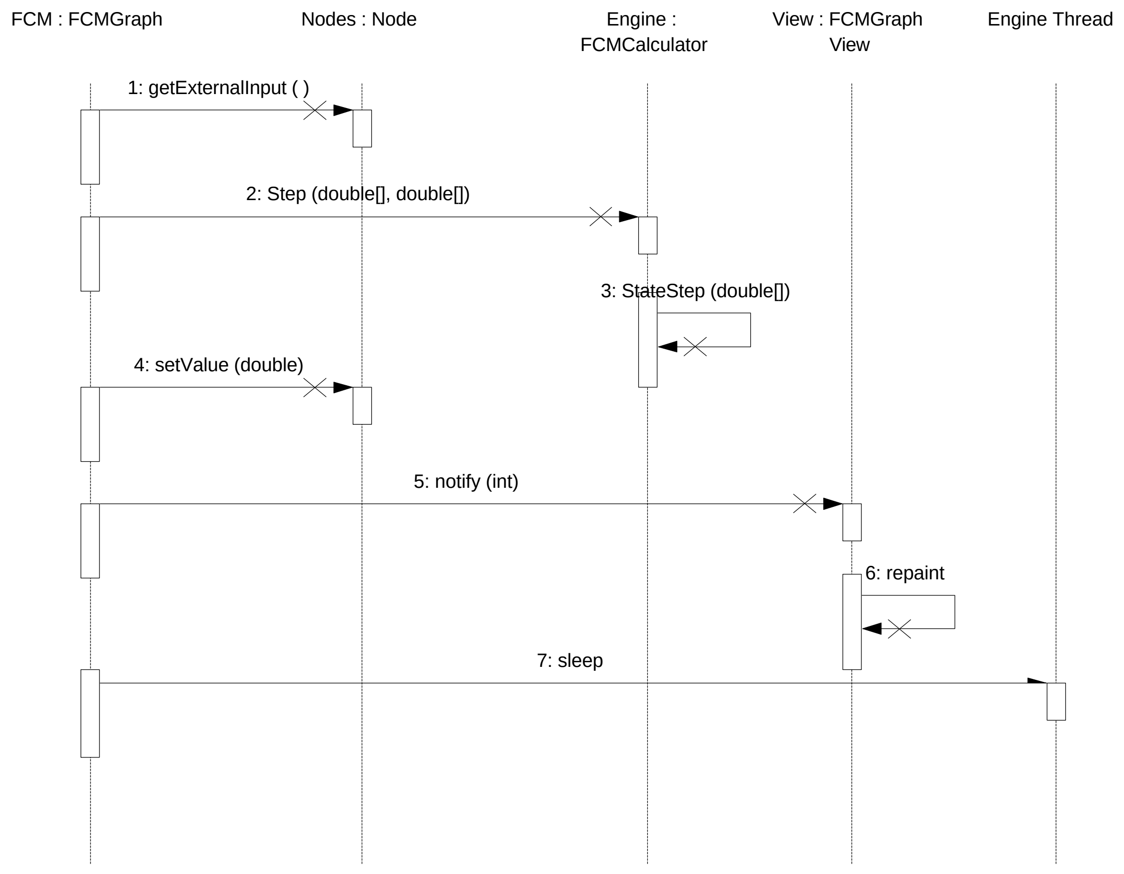

Figure 10: Message trace diagram for FCM simulator operation

Applets and Frame Windows

FCMApplication, derived from the JDK class Applet, implements the code that handles the standalone – embedded duality for the application. It contains the required bootstrapping method, main, for standalone operation. During initialization, it attempts to read a parameter, DataURL. If this value is present, it is embedded and the data file is read from the server using HTTP. If it is not, an instance of FCMFrame is created. FCMFrame is a frame window providing the menu bar.

Supporting Classes

AWT in JDK 1.0 provides minimal support for the construction of dialog boxes. Layout of the degree expected in modern GUI applications requires a good deal of creativity. To relieve the programmer of this burden, Microsoft's Visual J++ development environment provides a tool that generates the appropriate Java code given a dialog box laid out with tools adopted from Microsoft's other language tools. This is for control layout only. In the present design, the class ConceptPropertyDialog will be generated to accommodate the concept configuration dialog box. The class NodeDialog provides the logic behind the dialog box.

The classes PersistentInputStream and PersistentOutputStream are discussed in *Persistence*, above.

Machine Learning

Given qualitative information about a domain, e.g., normalized quantitative values or fuzzy values provided by experts, it is desirable to infer an FCM. Initially, this is useful to create a preliminary model of a system. A model's causal relationships may change over time as a result of policy changes. In this case, it is advisable to periodically infer an FCM and compare it to an existing model to help detect such changes. The similarity of FCMs to neural networks permits unsupervised learning of causal relationships in such situations.

A simple scheme allows causal relationships to be inferred provided no external influences are operating[7]. Given a discrete change in a concept value ΔC_i at time t and an edge strength $e_{ij}(t)$ between that conceptual node and the j^{th} node in the graph,

$$e_{ij}(t+1) = e_{ij}(t) + c_t\left[\Delta C_i \Delta C_j - e_{ij}(t)\right]$$

if ΔC_i is nonzero. If the concept value does not change, the edge weight is unchanged. The constant c_t is a learning coefficient which decreases in time. The value of the coefficient is given by

$$c_t(t) = 0.1\left[1 - \frac{t}{1.1N}\right]$$

where N is a constant controlling the rate of decrease. Some values of the learning coefficient are shown in Figure 11.

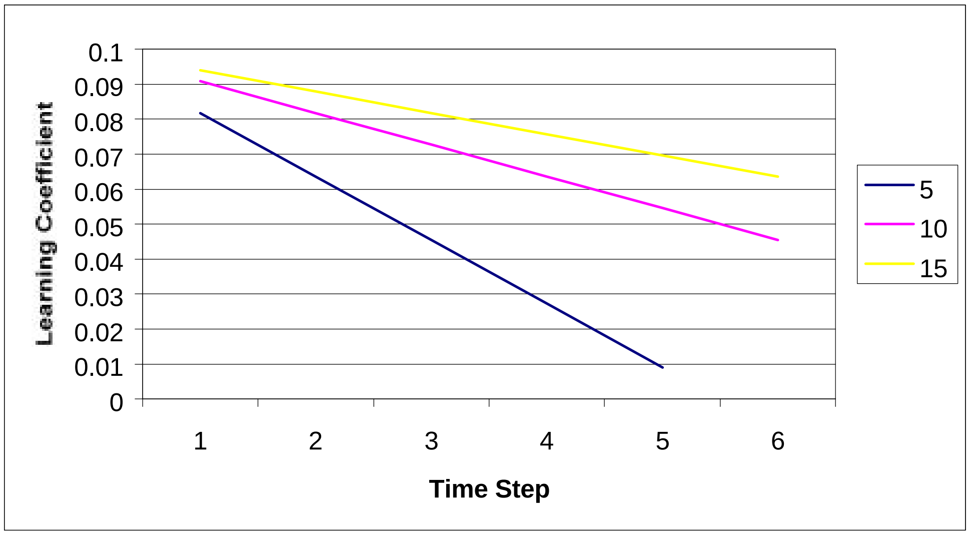

Figure 11: Learning coefficient curves

The projected application will permit learning when presented with a text file in the following format. The first n lines will be strings, each beginning with a strictly alphabetic character, denoting the names of the concept nodes. The next line will be the value of N for the learning coefficient equation. The remaining lines will consist of comma delimited state vectors in time series order. The application will create the appropriate nodes and learned edges, automatically provide a default layout, and display the resultant FCM. All objects in the system at that time will be in the same state they would have been in had they been manually created.

Summary

Fuzzy cognitive maps are qualitative tools useful for rapidly examining relationships between domain concepts. They draw on soft computing concepts from the areas of fuzzy logic and neural networks. They are computationally inexpensive and permit convenient examination of various scenarios and options.

We have presented an object oriented design arrived at using OMT for an FCM modeling application. Written in Java, it will permit desktop experimentation with FCMs and World Wide Web-based publishing and examination of FCMs built with the tool. This design anticipates a need for code reuse in the domain of digraph construction. The core classes of the application have been designed using a variation of the common model – view paradigm and segmented in such a way as to promote software reuse.

Given the project's focus on the use of FCMs, we sought aids to productivity in areas that do not directly bear on this focus. We found user interface assistance in Laffra's animation of Dijkstra's shortest path algorithm, a layout code generator in Microsoft's Java development environment, and classes supporting streamed persistence. We examined the latter's shortcomings and proposed modifications to improve performance. These proposals will be addressed as time permits in the remainder of the project.

A basic implementation of a machine learning algorithm from the literature will be incorporated in the finished application. We have reviewed the algorithm and specified a simple file format supporting its implementation.

Java's flexibility promises an interesting development project resulting in a simple yet useful tool for the study of fuzzy cognitive maps. Fuzzy cognitive maps bear study as potential aids to decision makers facing fluid and uncertain problems. We hope to reach a clearer understanding of their potential after using the application, whose design we have here presented, in the final phase of this project.

---

1 Kosko, Bart, *Fuzzy Engineering*, Prentice Hall, Upper Saddle River, New Jersey, 1997, p. 121
2 Laffra, Carla, "Dijkstra's Shortest Path Algorithm Animation in Java", 1996, http://www.cs.pace.edu/~www/javademos/DijkstraText.html
3 Erlingsson, Ulfar and Krishnamoorthy, Mukkai, "Interactive Graph Drawing", undated, http://www.cs.rpi.edu/projects/pb/graphdraw/index.html
4 Sedgewick, Robert, *Algorithms in C++*, Addison Wesley, Reading, Massachusetts, 1992, pp. 418 – 421.
5 Kosko, Bart, *Neural Networks and Fuzzy Systems*, Prentice Hall, Upper Saddle River, New Jersey, 1992, ppp. 153 – 154.
6 Cornell, Gary and Horstmann, Cay, *Core Java*, Sunsoft Press, Mountain View California, 1996, p. 487ff.
7 Op cit., Kosko (1997), p. 521