

## **An Implementation of Fuzzy Cognitive Maps**

### **Problem Definition**

The problem I worked on with this project involved Fuzzy Cognitive Maps (FCM).[1] There is a lot of research with FCMs and how to improve them. FCMs can be used to model and simulate varying environments[2, 3], behaviors of people and agents[4, 5], and help perform decision making procedures. The basic FCM is sometimes believed to be limiting. As a result, there have been proposals for extensions to FCMs. While there can be many applications for FCMs and the proposed extensions, actual FCM platforms, frameworks, or software seems to be lacking. Many existing neural network software do not include FCMs in their feature set. There are a couple freely available FCM programs[6, 7] but they generally have a limited set of features, work on limited platforms, or the source code is unavailable. If an individual wants to create an application using a FCM or perform some experimentation on FCMs and possible extensions, the individual would need to create the FCM software themselves.

### **Goals for the Project**

For this project, is to create an FCM framework. This framework will contain a base, simple implementation of FCMs. Then one or more extensions can be added to the framework that the user can optionally use; one extension in particular to include would be the Evolutionary Fuzzy Cognitive Map (E-FCM)[8].

Once the FCM framework is completed, an application will be built using it. This application will provide a simple simulation of a front yard. In this simulation, grass will grow given water and fertilizer. Lack of water, the occurrence of high and low temperature extremes, could hamper the growth of the grass. Weeds also exist in front yards and will grow with the same conditions as grass. Weeds will also attempt to overtake and kill good grass unless it is itself killed off. These are some of the objects and relationships that occur within a front yard that could be modeled by a FCM. This simulation can be run over a period of time using the E-FCM extensions to allow for probabilities of events, such as watering of the yard or poisoning of weeds, to occur.

### **Sources of Information**

[1] Kaikhah, Khosrow. *Neural Networks: Associative Memory Networks*.

[2] Dissanayake, M. and AbouRizk, S. M. 2007. Qualitative simulation of construction performance using fuzzy cognitive maps. In *Proceedings of the 39<sup>th</sup> Conference on Winter Simulation: 40 Years! The Best Is Yet To Come* (Washington D.C., December 09-12, 2007). Winter Simulation Conference. IEEE Press, Piscataway, NJ, 2134-2140.

[3] Gras, R., Devaurs, D., Wozniak, A., and Aspinall, A. 2009. An individual-based evolving predator-prey ecosystem simulation using a fuzzy cognitive map as the behavior model. *Artif. Life* 15, 4 (Oct. 2009), 423-463.

[4] Leong, P. and Chunyan, M. 2005. Fuzzy Cognitive Agents in Shared Virtual Worlds. In

*Proceedings of the 2005 international Conference on Cyberworlds* (November 23 - 25, 2005). CW. IEEE Computer Society, Washington, DC, 368-372. DOI= <http://dx.doi.org/10.1109/CW.2005.49>

[5] Luo, X., Wei, X., and Zhang, J. 2009. Game-based learning model using fuzzy cognitive map. In *Proceedings of the First ACM international Workshop on Multimedia Technologies For Distance Learning* (Beijing, China, October 23 - 23, 2009). MTDL '09. ACM, New York, NY, 67-76. DOI= <http://doi.acm.org/10.1145/1631111.1631123>

[6] *FCMapper*. [Web] <http://www.fcappers.net/joomla/>

[7] *Fuzzy Cognitive Maps*. [Web] <http://www.ochoadeaspuru.com/fuzcogmap/software.php>

[8] Y. Cai et al., "Context Modeling with Evolutionary Fuzzy Cognitive Map in Interactive Storytelling," *IEEE Int'l Conf. Fuzzy Systems* (WCCI 08), IEEE CS Press, 2008, pp. 2320–2325.

## **Solution Strategy**

The framework will be written in Java. This project should use object-oriented programming to create a component based approach to the framework. The base of the framework is the simple, basic, implementation of FCMs. Objects will include the nodes in the map as well as the links, or relationships, between the nodes. FCM extensions, such as E-FCM, will extend this basic framework to add additional features. For example, with E-FCM, each node and link will have additional fields to take into consideration the probability of an event occurring or having an effect. The framework will contain methods to run the FCM through an iteration, creating a matrix based on the link objects, accepting an input, and performing the required calculations.

With a completed FCM framework, the front yard simulation will then be created. The various events, such as fertilizing, watering, weed growth, can be created as node objects. The relationships between these events are created as link objects. When the simulation is executed, a starter input is provided. The output can be provided as the input for each iteration of the simulation. The results of each iteration can be displayed to determine if the FCM framework itself appears to be working correctly and to see if the simulation results appear to be correct overall.

## **Initial Expectation**

The expectation is to be able to complete the project. The FCM application will allow the user to be able to specify a fuzzy cognitive map, its nodes and relationships between node. The user will be able to provide this information for a basic FCM or for an E-FCM, as described above. The FCM will process the FCM for a specified number of iterations, displaying the results in terms of values of each node in the map.

Whether or not a GUI can be provided to display the map itself is unknown. But at the start of the project, this was considered to be a nice feature, but unlikely to be implemented.

## **Current Status of the Project**

The FCM application implements both basic FCMs and E-FCMs. As initially expected, it does not include a GUI to represent the map. Results of processing the FCM are displayed as text to “standard

out,” thus requiring the application to run from a command line prompt.

Also, the implementation of E-FCMs does not take full advantage of object-oriented programming (OOP) methodologies as described in the solution strategy. Ideally, the E-FCM is an extension of a basic FCM. An E-FCM node contains all of the properties of a basic FCM node, but with some additional properties. Utilizing full OOP, an E-FCM node would “extend” a basic FCM node and then declare the additional properties within the E-FCM node object. This is not being done with this implementation of the project. Rather the nodes and links for an E-FCM object are created “from scratch” and have no object-oriented relationship to basic FCM nodes and links.

### **Remaining Areas of Concern**

If there is an area of concern, it may be the possibility that I failed to understand how the values of nodes are updated, particularly for the implementation of an E-FCM. The paper describing E-FCMs did not spend a lot of time explaining the formulas for calculating the values for nodes on each iteration of the map. The example provided in their paper used binary values where as this project is allowing the use of real and fuzzy values. The paper also did not go into much detail on the probability of events occurring, including the possibility of self mutation. Therefore, some assumptions were made on how to implement that feature.

The algorithm to compute the values were created as I understood them with some assumptions. However, if there was a misunderstanding, then the results may differ from what a user may have been expecting.

### **Technical Lessons Learned**

One of the biggest lessons learned was one that is often quickly forgotten. It is the need for a clear, well thought-out design before implementing a software project. While I prepared some written notes about how to proceed with the implementation, I did not formalize any design documents. This led to some confusion in the middle parts of the project when attempting to implement E-FCM as an extension of a basic FCM. I found it quicker to simply implement an E-FCM as an independent set of classes. However, had I prepared a design sufficiently before attempting to implement the project the confusion may not have occurred or have been as high as it was.

In addition, in terms of writing research papers I realize there is a balance between the desire to provide as much detail about a process as possible and the page limitations set by the conference or journal. However, if there are important aspects of the research, such as how formulas are determined and how to apply them, that should be in the final paper. Another option is to write two papers the first one being a theoretical paper discussing the formulas and algorithms. Then write a second paper that does an implementation or shows examples of the algorithms at work. This would allow people in the future to be able to more easily interpret what the paper(s) discuss.

### **Recommendations for Future Projects**

A recommendation for myself on future projects would be to attempt to complete a project that is more closely tied to other work or extends my knowledge in areas that I am more interested in. This project could have accomplished this had I spent more time in the design phase, as mentioned in the previous section. Having done so, I could have explored more issues in software engineering and design such as

UML diagrams and object-oriented programming practices. While the implementation itself may still be a “one-off” program, never to be touched again, the process of completing the project would be beneficial.

## Appendix A: Source Code

```
package driver;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import basicFCM.BasicFCM;
import eFCM.EFCM;

public class Driver {

;;; driver.Driver.main()
;;;
;;; Purpose:      This is the entry point into the FCM application / framework.
;;; Returns:     Nothing.
;;; Arguments:   String: contains the path and name of a driver file to process.
;;; Side Effects: None
;;; Error Handling: Checking for invalid filename, or errors reading from the specified file.
;;; Calls:      BasicFCM.parseDriverFile(), EFCM.parseDriverFile()
;;; Called By:  Is called by running the application.
    public static void main(String[] args) {
        File driverFile = new File(args[0]);
        int rounds = Integer.parseInt(args[1]);

        // Welcome to the Fuzzy Cognitive Map Software....
        // Check the first line of the driver file to determine what type
of FCM is being used.
        try {
            BufferedReader input = new BufferedReader(new
FileReader(driverFile));
            String line = input.readLine();
            if (line == null)
                System.exit(0); // Nothing in the file = nothing to
do.

            if (line.equalsIgnoreCase("basicFCM"))
                BasicFCM.parseDriverFile(driverFile, rounds);
            else if (line.equalsIgnoreCase("eFCM"))
                EFCM.parseDriverFile(driverFile, rounds);
            // else if line == ... blah blah blah ...
        } catch (IOException e) {
            System.out.println("Problems working with driver file.");
            e.printStackTrace();
        }
    } // end main
}


```

---

```
package basicFCM;
```

```
import java.io.BufferedReader;
import java.io.File;
```

```

import java.io.FileReader;
import java.io.IOException;
import java.text.DecimalFormat;
import java.util.ArrayList;
import java.util.Iterator;

public class BasicFCM {

    // Global variables
    private static ArrayList<Node> nodes = null;
    private static Link[][] links = null;;
    private static int rounds;

;;; BasicFCM.parseDriverFile()
;;;
;;; Purpose:      This is the entry point into the Basic FCM code. It parses the input files, creates the FCM and
calls the processing method.
;;; Returns:      Nothing. It will output the FCM object states after each iteration.
;;; Arguments:    File, a file containing the text driver file to process. Int: incRounds, the number of times to process
the FCM.
;;; Error Handling: It checks for bad or non-existent files and for problems reading from the specified files.
;;; Calls:        createNodes() and createLinks() methods to create and modify nodes and links in the FCM,
process() to calculate the new states for the objects in the FCM.
;;; Called By:    driver.Driver.main()
    public static void parseDriverFile(File file, int incRounds) {
        // This is entry point to BasicFCM... Do some initialization.
        nodes = new ArrayList(); // links are initialized later.
        rounds = incRounds;

        // Working variables
        int count = 0;
        String line;
        File nodeFile, linkFile;

        try {
            BufferedReader input = new BufferedReader(new FileReader(file));
            try {
                while ((line = input.readLine()) != null) {
                    if (count == 0) {
                        // reading first line of file... skip
                        count++;
                        continue;
                    }
                    else if (count == 1) {
                        // Second line is the nodes file
                        nodeFile = new File(line);
                        createNodes(nodeFile);
                        count++;
                    }
                    else if (count == 2) {
                        // third line is the links file
                        linkFile = new File(line);
                        createLinks(linkFile);
                        count++;
                        break; // TODO not expecting any more lines in this file.
                    }
                }
            }
        }
    }
}

```

```

        } // end while
    } finally {
        input.close();
    }
} catch (IOException e) {
    System.out.println("Basic FCM has problems working with driver file.");
    e.printStackTrace();
} // Done reading file....

// Print out header
System.out.println(getHeader());
System.out.println(getWeights());

// Print out first set of values
System.out.println("-----");
System.out.println(getValues());

for (int i=0; i < rounds; i++) {
    // Process the FCM.
    nodes = process();

    // Print out set of values
    System.out.println(getValues());
}

} // end parseDriverFile()

```

```

;;; basicFCM.BasicFCM.createNodes()

```

```

;;;

```

```

;;; Purpose:      Given a file listing all nodes in a FCM, this method will parse the file and create the nodes.

```

```

;;; Returns:      Nothing

```

```

;;; Arguments:    File, the file containing the information about nodes in the FCM

```

```

;;; Error Handling: The method checks for bad or missing file names and for problems in reading from the file.

```

```

;;; Calls:        There are calls to methods within the basicFCM.Node class to create and modify nodes.

```

```

;;; Called By:    basicFCM.BasicFCM.parseDriverFile()

```

```

private static void createNodes(File nodeFile) {

```

```

    // Working variables

```

```

    String line;

```

```

    String[] parts;

```

```

    String name;

```

```

    double value;

```

```

    Node thisNode;

```

```

    // Read from the file....

```

```

    try {

```

```

        BufferedReader input = new BufferedReader(new FileReader(nodeFile));

```

```

        try {

```

```

            while ((line = input.readLine()) != null) {

```

```

                // parse the line.

```

```

                parts = line.split(" ");

```

```

                name = parts[0];

```

```

                value = Double.parseDouble(parts[1]);

```

```

                // create the node.

```

```

                thisNode = new Node(name, value);

```

```

        // add it to the set.
        nodes.add(thisNode);
    } // end while
} finally {
    input.close();
}
} catch (IOException e) {
    System.out.println("Basic FCM has problems reading form node file.");
    e.printStackTrace();
}
} // end createNodes()

```

```

;;; basicFCM.BasicFCM.createLinks()

```

```

;;;

```

```

;;; Purpose:      Given a file listing all links in a FCM, this method will parse the file and create the links.

```

```

;;; Returns:      Nothing

```

```

;;; Arguments:    File, the file containing the information about links in the FCM

```

```

;;; Error Handling: The method checks for bad or missing file names and for problems in reading from the file.

```

```

;;; Calls:        There are calls to methods within the basicFCM.Link class to create and modify links.

```

```

;;; Called By:    basicFCM.BasicFCM.parseDriverFile()

```

```

private static void createLinks(File linkFile) {

```

```

    // Initialize global links

```

```

    int count = nodes.size();

```

```

    links = new Link[count][count];

```

```

    for (int i=0; i < count; i++)

```

```

        for (int j=0; j < count; j++)

```

```

            links[i][j] = null;

```

```

    // Working variables

```

```

    Link thisLink;

```

```

    String line;

```

```

    String[] parts;

```

```

    Node fromNode, toNode;

```

```

    int fromIndex, toIndex;

```

```

    // Read the file.

```

```

    try {

```

```

        BufferedReader input = new BufferedReader(new FileReader(linkFile));

```

```

        try {

```

```

            while ((line = input.readLine()) != null) {

```

```

                // Parse the line.

```

```

                parts = line.split(" ");

```

```

                // Get the from node.

```

```

                fromNode = getNode(parts[0]);

```

```

                fromIndex = nodes.indexOf(fromNode);

```

```

                // Get to node

```

```

                toNode = getNode(parts[1]);

```

```

                toIndex = nodes.indexOf(toNode);

```

```

                // Create the link

```

```

                thisLink = new Link(Double.parseDouble(parts[2]),

```

```

                Boolean.parseBoolean(parts[3]), fromNode, toNode);

```

```

                // add it to the array.
                links[fromIndex][toIndex] = thisLink;
            } // end while
        } finally {
            input.close();
        }
    } catch (IOException e) {
        System.out.println("BasicFCM has problems reading links file.");
        e.printStackTrace();
    }
} // end createLinks()

```

```

;;; basicFCM.BasicFCM.getNode()

```

```

;;;
;;; Purpose:      This is a helper function that will return a Node object based on the string name of the node.
;;; Returns:     Node object
;;; Arguments:   String containing the name of the node to get the object for.
;;; Error Handling: None.
;;; Calls:      basicFCM.Node.getName()
;;; Called By:  basicFCM.BasicFCM.createLinks()

```

```

private static Node getNode(String nodeName) {
    Node thisNode;
    Iterator itr = nodes.iterator();
    while (itr.hasNext()) {
        thisNode = (Node) itr.next();
        if (thisNode.getName().equals(nodeName))
            return thisNode;
    }

    // nodeName wasn't found...
    return null;
}

```

```

;;; basicFCM.BasicFCM.getHeader()

```

```

;;;
;;; Purpose:      This method will create a string containing the names of all the Nodes in the FCM.
;;; Returns:     String with the names of all nodes in the FCM.
;;; Arguments:   None
;;; Error Handling: None
;;; Calls:      basicFCM.Node.getname()
;;; Called By:  basicFCM.BasicFCM.parseDriverFile()

```

```

private static String getHeader() {
    String output = "";

    Iterator itr = nodes.iterator();
    while (itr.hasNext()) {
        Node thisNode = (Node) itr.next();
        output += thisNode.getName()+"\t";
    }

    return output;
} // end getHeader()

```

```

;;; basicFCM.BasicFCM.getValues()

```

```

;;;
;;; Purpose:      This method will create a string containing the values of all nodes in the FCM.

```

```
;;; Returns:      String containing the values of nodes in the FCM.
;;; Arguments:   None
;;; Error Handling: None
;;; Calls:       basicFCM.Node.getValue()
;;; Called By:   basicFCM.BasicFCM.parseDriverFile()
```

```
private static String getValues() {
    String output = "";

    DecimalFormat format = new DecimalFormat("0.000");

    Iterator itr = nodes.iterator();
    while (itr.hasNext()) {
        Node thisNode = (Node) itr.next();
        output += format.format(thisNode.getValue()+"\t";
    }

    return output;
} // end getValues()
```

```
;;; basicFCM.BasicFCM.getWeights()
;;;
;;;
```

```
;;; Purpose:      This method creates a string containing the weight matrix of the FCM.
;;; Returns:      A string representing the weight matrix of the FCM.
;;; Arguments:    None
;;; Error Handling: None
;;; Calls:        basicFCM.Link.getValue()
;;; Called By:    None (in this implementation).
```

```
private static String getWeights() {
    String output = "";
    int size = links.length;

    for (int i=0; i < size; i++) {
        for (int j=0; j < size; j++) {
            if (links[i][j] == null)
                output += "0.0\t";
            else if (links[i][j].isPositive())
                output += links[i][j].getValue()+"\t";
            else
                output += "-" + links[i][j].getValue()+"\t";
        } // end inner for
        output += "\n";
    } // end outer for

    return output;
} // end getWeights()
```

```
;;; basicFCM.BasicFCM.process()
;;;
;;;
```

```
;;; Purpose:      This method performs the matrix multiplications to produce a new set of values for each node in
the FCM.
;;; Returns:      An array of Node objects containing each node in the FCM.
;;; Arguments:    None
;;; Error Handling: None
;;; Calls:        basicFCM.Node.getValue(), basicFCM.Node.getName()
;;; Called By:    basicFCM.BasicFCM.parseDriverFile()
public static ArrayList<Node> process() {
```

```

// return variable
ArrayList<Node> newNodes = new ArrayList();

// Working variables
int size = nodes.size();
double result;

// Perform 'matrix multiplication'
for (int i=0; i < size; i++) {
    result = 0;

    for (int j=0; j < size; j++) {
        if (links[j][i] == null)
            ;
        else if (links[j][i].isPositive())
            result += nodes.get(j).getValue()*links[j][i].getValue();
        else
            result -= nodes.get(j).getValue()*links[j][i].getValue();
    } // end inner for

    // Add result to newNodes list
    //nodes.get(i).setValue(result);
    newNodes.add(new Node(nodes.get(i).getName(), result));
} // end outer for

return newNodes;
} // end process()
}

```

---

```

package basicFCM;

```

```

public class Node {
    private String name;
    private double value;

```

```

;;; basicFCM.Node.Node()

```

```

;;;

```

```

;;; Purpose:      Constructor for creating Node objects.

```

```

;;; Returns:     N/A

```

```

;;; Arguments:   String containing the name of the node object; double the initial value of the Node object

```

```

;;; Error Handling: None

```

```

;;; Calls:      None

```

```

;;; Called By:  basicFCM.BasicFCM.createNodes()

```

```

    public Node(String name, double value) {
        this.name = name;
        this.value = value;
    }

```

```

;;; basicFCM.Node.setName()

```

```

;;;

```

```

;;; Purpose:     Sets the name of the node object.

```

```

;;; Returns:    Nothing.

```

```

;;; Arguments:  String containing the name of the Node object.

```

```

;;; Error Handling: None.

```

```

;;; Calls:          None
;;; Called By:
    public void setName(String name) {
        this.name = name;
    }

;;; basicFCM.Node.getName()
;;;
;;;
;;; Purpose:       Returns the name of the Node object.
;;; Returns:      String containing the name of the Node object.
;;; Arguments:    None.
;;; Error Handling: None.
;;; Calls:       None.
;;; Called By:    basicFCM.BasicFCM.process(), basicFCM.BasicFCM.getHeader()
    public String getName() {
        return name;
    }

;;; basicFCM.Node.setValue()
;;;
;;;
;;; Purpose:       Sets the current value of the Node object.
;;; Returns:      Nothing
;;; Arguments:    Double representing the value to set the Node object to.
;;; Error Handling: None
;;; Calls:       None
;;; Called By:    basicFCM.BasicFCM.process(), basicFCM.BasicFCM.createNodes()
    public void setValue(double value) {
        this.value = value;
    }

;;; basicFCM.Node.getValue()
;;;
;;;
;;; Purpose:       Returns the current value of the Node object.
;;; Returns:      Double representing the value of the node object.
;;; Arguments:    None
;;; Error Handling: None
;;; Calls:       None
;;; Called By:    basicFCM.BasicFCM.process(); basicFCM.BasicFCM.getValues()
    public double getValue() {
        return value;
    }

}

```

---

```

package basicFCM;

```

```

public class Link {
    private double value;
    private boolean positive;
    private Node from;
    private Node to;
}

```

```

;;; basicFCM.Link.Link()
;;;
;;;
;;; Purpose:       Constructor for Link class
;;; Returns:      Nothing

```

;;; Arguments: Double, the value/multiplier of the link; boolean, whether the link is a excitatory or inhibitory;  
Node where the link originates; Node where the link terminates.

;;; Error Handling: None

;;; Calls: None

;;; Called By: basicFCM.BasicFCM.createLinks()

```
public Link(double value, boolean positive, Node from, Node to) {
    this.value = value;
    this.positive = positive;
    this.from = from;
    this.to = to;
}
```

;;; basicFCM.Link.setValue()

;;;

;;;

;;; Purpose: Sets the value for the current Link object.

;;; Returns: None.

;;; Arguments: Double, the value/multiplier of the link.

;;; Error Handling: None

;;; Calls: None

;;; Called By: None

```
public void setValue(double value) {
    this.value = value;
}
```

;;; basicFCM.Link.getValue()

;;;

;;;

;;; Purpose: Returns the value for the current Link object.

;;; Returns: Double, the value/multiplier of the link.

;;; Arguments: None

;;; Error Handling: None

;;; Calls: None

;;; Called By: basicFCM.BasicFCM.process()

```
public double getValue() {
    return value;
}
```

;;; basicFCM.Link.setPositive()

;;;

;;;

;;; Purpose: Sets the 'positive' variable of the Link object.

;;; Returns: None

;;; Arguments: Boolean, whether the link is excitatory or not.

;;; Error Handling: None

;;; Calls: None

;;; Called By: None

```
public void setPositive(boolean positive) {
    this.positive = positive;
}
```

;;; basicFCM.Link.isPositive()

;;;

;;;

;;; Purpose: Returns a boolean on whether the Link object is excitatory or not.

;;; Returns: Boolean on whether the Link object is excitatory or not.

;;; Arguments: None

;;; Error Handling: None

;;; Calls: None

;;; Called By: basicFCM.BasicFCM.process()

```
public boolean isPositive() {
```

```

        return positive;
    }

;;; basicFCM.Link.setFrom()
;;;
;;; Purpose:      Sets the originating Node for the Link object.
;;; Returns:     None
;;; Arguments:   Node where the Link object originates.
;;; Error Handling: None
;;; Calls:      None
;;; Called By:  None
    public void setFrom(Node from) {
        this.from = from;
    }

;;; basicFCM.Link.getFrom()
;;;
;;; Purpose:      Returns the Node that the Link object originates from.
;;; Returns:     Node from where the Link object originates from.
;;; Arguments:   None
;;; Error Handling: None
;;; Calls:      None
;;; Called By:  None
    public Node getFrom() {
        return from;
    }

;;; basicFCM.Link.setTo()
;;;
;;; Purpose:      Sets the Node that the Link object terminates at.
;;; Returns:     None
;;; Arguments:   Node where the Link object terminates at.
;;; Error Handling: None
;;; Calls:      None
;;; Called By:  None
    public void setTo(Node to) {
        this.to = to;
    }

;;; basicFCM.link.getTo()
;;;
;;; Purpose:      Returns the Node where the Link object terminates at.
;;; Returns:     Node where the Link object terminates at.
;;; Arguments:   None
;;; Error Handling: None
;;; Calls:      None
;;; Called By:  None
    public Node getTo() {
        return to;
    }
}

```

---

```

package eFCM;

```

```

import java.io.BufferedReader;
import java.io.File;

```

```

import java.io.FileReader;
import java.io.IOException;
import java.text.DecimalFormat;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.Random;

```

```

public class EFCM {

```

```

    private static ArrayList<ENode> nodes;
    private static ELink[][] links;
    private static int rounds;
    private static double k1=0.5;
    private static double k2=0.05;

```

```

;;; eFCM.EFCM.parseDriverFile()

```

```

;;;

```

```

;;; Purpose:      This is the entry point into the E-FCM code. It will parse the specified file to find the other files
to get the nodes and links for the E-FCM graph and will then start the processing of the FCM.

```

```

;;; Returns:      None

```

```

;;; Arguments:    File of the driver file; Int for the number of rounds to run the FCM for.

```

```

;;; Error Handling: Checks whether the file exists and if there are problems reading from the file.

```

```

;;; Calls:        eFCM.EFCM.createNodes(), eFCM.EFCM.createLinks(), eFCM.EFCM.process()

```

```

;;; Called By:    driver.Driver()

```

```

    public static void parseDriverFile(File file, int incRounds) {
        // This is the entry point to eFCM ... do some initialization.
        nodes = new ArrayList(); // Links will be initialized later.
        rounds = incRounds;

        // Working variables
        int count = 0;
        String line;
        File nodeFile, linkFile;

        try {
            BufferedReader input = new BufferedReader(new FileReader(file));
            try {
                while ((line = input.readLine()) != null) {
                    if (count == 0) {
                        // Reading first line of the file... skip
                        count++;
                        continue;
                    }
                    else if (count == 1) {
                        // Second line is the node file
                        nodeFile = new File(line);
                        createNodes(nodeFile);
                        count++;
                    }
                    else if (count == 2) {
                        // third line is the links file.
                        linkFile = new File(line);
                        createLinks(linkFile);
                        count++;
                        break; // TODO not expecting more lines.
                    }
                }
            }
            finally {
                input.close();
            }
        }
    }

```

```

    }
} catch (IOException e) {
    System.out.println("eFCM has problems working with driver
file.");
    e.printStackTrace();
} // done reading file...

process();

} // end parseDriverFile()

;;; eFCM.EFCM.createLinks()
;;;
;;; Purpose:      This method parses a file with a list of links between nodes, creates the individual links, and
forms the 'weight' matrix.
;;; Returns:     None
;;; Arguments:   File with the links in the E-FCM graph.
;;; Error Handling: Checks that the file is a valid file and if there are problems reading from the file.
;;; Calls:      eFCM.ELink.ELink()
;;; Called By:  eFCM.EFCM.parseDriverFile()
    private static void createLinks(File file) {
        // Initialize global links
        int count = nodes.size();
        links = new ELink[count][count];
        for (int i=0; i < count; i++)
            for (int j=0; j < count; j++)
                links[i][j] = null;

        // Working variables
        ELink thisLink;
        String line;
        String[] parts;
        ENode fromNode, toNode;
        int fromIndex, toIndex;
        double value;
        boolean positive;
        double prob;

        // Read the file
        try {
            BufferedReader input = new BufferedReader(new FileReader(file));
            try {
                while ((line = input.readLine()) != null) {
                    // parse the line
                    parts = line.split(" ");

                    // Get the from node
                    fromNode = getNode(parts[0]);
                    fromIndex = nodes.indexOf(fromNode);

                    // Get the to node
                    toNode = getNode(parts[1]);
                    toIndex = nodes.indexOf(toNode);

                    // Get other variables
                    value = Double.parseDouble(parts[2]);
                    positive = Boolean.parseBoolean(parts[3]);
                    prob = Double.parseDouble(parts[4]);
                }
            }
        }
    }
}

```

```

        // Create the link.
        thisLink = new ELink(fromNode, toNode, value,
positive, prob);

        // Add it to the array
        links[fromIndex][toIndex] = thisLink;
    } // end while
} finally {
    input.close();
}
} catch (IOException e) {
    System.out.println("eFCM has problems working with link file.");
    e.printStackTrace();
} // done reading file

} // end createLinks()

```

```

;;; eFCM.EFCM.createNodes()

```

```

;;;

```

```

;;; Purpose:      This method parses the specified file with Nodes for the E-FCM graph, creates the nodes, and
adds them to an array.

```

```

;;; Returns:      None

```

```

;;; Arguments:    File containing the nodes in the E-FCM graph.

```

```

;;; Error Handling: Checks that the file is valid and if there are problems reading from the file.

```

```

;;; Calls:        eFCM.ENode.ENode()

```

```

;;; Called By:    eFCM.EFCM.parseDriverFile(0

```

```

    private static void createNodes(File file) {
        // Working variables
        String line;
        String[] parts;
        String name;
        double value;
        int timeVal;
        double mutateProb;
        ENode thisNode;

        // Read from the file...
        try {
            BufferedReader input = new BufferedReader(new FileReader(file));

            try {
                while ((line = input.readLine()) != null) {
                    // parse the line
                    parts = line.split(" ");
                    name = parts[0];
                    value = Double.parseDouble(parts[1]);
                    timeVal = Integer.parseInt(parts[2]);
                    mutateProb = Double.parseDouble(parts[3]);

                    // Create the node.
                    thisNode = new ENode(name, value, timeVal,
mutateProb);

                    // Add it to the set.
                    nodes.add(thisNode);
                }
            } finally {
                input.close();
            }
        } // done reading file...
    }

```

```

    } catch (IOException e) {
        System.out.println("eFCM has problems working with nodes file.");
        e.printStackTrace();
    }
} // end createNodes()

```

;;; eFCM.EFCM.getNode()

;;; Purpose: Given a string representing the name of a Node, this method will return the Node object.  
 ;;; Returns: ENode containing the node Object being sought, or null if the Node object was not found.  
 ;;; Arguments: String representing the name of the Node being sought.  
 ;;; Error Handling: None.  
 ;;; Calls: eFCM.ENode.getName()  
 ;;; Called By: eFCM.EFCM.createLinks()

```

private static ENode getNode(String nodeName) {
    ENode thisNode;
    Iterator itr = nodes.iterator();
    while (itr.hasNext()) {
        thisNode = (ENode) itr.next();
        if (thisNode.getName().equals(nodeName))
            return thisNode;
    }

    // nodeName wasn't found...
    return null;
}

```

;;; eFCM.EFCM.getHeader()

;;; Purpose: This method returns a tab delimited string representing the names of all Nodes in the E-FCM graph.  
 ;;; Returns: String representing the Node names in the graph.  
 ;;; Arguments: None  
 ;;; Error Handling: None  
 ;;; Calls: eFCM.ENode.getName()  
 ;;; Called By: eFCM.EFCM.process(0)

```

private static String getHeader() {
    String output = "";

    Iterator itr = nodes.iterator();
    while (itr.hasNext()) {
        ENode thisNode = (ENode) itr.next();
        output += thisNode.getName()+"\t";
    }

    return output;
} // end getHeader()

```

;;; eFCM.EFCM.getValues()

;;; Purpose: This method returns a tab delimited string representing the current state values of all Nodes in the E-FCM graph.  
 ;;; Returns: String representing the state values of all Nodes.  
 ;;; Arguments: None  
 ;;; Error Handling: None  
 ;;; Calls: eFCM.ENode.getValue()

```

;;; Called By:    eFCM.EFCM.process()
    private static String getValues() {
        String output = "";

        DecimalFormat format = new DecimalFormat("0.000");

        Iterator itr = nodes.iterator();
        while (itr.hasNext()) {
            ENode thisNode = (ENode) itr.next();
            output += format.format(thisNode.getValue())+"\t";
        }

        return output;
    }

;;; eFCM.EFCM.process()
;;;
;;; Purpose:      This method processes the E-FCM graph as per the E-FCM paper.[8]
;;; Returns:      None
;;; Arguments:    None
;;; Error Handling: None
;;; Calls:        eFCM.ENode.ENode(), eFCM.ENode.getValue(), eFCM.ENode.getSelfMutate(),
eFCM.ENode.getTimeSlice(), eFCM.ENode.getName(), eFCM.ELink.getProb(), eFCM.ELink.isPositive(),
eFCM.ELink.getValue()
;;; Called By:    eFCM.EFCM.parseDriverFile()
    private static void process() {
        // Print out header for the output.
        System.out.println(getHeader());
        System.out.println(getValues());

        // Working variables
        int timer = 0;
        int numNodes = nodes.size();
        double result;
        ArrayList<ENode> newNodes;

        while (timer < rounds) {
            newNodes = new ArrayList();

            for (int i=0; i < numNodes; i++) {
                result = 0;

                // See if node changes this time.
                if ( timer % nodes.get(i).getTimeSlice() != 0) {
                    newNodes.add(new ENode(nodes.get(i).getName(),
nodes.get(i).getValue()+result, nodes.get(i).getTimeSlice(),
nodes.get(i).getSelfMutate()));
                    continue;
                }

                //System.out.println("Updating value for
"+nodes.get(i).getName());

                for (int j=0; j < numNodes; j++) {
                    if (links[j][i] == null)
                        ; // continue;
                    else if (isProbable(links[j][i].getProb())) {
                        if (links[j][i].isPositive()) {
                            result +=

```

```

nodes.get(j).getValue()*links[j][i].getValue(); // Add effect of j on i.
//System.out.println("\tAdding to
summation, "+nodes.get(j).getName()+" value of "+nodes.get(j).getValue()+" times
weight of "+links[j][i].getValue());
        } else {
            result -=
nodes.get(j).getValue()*links[j][i].getValue(); // Add effect of j on i.
//System.out.println("\tSubtracting from
summation, "+nodes.get(j).getName()+" value of "+nodes.get(j).getValue()+" times
weight of "+links[j][i].getValue());
        }
    } // end inner for

    // Multiply the above summation by the weight factor.
    result = k1*result;

    // see if node self-mutates
    if (isProbable(nodes.get(i).getSelfMutate())) {
        System.out.println("Node is self mutating:
"+nodes.get(i).getName());
        result += k2*nodes.get(i).getValue();
    }

    // Apply activation function:
    //System.out.println("\tResult before activation function:
"+result);

    result = activeFunc(result);
    //System.out.println("\tResult after activation function:
"+result);

    // result now = change.
    newNodes.add(new ENode(nodes.get(i).getName(),
nodes.get(i).getValue()+result, nodes.get(i).getTimeSlice(),
nodes.get(i).getSelfMutate()));
    newNodes.get(i).setChange(result);
} // end outer for

// Finished one round of calculations.
timer++;
nodes = newNodes;
System.out.println(getValues());
} // end while

} // end process

```

```

;;; eFCM.EFCM.isProbable()

```

```

;;;

```

```

;;; Purpose: This method generates a random number from zero to one. It then compares this random number
with the provided double and returns a boolean as to whether the random number is less than or equal to the provided
number. In other words, it returns true if the event is supposed to occur.

```

```

;;; Returns: Boolean, true if a random number is less than or equal to the provided number. AKA, whether
the event will occur.

```

```

;;; Arguments: Double, the probability that an event will occur.

```

```

;;; Error Handling: None

```

```

;;; Calls: None

```

```

;;; Called By: eFCM.EFCM.process()

```

```

private static boolean isProbable(double prob) {
    Random generator = new Random();

```

```

        double num = generator.nextDouble();

        if (num <= prob)
            return true;
        else
            return false;
    } // end isProbable()

;;; eFCM.EFCM.activeFunc()
;;;
;;; Purpose:      This method performs a bipolar sigmoid activation function.
;;; Returns:     Double resulting from the activation function.
;;; Arguments:   Double representing the input to the activation function.
;;; Error Handling: None
;;; Calls:      None.
;;; Called By:   eFCM.EFCM.process()
        private static double activeFunc(double change) {

            change = (1 - Math.pow(Math.E, -change)) / (1 + Math.pow(Math.E,
-change));

            return change;
        }
}

```

---

```
package eFCM;
```

```
public class ENode {
```

```

    private String name;
    private double value;
    private int timeSlice;
    private double selfMutate;
    private double change;

```

```
;;; eFCM.ENode.ENode()
```

```

;;;
;;; Purpose:      Constructor for creating Nodes in an E-FCM graph.
;;; Returns:     None.
;;; Arguments:   String, representing the name of the Node; double for the state value of the node; int for the time
slice value; double, for the probability the node self mutates.
;;; Error Handling: None
;;; Calls:      None
;;; Called By:   eFCM.EFCM.createNodes, eFCM.EFCM.process()
        public ENode(String name, double value, int timeVal, double mutateProb) {
            this.setName(name);
            this.setValue(value);
            this.timeSlice = timeVal;
            this.selfMutate = mutateProb;
            this.setChange(0.0);
        }

```

```
;;; eFCM.ENode.setName()
```

```

;;;
;;; Purpose:      This method sets the name of the Node object.
;;; Returns:     None

```

```

;;; Arguments:      String representing the name of the Node object.
;;; Error Handling: None
;;; Calls:          None
;;; Called By:      None
    public void setName(String name) {
        this.name = name;
    }

;;; eFCM.ENode.getName()
;;;
;;; Purpose:        Returns the name of the Node object.
;;; Returns:        String representing the name of the Node object.
;;; Arguments:      None
;;; Error Handling: None
;;; Calls:          None
;;; Called By:      eFCM.EFCM.process(), eFCM.EFCM.getHeader()
    public String getName() {
        return name;
    }

;;; eFCM.ENode.setValue()
;;;
;;; Purpose:        Sets the state value of the Node object.
;;; Returns:        None
;;; Arguments:      Double representing the state value of the Node object
;;; Error Handling: None
;;; Calls:          None
;;; Called By:      None
    public void setValue(double value) {
        this.value = value;
    }

;;; eFCM.ENode.getValue()
;;;
;;; Purpose:        The method returns the current state value of the Node object.
;;; Returns:        Double, representing the state value of the Node object.
;;; Arguments:      None
;;; Error Handling: None
;;; Calls:          None
;;; Called By:      eFCM.EFCM.process(), eFCM.EFCM.getValues()
    public double getValue() {
        return value;
    }

;;; eFCM.ENode.setTimeSlice()
;;;
;;; Purpose:        This method sets the time slice value for the Node object.
;;; Returns:        None.
;;; Arguments:      Integer, the time slice value for the Node object.
;;; Error Handling: None
;;; Calls:          None
;;; Called By:      None
    public void setTimeSlice(int timeSlice) {
        this.timeSlice = timeSlice;
    }

;;; eFCM.eNode.getTimeSlice()

```

```

...
;;; Purpose:      This method returns the current time slice for the Node object.
;;; Returns:     Integer representing the time slice for the Node object.
;;; Arguments:   None
;;; Error Handling: None
;;; Calls:       None
;;; Called By:   eFCM.EFCM.process()
    public int getTimeSlice() {
        return timeSlice;
    }

;;; eFCM.ENode.setSelfMutate()
...
;;; Purpose:      This method sets the self mutation probability for the Node object.
;;; Returns:     None
;;; Arguments:   Double representing the probability the Node object self-mutates.
;;; Error Handling: None
;;; Calls:       None
;;; Called By:   None
    public void setSelfMutate(double selfMutate) {
        this.selfMutate = selfMutate;
    }

;;; eFCM.ENode.getSelfMutate()
...
;;; Purpose:      This method returns the probability value for the Node object to self-mutate.
;;; Returns:     Double representing the probability the Node object self-mutates.
;;; Arguments:   None
;;; Error Handling: None
;;; Calls:       None
;;; Called By:   eFCM.EFCM.process()
    public double getSelfMutate() {
        return selfMutate;
    }

;;; eFCM.ENode.setChange()
...
;;; Purpose:      This method sets the relative change of the state value for the Node object.
;;; Returns:     None
;;; Arguments:   Double representing the amount of change in the state value.
;;; Error Handling: None
;;; Calls:       None
;;; Called By:   eFCM.EFCM.process()
    public void setChange(double change) {
        this.change = change;
    }

;;; eFCM.ENode.getChange()
...
;;; Purpose:      This method returns the relative change in the state value for the Node object.
;;; Returns:     Double representing the amount of change in the Node object's state value.
;;; Arguments:   None
;;; Error Handling: None
;;; Calls:       None
;;; Called By:   eFCM.EFCM.process()
    public double getChange() {
        return change;
    }

```

```
}  
}
```

---

```
package eFCM;
```

```
public class ELink {
```

```
    private double value;  
    private boolean positive;  
    private double prob;  
    private ENode fromNode;  
    private ENode toNode;
```

```
;;; eFCM.ELink.ELink()  
;;;
```

```
;;; Purpose:      This is the constructor for creating an ELink object.  
;;; Returns:      None  
;;; Arguments:    ENode where the link originates from; ENode, where the link terminates at; double, for the  
value/multiplier of the link; boolean, whether the link is excitatory or not; double, representing the probability the link  
is activated.
```

```
;;; Error Handling: None
```

```
;;; Calls:        None
```

```
;;; Called By:    eFCM.EFCM.createLinks()
```

```
    public ELink(ENode fromNode, ENode toNode, double value, boolean positive,  
double prob) {
```

```
        this.fromNode = fromNode;  
        this.toNode = toNode;  
        this.value = value;  
        this.positive = positive;  
        this.prob = prob;  
    }
```

```
;;; eFCM.ELink.setValue()  
;;;
```

```
;;; Purpose:      This method sets the value/multiplier of the Link object.
```

```
;;; Returns:      None
```

```
;;; Arguments:    Double, representing the value/multiplier of the Link object.
```

```
;;; Error Handling: None
```

```
;;; Calls:        None
```

```
;;; Called By:    None
```

```
    public void setValue(double value) {
```

```
        this.value = value;  
    }
```

```
;;; eFCM.ELink.getValue()  
;;;
```

```
;;; Purpose:      This method returns the value/multiplier of the Link object.
```

```
;;; Returns:      Double representing the value/multiplier of the Link object.
```

```
;;; Arguments:    None
```

```
;;; Error Handling: None
```

```
;;; Calls:        None
```

```
;;; Called By:    eFCM.EFCM.process()
```

```
    public double getValue() {
```

```
        return value;  
    }
```

```

;;; eFCM.ELink.setPositive()
;;;
;;; Purpose:      This method sets whether the Link object is excitatory or not.
;;; Returns:      None
;;; Arguments:    Boolean on whether the Link object is excitatory or not.
;;; Error Handling: None
;;; Calls:        None
;;; Called By:    None
    public void setPositive(boolean positive) {
        this.positive = positive;
    }

;;; eFCM.ELink.isPositive()
;;;
;;; Purpose:      This method returns a boolean representing whether the Link object is excitatory or not.
;;; Returns:      Boolean representing whether the Link object is excitatory or not.
;;; Arguments:    None
;;; Error Handling: None
;;; Calls:        None
;;; Called By:    eFCM.EFCM.process()
    public boolean isPositive() {
        return positive;
    }

;;; eFCM.ELink.setProb()
;;;
;;; Purpose:      This method sets the probability value for the Link object
;;; Returns:      None
;;; Arguments:    Double representing the probability value for the Link object.
;;; Error Handling: None
;;; Calls:        None
;;; Called By:    None
    public void setProb(double prob) {
        this.prob = prob;
    }

;;; eFCM.ELink.getProb()
;;;
;;; Purpose:      This method returns the probability value of the Link object.
;;; Returns:      Double representing the probability value of the Link object.
;;; Arguments:    None
;;; Error Handling: None
;;; Calls:        None
;;; Called By:    eFCM.EFCM.process()
    public double getProb() {
        return prob;
    }

;;; eFCM.ELink.setFromNode()
;;;
;;; Purpose:      This method sets the originating Node for the Link object.
;;; Returns:      None
;;; Arguments:    Node from where the Link object originates from.
;;; Error Handling: None
;;; Calls:        None
;;; Called By:    None
    public void setFromNode(ENode fromNode) {

```

```

        this.fromNode = fromNode;
    }

;;; eFCM.ELink.getFromNode()
;;;
;;; Purpose:          This method returns the Node that the Link object originate from.
;;; Returns:         Node from where the Link object originates from.
;;; Arguments:       None
;;; Error Handling:  None
;;; Calls:           None
;;; Called By:       None
    public ENode getFromNode () {
        return fromNode;
    }

;;; eFCM.ELink.setToNode()
;;;
;;; Purpose:          This method sets the Node where the Link object terminates to.
;;; Returns:         None
;;; Arguments:       Node where the Link object terminates to.None
;;; Error Handling:  None
;;; Calls:           None
;;; Called By:       None
    public void setToNode(ENode toNode) {
        this.toNode = toNode;
    }

;;; eFCM.ELink.getToNode()
;;;
;;; Purpose:          This method returns the Node where the Link object terminates to.
;;; Returns:         Node where the Link object terminates to.
;;; Arguments:       None
;;; Error Handling:  None
;;; Calls:           None
;;; Called By:       None
    public ENode getToNode() {
        return toNode;
    }

}

```