DOCUMENTATION
CEGA: A Cytoscape Layout Plugin using a Memetic Algorithm

R. Barth & S. Lagarde & S. Rekké

July 2, 2010

1 Introduction

Cytoscape is an open source bioinformatics software platform for visualizing molecular interaction networks and biological pathways and integrating these networks with annotations, gene expression profiles and other state data. Although Cytoscape was originally designed for biological research, now it is a general platform for complex network analysis and visualization. Cytoscape core distribution provides a basic set of features for data integration and visualization. Additional features are available as plugins. A specific set of plugins are especially designed for visualizing graphs.

In this documentation, we describe the CEGA Layout plugin for Cytoscape. CEGA uses a previously proven approach of using memetic algorithms for graph visualization. We aim to provide a guide to the software for both users and developers. We show how the user can adjust the importance of fitness measures by scripting their own fitness function and we show how developers can easily add additional fitness measures.

2 User Manual

Working with the evolutionary Cytoscape Layout plugin is quite easy. In this section we shall guide you through the general use.

2.1 Installation Instructions

To install the plugin for Cytoscape, be sure you first have Cytoscape installed. Go to the Cytoscape directory, and add the following files into the Plugins folder: Please restart Cytoscape if open.

* `CEGA.jar (The actual plugin.)`
* `giny.jar`
* `jgap.jar`
* `colt.jar`

* jung-api-2.0.1.jar
* jung-algorithms-2.0.1.jar
* jung-graph-impl-2.0.1.jar
* ocutil-2.5.2.jar
* asm-all-3.3.jar

2.2 Usage Instructions

Before any layout algorithm can be applied, a network must be loaded. This can be done by going to the menu and click *File -> Import*

When a network is loaded, it is first visualized as a rectangular grid of nodes. To better visualize networks, a Layout plugin can be used. Before applying such Layout algorithm, first certain settings have to be set. To do this go the menu *Layout -> Settings*. After this a window pops up, where you can select an algorithm to view the settings from a dropdown menu. Choose *Memetic Algorithm* The following menu should now be seen:

In this settings menu, three subgroups can be distinguished:

- Evolutionary properties
- Fitness function properties
- Fitness measure properties

Evolutionary properties In this group the general evolutionary parameters can be specified. The population size is defined as the number of individuals of the evolutionary algorithm. It is recommended due to computational complexity to keep this value medium (100) for small networks (150 nodes) and low (30) for large networks (150+ nodes). The larger the number of individuals the more variability in the population exists, possibly improving searching performance.

The number of generations defines how many times the algorithm optimizes the search for a optimal graph. The search time linearly increases with the value defined here. In general and dependent on the complexity of the fitness function defined later, 100 generations takes about 5-10 minutes to complete.

The mutation rate influences the degree of spontaneous changes of any solution graph. This helps the search by preserving and introducing diversity of solutions. A mutation rate of 20% is recommended as default.

The tick box for using Local Search, sets the evolutionary algorithm to memetic if set true. This means that local search is added to find optimal solution. In general, this is computationally more expensive, however should reduce the overall time needed to compute an optimal graph. The setting True is recommended.

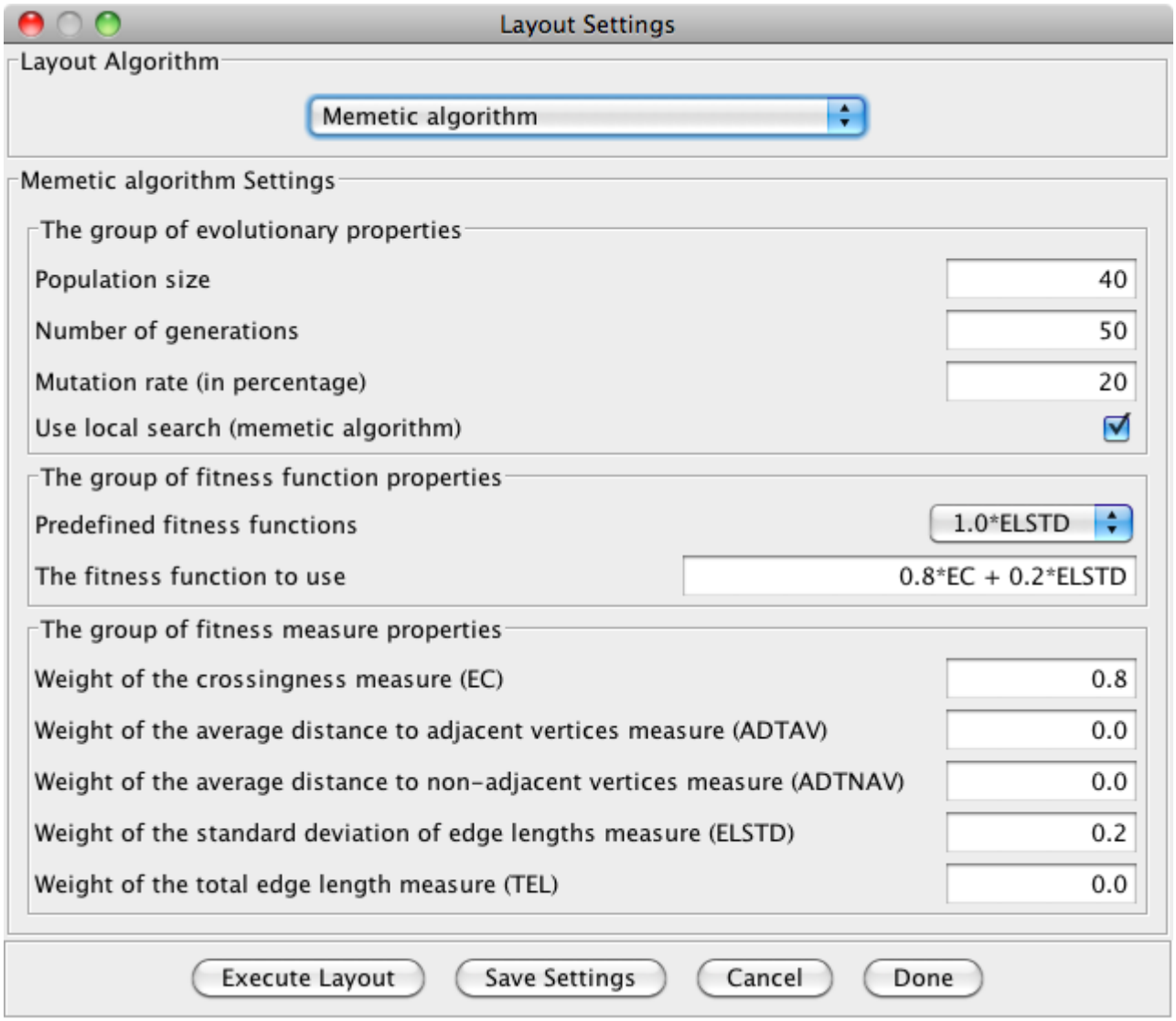

Figure 1: Settings menu for the memetic layout algorithm

Fitness function properties Here a fitness function can be defined. You can either select a predefined fitness function from the list, or create your own.

Creating a fitness function can be done by combining different fitness measures. Combining can be done by addition, subtraction, multiplication and dividing by other measures, or constants. When more complex operators are required, they can be selected from the following list.

```
Math.abs(a)     // the absolute value of a
Math.acos(a)    // arc cosine of a
Math.asin(a)    // arc sine of a
Math.atan(a)    // arc tangent of a
Math.atan2(a,b) // arc tangent of a/b
Math.ceil(a)    // integer closest to a and not less than a
```

```
Math.cos(a)     // cosine of a
Math.exp(a)     // exponent of a
Math.floor(a)   // integer closest to and not greater than a
Math.log(a)     // log of a base e
Math.max(a,b)   // the maximum of a and b
Math.min(a,b)   // the minimum of a and b
Math.pow(a,b)   // a to the power b
Math.random()   // pseudorandom number in the range 0 to 1
Math.round(a)   // integer closest to a
Math.sin(a)     // sine of a
Math.sqrt(a)    // square root of a
Math.tan(a)     // tangent of a
```

If you want to define your own function doSomething(), it is recommended to implement it as a separate fitness measure. How this can be done is described in Section 4.

An example of a combination of measures can be found in the predefined list. For example:

```
1000 + (1.0*ADTNAV / 1.0*ADTAV) - 5.0*EC - 1.0*ELSTD - 1.0*TEL
```

It is important to note that the first constant has the goal that the fitness value cannot drop below zero (a JGAP limitation). Hence subtracting a large result from any measure could result in a non working algorithm. Measures in the function above all are defined as capital abbreviations. Each such a measure returns a value, based on graph properties.

When a fitness function is typed in, any constant value before the measure will be updated as weight in the field of fitness measure properties.

Fitness measure properties In this field all the measures are listed with the corresponding weights as defined in the fitness function. Furthermore, these values can be adjusted to the field adjacent to the measures. If so, this value is automatically updated in the fitness function above.

2.2.1 Saving and Running

When all settings are specified, the optimization can be run. Before running, it is recommended to save the settings by clicking on the respective button. Running can be done by clicking on 'Execute Layout'. Note that some simulations might take a long time to compute. However, when you cancel the simulation, the current solution of the graph is saved and showed.

3 Technical details

For users interested in the technical details, we now further specify how the components of the plugin are organized. In the Figure 2 the general system architecture is displayed. The layout plugin is primarily attached to JGAP, which handles to evolutionary processes. Jung is a graph visualization package, which is used here to solely to add the memetic part to the algorithm.

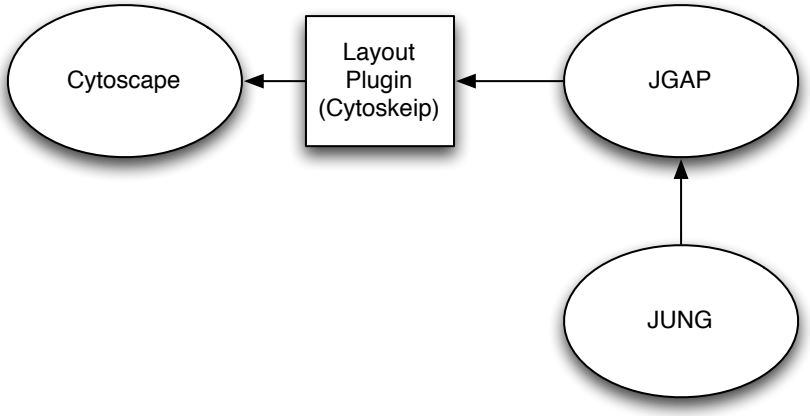

Figure 2: Plugin Architecture

3.1 JGAP

The evolution of graphs is handled by JGAP. JGAP is a Java framework that provides basic genetic mechanisms that can be easily used to apply evolutionary principles to problem solutions.

3.2 JUNG

The Java Universal Network/Graph Framework (JUNG) is a software library that provides a common and extendible language for the modeling, analysis, and visualization of data that can be represented as a graph or network. Furthermore, it offers means to visualize the graph. JUNG was built using the Java programming language, thus allowing us to directly integrate it into our evolutionary algorithm. For this Cytoscape plugin, it is solely used for the local (Spring-based) layout optimization.

3.3 Measures

The fitness function can be created from a weighted composition of different measures. Each measure describes a property of a graphical network. In this section we shall further highlight each measure. The measures are based on previous research on properties influencing the aesthetic visualizations of graphs [2] [1].

3.3.1 Edge Crossings

This measure is defined as the number of crossing edges divided by the number of edges * (number of edges - 1). It is thought that the lower the number of crossing edges in a visualized graph is, the less chaotic and thus the more conveniently arranged the graph is represented. This should increase the usability of the graph. For some graphs, such as the fully connected graph, edge crossingness should be inevitable.

3.3.2 Total Edge Length

This measure is defined as the total length of the edges divided by the total number of edges. When the total length of edges is large this might indicate that the network is not yet optimally arranged. When neighboring nodes are situated far apart, the length of the edge connecting them is large as well. When neighboring nodes are near each other the length of the edge connecting them is small. In general, a smaller total edge length results in better visual graphs due to reduced visual complexity. On the other hand, you might be more interested in graphs that are less dense. You can adjust this measure to reflect your preference.

3.3.3 Edge Length Standard Deviation

When a graph is desired with edges with more or less the same length, a low standard deviation can be used as a measure. If edges are allowed to 'stretch' more a higher standard deviation can be used.

3.3.4 Average Distance to Adjacent Vertices

This measure returns the average distance of nodes which are connected with each other with an edge. This can be used in the fitness function to group nodes which are neighboring.

3.3.5 Average Distance to Non-Adjacent Vertices

This measure returns the average distance of nodes which are not connected with each other with an edge. This can be used in the fitness function in combination with the previous measure in order to promote clustering of nodes.

4 Developer Manual

This section describes how developers can extend the CEGA Layout plugin by developing fitness measures and adding them to CEGA's scripting engine. In future versions of this manual, we will also describe how to take part in development of the CEGA framework. In order to implement your own FitnessMeasure for use in CEGA's fitness functions, you need to implement the FitnessMeasure interface. Listing 1 shows an example implementation of a FitnessMeasure. Once you have implemented such a FitnessMeasure, you need to make it accessible by CEGA and Cytoscape. CEGA uses utilities from the *ocutils* and *asm* libraries to find all classes that implement a FitnessMeasure and adds them to the ScriptEngine. You can make your code 'detectable' in several different ways:

- Add the .class file to the "plugins" folder
- Add the .class file to the "plugins/measures" folder (create the measures folder if it does not exist)
- Build a .jar file and rename it to "plugins/Measures.jar"
- Add the .class file to the classpath
- If you know what you're doing, you could also add it to plugins/CEGA.jar

Listing 1: Multi-Page Java Code

```
1  package net.sf.cega.examples;

   import cytoscape.CyNetwork;
   import java.awt.geom.Point2D;
   import java.util.Map;
6  import net.sf.cega.fitness.FitnessMeasure;

   /**
    * An Example FitnessMeasure.
    * @author The CEGA project
11  */
   public class ExampleMeasure implements FitnessMeasure {

       /**
        * This performs the actual computation of your measure.
16      * This example shows how to use the network and coordinates.
        * @param network This contains only the graph representation,
        * not the visualization.
        * @param coordinates These are the coordinates of the nodes
        * in a Map from node index (Integer) to Point2D.Double
21      * @return the computation result of the measure.
        */
       public double computeMeasure(CyNetwork network,
                           Map<Integer, Point2D.Double> coordinates) {
           double totalAvg = 0.0;
```

```
26          int[] nodes = network.getNodeIndicesArray();
            for (int node1 : nodes) {
                int nNonAdjacent = 0;
                double dist_sub = 0.0;
                for (int node2 : nodes) {
31                  if (!network.edgeExists(node1, node2)) {
                        nNonAdjacent++;
                        dist_sub += (coordinates.get(node1))
                                    .distance(coordinates.get(node2));
                    }
36              }
                if (nNonAdjacent != 0) {
                    totalAvg += dist_sub / nNonAdjacent;
                }
            }
41          if (nodes.length != 0) {
                return totalAvg / nodes.length;
            } else {
                return 0.0;
            }
46      }

        /**
         * Specify the full description of your new FitnessMeasure here.
         * @return The full description of your measure.
51       */
        public String getDescription() {
            return "Type the full description of your new measure.";
        }

56      /**
         * This is where you specify what name is used to refer to
         * your new measure in the fitness function. Use a short
         * measure name without special characters or spaces.
         * @return The name of your fitness measure.
61       */
        public String getName() {
            return "EM";
        }

66      /**
         * Here you specify a short description of your measure.
         * This is used in the Layout Settings GUI of Cytoscape.
         * @return A short description of your measure.
         */
71      public String getShortDescription() {
            return "Testing measure";
        }
}
```

References

[1] P. Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.

[2] H. Purchase, R. Cohen, and M. James. Validating graph drawing aesthetics. *Lecture Notes in Computer Science*, 1027:435–446, 1998.