9-23-2014

Development of an integrated network visualization and graph analysis tool for biological networks

David Carbonetta

DEVELOPMENT OF AN INTEGRATED NETWORK VISUALIZATION AND GRAPH ANALYSIS TOOL FOR BIOLOGICAL NETWORKS

by
David M. Carbonetta

A Thesis

Submitted to the
Department of Electrical & Computer Engineering
College of Engineering

In partial fulfillment of the requirement
For the degree of
Master of Science in Engineering

at
Rowan University
Sep 22, 2014

Thesis Chair: Ying Tang, Ph.D.

Dedication

I'd like to thank my parents for their love and continual support as well as the sacrifices they made to ensure that I received the best education possible. I would also like to thank Dr. Ying Tang and Dr. Sachin Shetty for their intellectual support and technical guidance.

Abstract

David Carbonetta
DEVELOPMENT OF AN INTEGRATED NETWORK VISUALIZATION AND GRAPH ANALYSIS TOOL FOR BIOLOGICAL NETWORKS
2014
Ying Tang, Ph.D.
Master of Science in Electrical & Computer Engineering

There has been steady increase in the amount of molecular data generated by experiments and computational methods performed on biological networks. There is a growing need to obtain an insight into the organization and structure of the massive and complex biological networks formed by the interacting molecules. To that end, this work presents the development of an integrated network visualization and graph analysis plugin within the Cytoscape framework. The plugin is capable of computing and visualizing a comprehensive set of dyad, node, and graph level statistics. The evaluation of the plugin on a range of biological networks and its memory performance is conducted. The plugin, proven to be scalable, is an interactive and highly customizable application that expects no prior knowledge in graph theory from the user.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

There has been steady increase in the amount of molecular data generated by experiments and computational methods performed on biological networks. There is a growing need to obtain an insight into the organization and structure of the massive and complex biological networks formed by the interacting molecules. Given the large size of these networks it is necessary to employ a systematic approach to analyze the raw data in order to better understand their properties. This need is the motivation for undertaking this work. The system presented here is a network analysis plugin, titled *Integrated Network Visualization and Graph Analysis* (INVGA) for the popular network visualization tool Cytoscape. While the theory of complex networks is integral to a variety of disciplines, including power systems engineering, communications, social networking, and molecular biology, the work comprised here focuses on protein-protein interactions in the field of biology. One promising use for this technology is that it could help researchers predict the effects and limit potential side effects

of drugs before they even begin testing on animals or humans, which could potentially save researchers large amounts of time and money.

1.1 Protein- Protein Interactions

Protein-protein interactions are defined as two or more proteins binding together, usually to carry out a biological process. Proteins interact with each other within cells. These interactions are very complex, outside events can affect them by signal molecules, proteins can bind to each and change their function and sometimes proteins can carry other proteins to a different area [1]. The combination of these interactions makes the outcome difficult to predict. Effective techniques are needed to visualize these complex interactions so that we can better understand them. Visualization of these protein interactions will provide an excellent way to analyze these large, complex data sets quickly.

1.2 Cytoscape Plugins

Cytoscape is an open source platform for complex network analysis and visualization. Cytoscape only provides the framework with a basic set of features for data integration, analysis, and visualization. The task of actually employing Cytoscape's extensive network analysis and visualization capabilities is accomplished by 3rd party plugins. INVGA is one such plugin.

1.3 Objective of Thesis

The objective of this thesis is to provide researchers a tool that assists them in gaining better insight into the properties of complex protein-protein interaction networks. The tool titled INVGA implements the fundamental graph analysis algorithms that are essential to understand protein-protein interaction networks but are not provided by either Cytoscape or the Cytoscape plugin development community.

1.4 Scope of Thesis

The graph analysis operations included in INVGA fall into three categories: node, dyad, and network level operations. Node level operations pertain to the properties of a single node including how it relates to the network, dyad level operations analyze the relationship of two nodes by calculating paths that connect them, and finally network level operations calculate properties of the network as a whole. Plugins that perform node and dyad level operation in Cytoscape are nearly nonexistent and therefore are a particular focus of this work. Many of the network level operations provided by INVGA are available in other plugins, they are provided by INVGA to provide researchers with a

complete set of tools that have a common, user-friendly interface.

Chapter 2

Background

Recent advancement in genomic technologies results in large complex networks with an ever-increasing amount of interacting biological data. The complexity of biological processes and the wealth of data needed for the proper consideration of underlying interactions make data visualization and analysis a fundamental prerequisite for the exploration and interpretation of biological data [2]. For instance, an easy visualization of the highly complex linkage of proteins is of significant importance to large-scale cross-species comparisons of proteins and genes [3]. Such visualization often represents the interactions of protein complexes and functional modules as a graph of nodes (proteins) and edges (interactions between proteins). Graph-based methods and tools are then designed to interpret experimental results and gain insight into key controllers of biochemical pathways and complex networks.

In this chapter we discuss some existing plugins related to the functions of INVGA. The Cytoscape plugin development community has fulfilled many needs of researchers in many fields. There have been many plugins developed

for clustering and other functions for instance. However, INVGA fills a specific need of node-level and dyad-level analysis. Dyad-level functions are of particular uniqueness to this work. Other than one plugin for finding shortest path, there are no path-finding analysis tools for node pairs available for Cytoscape. INVGA also provides many network-level analysis functions, some of which have been previously provided by other plugins. The reason for INVGA to include these functions is to provide researchers with one comprehensive plugin for all of their needs concerning protein-protein interactions. While there are no other plugins which do exactly what INVGA does, there are a few plugins that share a small number of functionalities or offer some similar functions. We go into depth about these plugins, what they offer, and any similarities they share with INVGA in the next section.

2.1 ShortestPath Plugin

ShortestPath Plugin [4] is the only plugin available for Cytoscape that features a dyad operation. It only performs a single operation, which is to find the shortest path between two nodes. This plugin works on both directed and undirected graphs. The user selects two nodes and executes the plugin, which highlights the shortest path between the selected nodes and returns the path length in a pop-up

box. This plugin was the inspiration for the dyad operations of INVGA. It features a very simple visual style that allows the user to easily visualize a specific path in a complex and cluttered network. INVGA's shortest path operation works very similarly to ShortestPath Plugin, however it also visually distinguishes the source and sink nodes from the rest of the path. INVGA applies this format to a variety of other dyad operations and provides a variety of numerical information appropriate to each operation in a pop-up box.

2.2 CytoHUBBA

CytoHUBBA [5] explores important nodes/hubs and fragile motifs in an interactome network by several topological algorithms including Degree, Edge Percolated Component (EPC), Maximum Neighborhood Component (MNC), Density of Maximum Neighborhood Component (DMNC), Maximal Clique Centrality (MCC) and centralities based on shortest paths, such as Bottleneck (BN), EcCentricity, Closeness, Radiality, Betweenness, and Stress. It ranks nodes based on their importance to the operation executed. The amount of nodes included in the ranking is set by the user. Comparing INVGA with CytoHUBBA, there are two types of function the two plugins share with similarities and

differences. CytoHUBBA features several methods of calculating the centrality of a network, including closeness and betweenness; while the centrality that INVGA calculates is degree centrality. CytoHUBBA also offer an operation called "Degree"; that ranks nodes from highest to lowest degree; while INVGA features a related operation called "Degree Distribution" that returns what percentage of nodes in the network has a specific degree.

2.3 CentiScaPe

CentiScaPe [6] was developed in 2009 by Giovanni Scardoni, Michele Petterlini, and Carlo Laudanna. The plugin computes a variety of centrality parameters to assist users in finding the most significant nodes in a complex network. CentiScaPe employs both numerical and graphical output in an effort to make the results as clear as possible. The plugin can calculate centricity, closeness, betweenness, stress, centroid, and radiality. The shared functionality between CentiScaPe and INVGA is calculating centrality (albeit by applying different methods), graph diameter and mean shortest path.

2.4 MCODE

Clusters are highly interconnected groups of nodes in a network. In a protein-protein interaction network, these clusters usually represent protein complexes or parts of pathways. MCODE [7] finds these clusters, displays the results visually and ranks the resulting clusters. MCODE was developed by Bader GD and Hogue CWV in 2003. It is one of the oldest and most widely used clustering plugins available for Cytoscape. There are actually a large number of clustering plugins available for Cytoscape. It is for this reason that a new clustering algorithm is not developed for INVGA, instead INVGA employs MCODE for its clustering calculations.

2.5 Network Analyzer

Network Analyzer [8] was developed in 2008 by Y. Assenov, et al., which was the most expansive tool for graph analysis available for Cytoscape. Although it has many analytical features, it was developed for network-level analysis and offers very limited node-level analysis and zero dyad-level functionality.

In summary, the development of INVGA bridges the aforementioned functionality gaps in biological network analysis. The successful implementation

of INVGA offers researchers a comprehensive package for all basic protein-protein network analysis. In particular, INVGA includes many of the same network-level analysis tools that Network Analyzer offers. This common overlap includes functions such as: mean shortest path, degree distribution, average number of interaction partners, topological coefficient, diameter of graph, and degree centrality. The way in which INVGA distinguishes itself from Network Analyzer is in its node-level and dyad-level functionality.

Chapter 3

Design and Implementation

INVGA is a Cytoscape plugin that efficiently computes graph-, node- and dyad-level topological operations for undirected networks loaded into Cytoscape from a variety of file formats which are supported by Cytoscape. INVGA is written in JAVA and utilizes Cytoscape's API and their corresponding libraries. The node- and dyad- level operations that INVGA is capable of calculating are all paths, shortest path, critical path, max flow, and topological coefficient. The network operations it offers include mean shortest path, degree distribution, average number of interaction partners, clustering, diameter of graph, and degree centrality. Given that those topological operations are frequently used in the literature, the work presented here focuses their design and integration with Cytoscape as well as their biological relevance. In particular we examine the implementation of every operation of which INVGA is capable, including the Java code where relevant.

3.1 Search Algorithms

All of the path calculating operations, which include all of the dyad operations and several network operations, employ either depth first search (DFS) or

breadth first search (BFS). For clarity, the two algorithms (DFS/BFS) are first briefly described with the focus on their implementation in INVGA.

3.1.1 Depth First Search. Depth first search (DFS) is a recursive algorithm that is used to find a path between two nodes in a graph or tree structure data set. DFS is categorized as an uninformed search. In a tree structure data set, the algorithm starts at the root and progresses through the branches. If the data set is represented as a graph, the algorithm starts at a specified node which is referred to as the "source" node. This latter case is the focus of this work. The DFS algorithm starts at the source and ends when it reaches what is known as the "sink" node. The algorithm examines the first node directly connected to the source node. All nodes directly connected to the node currently being examined are known as the neighbor nodes or child nodes. If this node is not the sink node, the algorithm progresses to that node's first neighbor node. This continues until the sink node is found or

a node is reached which has no neighbors. If the latter is the case, the algorithm backtracks to the previous node and examines its next neighbor node. This process repeats until the sink node is reached.

For the implementation, INVGA utilizes Boolean arrays and stacks wherever possible to maximize efficiency. This is an important consideration given that networks can be extremely large. The Java code for the implementation of DFS is presented below. In it you can see that the variable *u* is the node currently being examined. The algorithm starts by setting the user selected source node as *u*. The Boolean array *inAllPaths* holds the current path being built. The stack *path* holds the nodes currently being examined. As nodes are found to be in the path, they are popped from the stack and recorded as *true* in *inAllPaths*. When a node is recorded in *inAllPaths*, it is marked as visited. Once a path is completed it is added to the arraylist *pathMatrix2*, where all of the resulting paths can be later examined. This process is repeated until all the nodes in the network are marked as visited and every path is found. The matrix that results from the combination of the arrays is ordered in such a way that the IDs of the nodes in each path can be calculated from their placement in the matrix, negating the need to call and store every node ID. With this method of implementation, only the IDs of the nodes of interest are called at the necessary time. Below is the presentation of the implementation.

```
private boolean DFS(int u){
        if (visited[u] == true){
           return true;
        }
           if (u == sink){
              for (int i = 0; i < size; i++){
                  inAllPaths[i] = true;
              }
               for (int i = 0; i < size; i++){
                 if (!path.contains(i)){
                     inAllPaths[i] = false; // inAllPaths contains true only at vertices that exist in all paths from
                    source to sink.
                     }
              }

              inAllPaths[sink] = true;
              pathMatrix2.add(deepCopy(inAllPaths));
              return true;
             }else {
              visited[u] = true;
              path.push(u);
              //foreach edge (u, v)
              for (int k = 0; k < size; k++){
                 int v = capacity[u][k];
                 if (v != 0 && visited[k] != true){
                    DFS(k);
                 }
              }



              path.pop();
              visited[u] = false;
              return false;
                       }
                    }
```

3.1.2 Breadth First Search. Breadth First Search (BFS) is another uninformed search algorithm. BFS begins by adding the source node to the queue if the data set is a graph or in the case of a tree structure data set, the root node. The node is then removed from the queue

and examined. If the node being examined is the sink node the algorithm ends. If not, that node's neighbor nodes are added to the queue and the previous step is repeated until the sink node is found.

INVGA implements BFS by employing the Edmonds-Karp algorithm. Its coding utilizes Boolean and integer arrays exclusively for optimal computational speed. The algorithm results in building a 2 dimensional Boolean array which represents the nodes in the path between the source and sink nodes in the network. The coordinates of all the true values in the flow matrix can be used to calculate the node IDs from which they resulted. Again, this technique negates the need to record additional information in that matrix which would bloat the matrix and reduce computational speed. As long as you have the ID of a node in Cytoscape, you can call any information that you may need as you need it. The implementation of BFS is displayed below.

```
//Breadth First Search
    private boolean BFS(int source){
        for (int i = 0; i < size; i++){
            color[i] = WHITE;
            min_capacity[i] = Integer.MAX_VALUE;
        }

        first = last = 0;
        queue[last++] = source;
        color[source] = GRAY;

        // While "queue" not empty
        while (first != last){
            int v = queue[first++];
            for (int u = 0; u < size; u++)
                if (color[u] == WHITE && res_capacity[v][u] > 0){
                    min_capacity[u] = Math.min(min_capacity[v], res_capacity[v][u]);
                    parent[u] = v;
```

```
                    color[u] = GRAY;
                    if (u == sink) return true;
                    queue[last++] = u;
                }
        }
        return false;
    }
```

3.2 Dyad-Level Operations

3.2.1 All Paths. The All Paths operation is a simple and direct dyad-level operation offered by INVGA. In biological networks, finding all paths connecting two given substances is of importance to identify chains of reactions. Therefore, the All Paths operation is designed in INVGA, where an iterative depth first search algorithm is executed on the connectivity matrix of a loaded network to find all discrete, noncyclical paths between two selected nodes. Once the paths are found, the nodes along the paths are highlighted to assist the user in easily visualizing the paths and their cumulative connectivity. A pop-up window, as shown in Fig. 3.2.1, is also displayed offering numerical values of the resulting number of path calculated as well as their average path length. The following is the Java code for the All Paths algorithm including the initial construction of the matrices. One note of worth, the nodes are referenced by index, however all indices in Cytoscape are negative. Also, Cytoscape indices start a (-)1, while arrays start at 0.

This added complication is what the comment "// Makes index negative and adjusted" is referring to.

```
private boolean[][] getAllPaths(int node1, int node2) {

        indexArray = root.getNodeIndicesArray();
        size = indexArray.length;
        String output = "";
        int colorIndex = 0;
        double sum = 0;
        double NoP = 0;
        double AvLength = 0;
        APSaveData = "";

        parent = new int[size];
        color = new int[size];
        queue = new int[size];
        capacity = new int[size][size];
        int index = 0;
        inAllPaths = new Boolean[size];
        java.util.Arrays.fill(inAllPaths,true); // init to all true
        visited = new boolean[size];
        java.util.Arrays.fill(visited,false); // init to all false
        path = new Stack<Integer>();
        path.clear(); // init empty
        pathMatrix = new boolean[size][size];

        CyNode cynode1 = (CyNode) network.getNode(node1);
        CyNode cynode2 = (CyNode) network.getNode(node2);
        source = -1*root.getIndex(cynode2)-1;
        sink = -1*root.getIndex(cynode1)-1;

        //Build capacity[size][size] matrix
        for (int i = 0; i < size; i++){
         adjArray = network.neighborsArray(-i-1);
         for (int j = 0; j < adjArray.length; j++){
                    index = -1*adjArray[j]-1;
                        capacity[i][index] = 1 ;//1 means connected
           }
        }

        Arrays.fill(inAllPaths,true); // init to all true
        Arrays.fill(visited,false); // init to all false
        path.clear(); // init empty

        DFS(source);

        //Iterate through results and change node colors
        for(int i=0; i<pathMatrix2.size(); i++){
           Boolean[] temp = pathMatrix2.get(i);
```

```
    ArrayList<String> idArray = new ArrayList<String>();
    String NodeID ="";
    output += "[";
    for(int j=0; j<temp.length; j++){
       if(temp[j] == true){
          colorIndex = -1*j - 1; // Makes index negative and adjusted
          Node node = root.getNode(colorIndex); //Get node by index
          NodeView nodeView = view.getNodeView(node);
          nodeView.setUnselectedPaint(Color.BLUE);
          sum += 1;
          //Fill idArray
          Node node3 = root.getNode(colorIndex);
          NodeID = node3.getIdentifier();
          idArray.add(NodeID);
    }else{}
    }
  //Send idArray to string of Save Data
    for(int k=0; k<idArray.size(); k++){
       output += " " + idArray.get(k);
    }
    output += "] ";
  idArray.clear();
  }
```

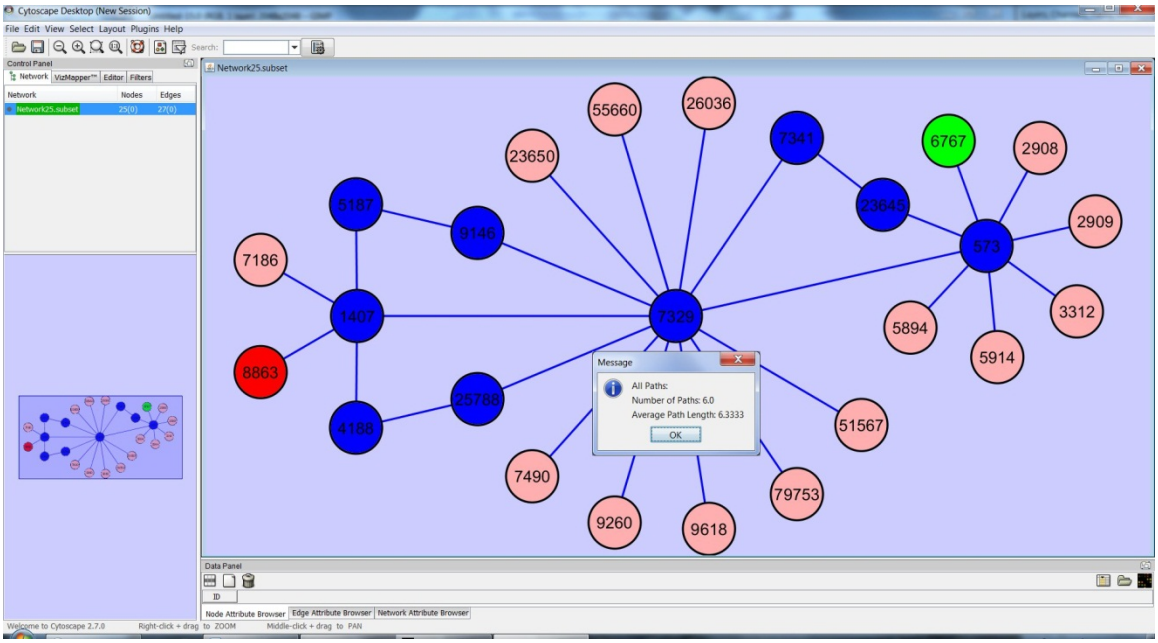

Fig. 3.2.1 - All Paths

3.2.2 Shortest Path. This Shortest Path operation plays a critical role in analyzing biological networks. For instance, the shortest path between a receptor and a DNA binding protein allows the prediction of the signal transduction pathway from a protein-protein interaction network. Similarly, it helps the prediction of a metabolic pathway given two reactions or compounds of interest and a metabolic network. The shortest path very likely traverses the hub nodes of a network. It depends on the biological context, whether this behavior is desired or not [9, 10].

Shortest Path is another dyad operation which uses DFS to find the desired path. The Shortest Path operation essentially runs the All Paths algorithm and then examines all of the resulting paths by comparing their lengths. Path length is defined as the number of edges which must be traveled to connect two nodes. In the case, the path that connected the user specified source and sink nodes that has the fewest number of edges is desired. In protein-protein interaction networks, the "length" of an edge does not apply as opposed to a road map. All that matters is if they are connected, therefore this algorithm considers all edges to have a weight of 1. The resulting shortest path is then highlighted in the network and the length of the path is displayed numerically in a pop-up box for the user as shown in Fig. 3.2.2. Some of the implementation of

the Java code for Shortest Path is presented here after omitting some of the redundancy with the All Paths algorithm.

```
public int getShortestPath(int node1, int node2) {
        …

        …
        DFS(source);

        shortest = pathMatrix2.get(0).length;
        sTemp = pathMatrix2.get(0).length;
        for(int i=0; i<pathMatrix2.size(); i++){
           Boolean[] temp = pathMatrix2.get(i);
           sTemp = 0;
           for(int j=0; j<temp.length; j++){
              if(temp[j] == true){
                 sTemp += 1;
              }else{}
           }
           if(sTemp < shortest){
              shortest = sTemp;
              sRow = i;
           }
        }

        // Iterate through results and change node colors
        Boolean[] temp = pathMatrix2.get(sRow);
           for(int j=0; j<temp.length; j++){
              if(temp[j] == true){
                 colorIndex = -1*j - 1; // Makes index negative and adjusted
                 Node node = root.getNode(colorIndex); //Get node by index
                 SLength += 1;
           }else{}
              output += " " + temp[j];     //Save Data
           }
        output += "\n";

        pathMatrix2.clear();
        SLength = SLength-1;

        return SLength;
      }
```

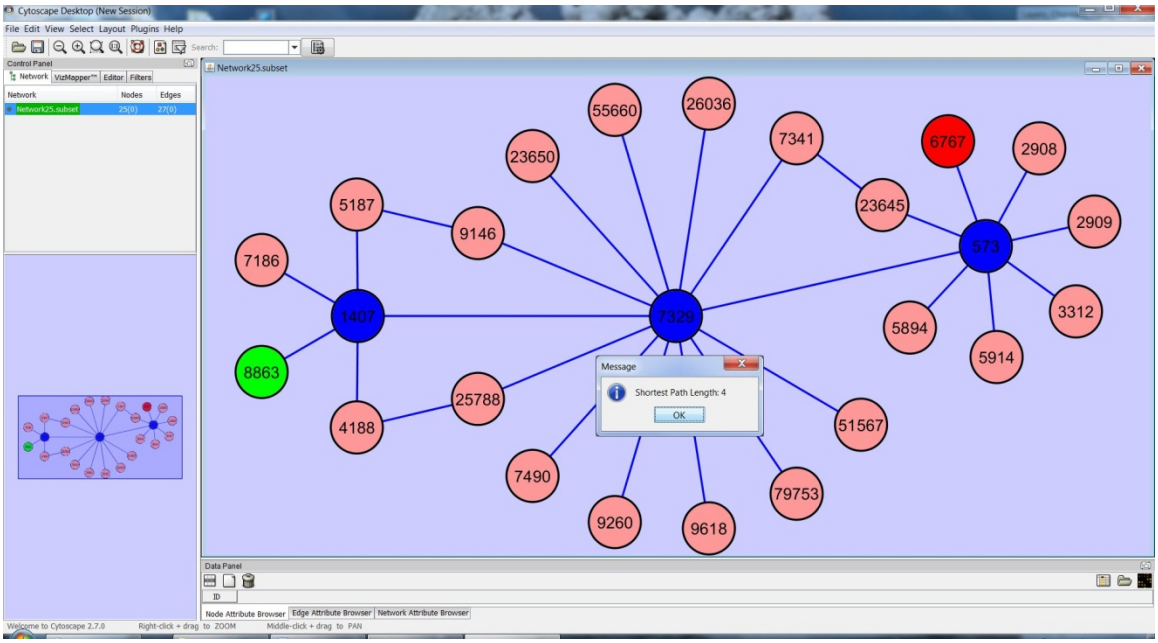

Fig. 3.2.2 - Shortest Path

3.2.3 Critical Path. Critical-path parameter is concerned about all node degrees along a network path. In analyzing protein networks, the critical path length property provides an insight into the number of core proteins and the interacting proteins in various linear network paths. Durmus *et al* have shown that during the analysis of a protein-protein network of insulin signaling in homospaiens, the critical path plays a main role in the glucose transportation response of the signaling network [11]. To satisfy this analytical need, INVGA provides the critical-path computation for two nodes being selected by a user.

Critical Path is a dyad operation offered by INVGA which utilizes DFS. The goal of Critical Path is to find the mostly highly connected path between any pair of nodes. The Critical Path algorithm runs the All Paths algorithm to return all of the unique, noncyclical paths between the selected source and sink nodes. Similarly to the Shortest Path algorithm, after Critical Path runs All Paths, it then examines the resulting paths by a specific set of criteria. The criterion used in our implementation is the degree of a node defined as the number of incident edges of a node. The algorithm sums the degrees of every node along each path and returns the path with the highest summation. The resulting path is highlighted in the network so that it can be easily visualized by the user and the values of the summation and path length are displayed numerically in a pop-up box as shown in Fig. 3.2.3.

```
public boolean[][] getAllPaths(int node1, int node2) {
        …

        …
        DFS(source);

        int tempCrit = 0;
        int maxCrit = 0;
        int row = 0;
        int tempLength = 0;
        //Calculate critical path from all paths
        for(int i=0; i<pathMatrix2.size(); i++){
           Boolean[] temp = pathMatrix2.get(i);
           tempCrit = 0;
           for(int j=0; j<temp.length; j++){
              if(temp[j] == true){
                 indexCrit = -1*j - 1;
                 tempCrit += network.getDegree(indexCrit);
                 tempLength += 1;
```

```
                if(tempCrit > maxCrit){
                   maxCrit = tempCrit;
                   critLength = tempLength-1;
                   row = i;
                }
             }else{}
          }
       }
       //Color nodes in critical path
       Boolean[] temp = pathMatrix2.get(row);
       ArrayList<String> idArray = new ArrayList<String>();
       String NodeID ="";
       for(int k=0; k<temp.length; k++){
            if(temp[k] == true){
               colorIndex = -1*k - 1; // Makes index negative and adjusted
               Node node = root.getNode(colorIndex); //Get node by index
               NodeView nodeView = view.getNodeView(node);
               nodeView.setUnselectedPaint(Color.BLUE);
               //Fill idArray
               Node node3 = root.getNode(colorIndex);
               NodeID = node3.getIdentifier();
               idArray.add(NodeID);
            }else{}
          }
       //Send idArray to string
       for(int k=0; k<idArray.size(); k++){
          output += " " + idArray.get(k);
       }

       //Change color of source and sink nodes
       NodeView nodeView1 = view.getNodeView(cynode1); //Source = "Go"
       nodeView1.setUnselectedPaint(Color.GREEN);
       String sourceNodeID = cynode1.getIdentifier();
       NodeView nodeView2 = view.getNodeView(cynode2); //Sink = "Stop"
       nodeView2.setUnselectedPaint(Color.RED);
       String sinkNodeID = cynode2.getIdentifier();

       JOptionPane.showMessageDialog(Cytoscape.getDesktop(), "Critical Path Value: " + maxCrit +
"\n\nCritical Path Length: " + critLength);
       pathMatrix2.clear();

       //Save Data
       CPSaveData = "Critical Path("+sourceNodeID+", "+sinkNodeID+"): "
            +"["+ output +"]"+ "   Length: " +critLength;
       SaveData SDobj = new SaveData();
       SDobj.WriteFile(CPSaveData);

       return pathMatrix2;
    }
```

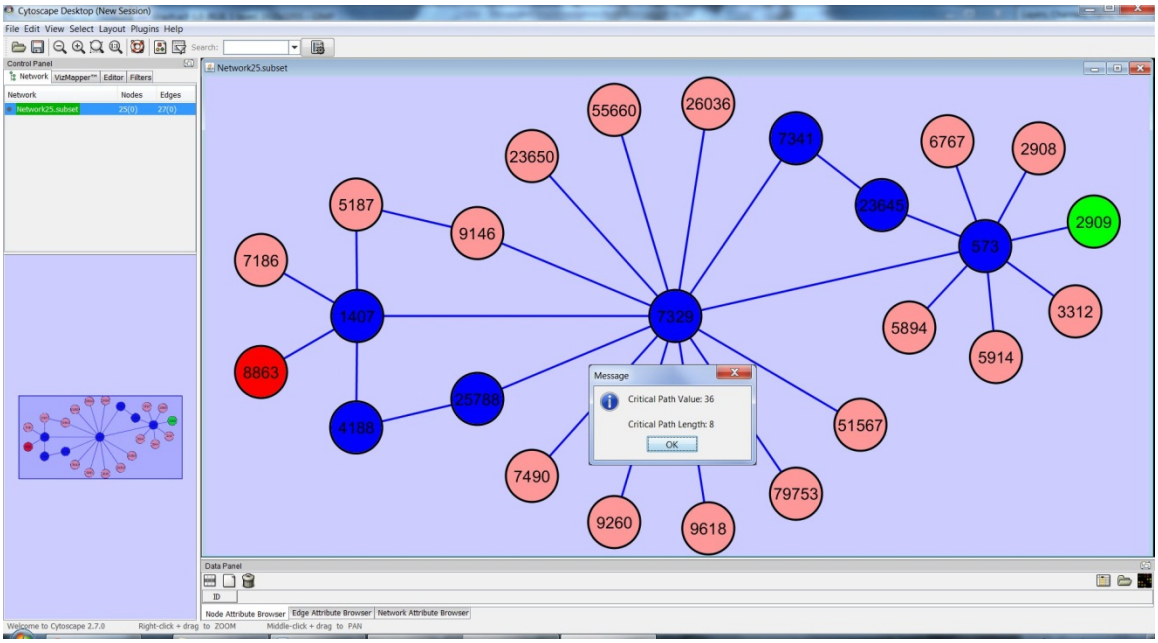

Fig. 3.2.3 - Critical Path

3.2.4 Max Flow. The maximum flow property has been widely used in automatic decomposition of proteins. With the exponential growth in number of protein structures, the decomposition of multi-domain proteins into individual proteins becomes essential. Such a problem is usually formulated in a network flow framework in which residue of a protein is represented as a node of the network and each residue-residue contact as an edge with a particular capacity. Finding the maximum flow of the network can help determine the bottleneck of the network, which is the core of the decomposition [12].

Flow can be visualized as the rate that information is traveling between two nodes. Max flow seeks to find the configuration of paths that result in the highest possible rate of flow. An easily understood analogy would be to think of the nodes in the network to be computers and the edges to be data lines.

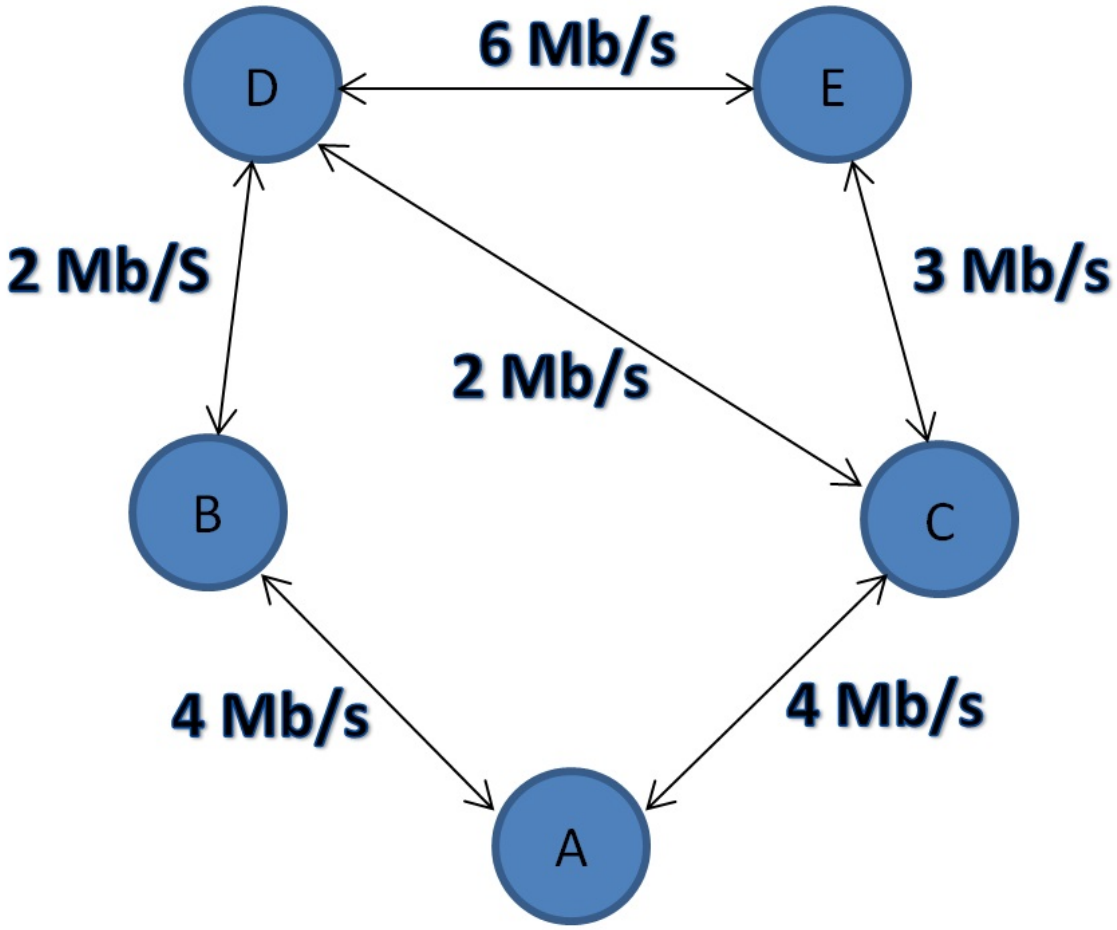

Fig. 3.2.4.1 - Example Network [13]

Given the undirected network presented in Figure 3.2.4.1, suppose that we wanted to send a large file from computer *A* to computer *E*. What route (paths)

should be taken to transfer the file at the maximum rate possible and what is the value of that rate? Two important things to consider are that no node can send more data than it receives and no node will receive more data than it can send. That is to say that all data is conserved. Computer *A* can send data out at a rate of *8 Mb/s* since it had two *4 Mb/s* data lines. Computer *B*'s date line from computer *A* is capable of receiving data at *4 Mb/s*, however it can only send out *2 Mb/s* of data, so *B* will then only receive *2 Mb/s* from *A*. Computer *C* can receive *4 Mb/s* of data from *A* and can send *3 Mb/s* to *E*, but the extra *1 Mb/s* can be sent over its data line to *D*. Computer *D* then receive *2 Mb/s* from *B* and *1 Mb/s* from *C* and can easily send all *3 Mb/s* over its line to *E*. So then Computer *E* receives *3 Mb/s* from *D* and another *3 Mb/s* from *C*. This results in the maximum flow of data from node *A* to node *E* to be *6 Mb/s*.

As clearly shown in the example, there are several important steps to derive the max flow of a given network. The first of these is to find all paths between the source and sink nodes. For that purpose, Breadth First Search (BFS) is utilized. The traditional implementation of BFS is using the queue structure to traverse the network and to determine the connectivity of individual nodes. The source node is first added to the queue and marked as examined (i.e., `color[source] = GRAY`). A loop is then used to check the connectivity of the node

being examined (i.e., node *v*) with all of the other nodes (i.e., node *u*) in the network. Only if u has not been visited (i.e., `color[u] == WHITE`) and it is connected with *v* (i.e., `res_capacity[v][u] > 0`), u is pushed into the queue for further examination, and changed its color to gray. Meantime, *v* is marked as *u* 's parent. This process is continued until the queue is empty.

The second important aspect is the *residual capacity* matrix as we are concerned not only the connectivity of two nodes but also the flow in between. The *residual capacity* matrix is initially set to be the *capacity* matrix that contains all of the nodes in the network with their corresponding weighted edges. BFS operates on the *residual capacity* matrix. Each time BFS finds a path between the source and the sink, the *residual capacity* matrix is updated to reflect the portion of the network capacity that remains. This allows the same BFS to run repeatedly for unique paths until the *residual capacity* reaches zero.

The final important aspect of Max Flow is the concept of *minimum capacity* that is implemented as a necessary check to ensure the conservation of flow in the network. The flow that *v* is sending to *u* over their shared edge can't be more than the maximum value of flow available to *v* as illustrated in the equation below.

```
min_capacity[u] = Math.min(min_capacity[v], res_capacity[v][u]
```

Minimum capacities are updated and recorded in the *flow* matrix each time BFS is run. When Max Flow concludes, the *flow* matrix contains the portions of all of the edges which the flow traverses between the source and sink, that is to say it contains all of the resulting paths with their respective flows.

The protein-protein interaction networks of concern here are undirected networks with all edges having a weight of 1. In INVGA's Max Flow operation, all of the paths traveled between the user selected source and sink nodes are highlighted in the network and the value of the flow is displayed numerically as shown in Fig. 3.2.4.2.

```
// Edmonds-Karp algorithm
public int[][] getMaxFlow(int node1, int node2) {
        …

        …
        //Build capacity[size][size] matrix
        for (int i = 0; i < size; i++){
         adjArray = network.neighborsArray(-i-1);
         for (int j = 0; j < adjArray.length; j++){
                    index = -1*adjArray[j]-1;
                    //nodeIndex = -1*adjArray[j];
                        capacity[i][index] = 1 ;//1 means connected
           }
        }

           for (int i = 0; i < size; i++)
                 for (int j = 0; j < size; j++)
                       res_capacity[i][j] = capacity[i][j];

          while (BFS(source)){
                 max_flow += min_capacity[sink];
                 int v = sink, u;
                 while (v != source){
                       u = parent[v];
```

```
                    flow[u][v] += min_capacity[sink];
                    flow[v][u] -= min_capacity[sink];
                    res_capacity[u][v] -= min_capacity[sink];
                    res_capacity[v][u] += min_capacity[sink];
                    v = u;
                }
            }

        //Change Node Color & Print matrix for debugging
        ArrayList<String> idMatrix = new ArrayList<String>();
        String NodeID ="";
        for (int row = 0; row < size; row++) {
        for (int col = 0; col < size; col++) {
          if(flow[row][col] != 0){
            colorIndex = -1*row - 1; // Makes certain index is negative and adjusted
            Node node = root.getNode(colorIndex); //Get node by index
            NodeView nodeView = view.getNodeView(node);
            nodeView.setUnselectedPaint(Color.BLUE);
            //Fill idArray
            Node node3 = root.getNode(colorIndex);
            NodeID = node3.getIdentifier();
            idMatrix.add(NodeID);
          }else{}
        }
      }
        //Send idArray to string
        for(int k=0; k<idMatrix.size(); k++){
          output += " " + idMatrix.get(k);
        }

        //Change color of source and sink nodes
        NodeView nodeView1 = view.getNodeView(cynode1); //Source = "Go"
        nodeView1.setUnselectedPaint(Color.GREEN);
        String sourceNodeID = cynode1.getIdentifier();
        NodeView nodeView2 = view.getNodeView(cynode2); //Sink = "Stop"
        nodeView2.setUnselectedPaint(Color.RED);
        String sinkNodeID = cynode2.getIdentifier();

        JOptionPane.showMessageDialog(Cytoscape.getDesktop(), "Max Flow: " + max_flow);

        //Save Data
        MFSaveData = "Max Flow Path("+sourceNodeID+", "+sinkNodeID+"): "
             +"["+ output +"]"+ "   Flow Value: " +max_flow;
        SaveData SDobj = new SaveData();
        SDobj.WriteFile(MFSaveData);

        return flow;
    }
```

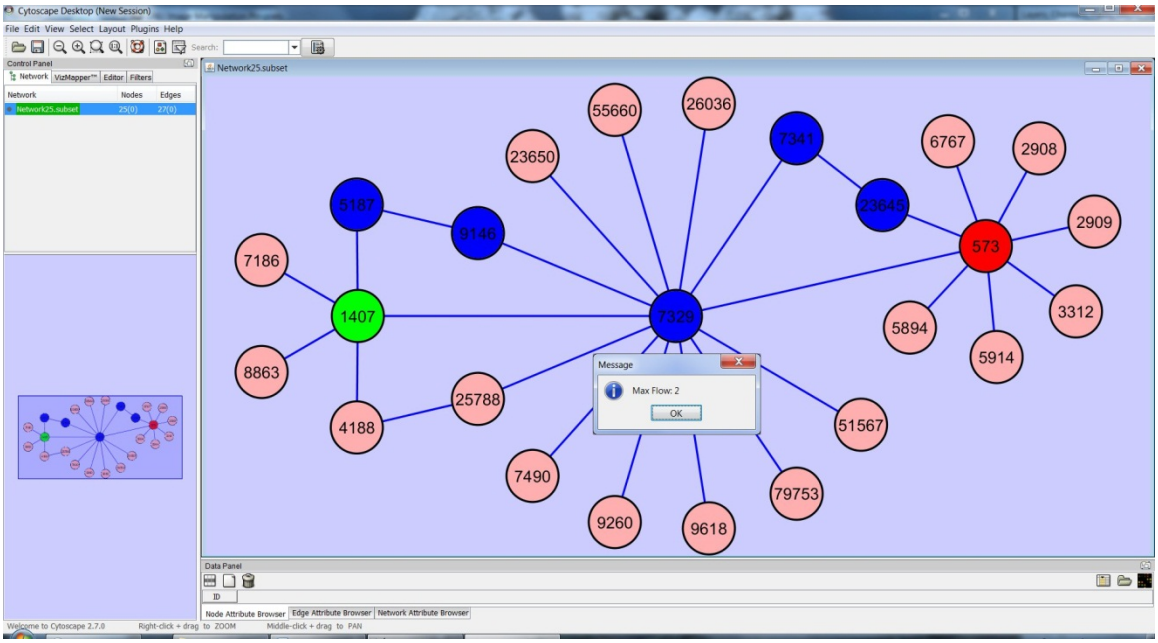

Fig. 3.2.4.2 - Max Flow

3.3 Node-Level Operations

3.3.1 Topological Coefficient. The topological coefficient is a relative measure for the extent to which a node shares neighbors with other nodes. In the context of analyzing proteins, the decreasing behavior of the topological coefficient as the number of interactions of a protein increases provides an indication of the modular network organization. For instance, the decreasing behavior of topological coefficient will indicate that the neighbors of hub proteins are not more connected than the neighbors of sparsely connected proteins [14].

Topological coefficient is a node-level operation that is used to measure the relative amount to which a given node shares neighbors with other nodes. When the Topological Coefficient operation is executed on a selected node, the following equation is calculated:

$$T_n = \frac{avg(J(n,m))}{k_n} \qquad (m=1, 2, \ldots k_n)$$

where *J(n,m)* is the number of nodes shared between the selected node n and *m*, where *m*=1, 2, …k_n, the node that has at least one neighbor shared with node *n* and k_n is the number of neighbors of node *n*. A value of 1 is added when node *m* has a direct link to node *n*. The results for every node *m* is averaged to give *avg(J(n,m))*. Finally, this value is divided by k_n. The resulting value of the topological coefficient (T_n) is displayed in a pop-up box as shown in Fig. 3.3.1.

```
public void TCcalc(){

   network = Cytoscape.getCurrentNetwork();
   view = Cytoscape.getCurrentNetworkView();
   root = Cytoscape.getRootGraph();
   TCSaveData = "";

   int Tt = 0; //topological total
   int counter = 0;
   double Tav = 0; //average topological coefficient
   double T = 0; //topological coefficient of nodeN
   String NodeID = "";

   //Get selected node
   //Node NodeN = nodeView.getNode(); //Get nodeN (selected by user)

   //int NIndex = root.getIndex(NodeN);
```

```
int[] NodeN = view.getSelectedNodeIndices();
int NIndex = NodeN[0];
int[] neighborArrayN  = network.neighborsArray(NIndex); //Get neighbors of N
int kn = neighborArrayN.length; //number of neighbors of nodeN
int[] indexArray = root.getNodeIndicesArray();
List nodeList = root.nodesList();
int NI = 0;
int MI = 0;

if (kn <= 1){
   T = 0;
}
else{
   for (Iterator iter = nodeList.iterator(); iter.hasNext();) {
      CyNode NodeM = (CyNode) iter.next();
      int MIndex = root.getIndex(NodeM);
      int[] neighborArrayM  = network.neighborsArray(MIndex); //Get neighbors of M

      if (MIndex != NIndex){

         //For every node contained in neighborArrayN && neighborArrayM {
         //Add 1 to Tt & add 1 to counter }
         for(int i=0; i<neighborArrayN.length; i++){
            NI = neighborArrayN[i];
            for(int j=0; j<neighborArrayM.length; j++){
               MI = neighborArrayM[j];
               if(MI == NI){
                  Tt += 1;
                  counter += 1;
               }
               //If nodeM is contained in neighborArrayN {
               //Add 1 to Tt }
               if(MIndex == neighborArrayN[i]){
                  Tt += 1;
            }
            }

         }
      }
      else{}
}
}
if(counter != 0){
Tav = Tt / counter;
T = Tav/kn;
}else{
   T = 0;
}

DecimalFormat df = new DecimalFormat("#.#####");
Node node = root.getNode(NIndex);
```

```
    NodeID = node.getIdentifier();

    JOptionPane.showMessageDialog(Cytoscape.getDesktop(), "Topological Coefficient of node " +
NodeID + ":\n" + df.format(T));

    //Save Data
    TCSaveData = "Topological Coefficient("+NodeID+"): " + df.format(T);
    SaveData SDobj = new SaveData();
    SDobj.WriteFile(TCSaveData);
  }
```

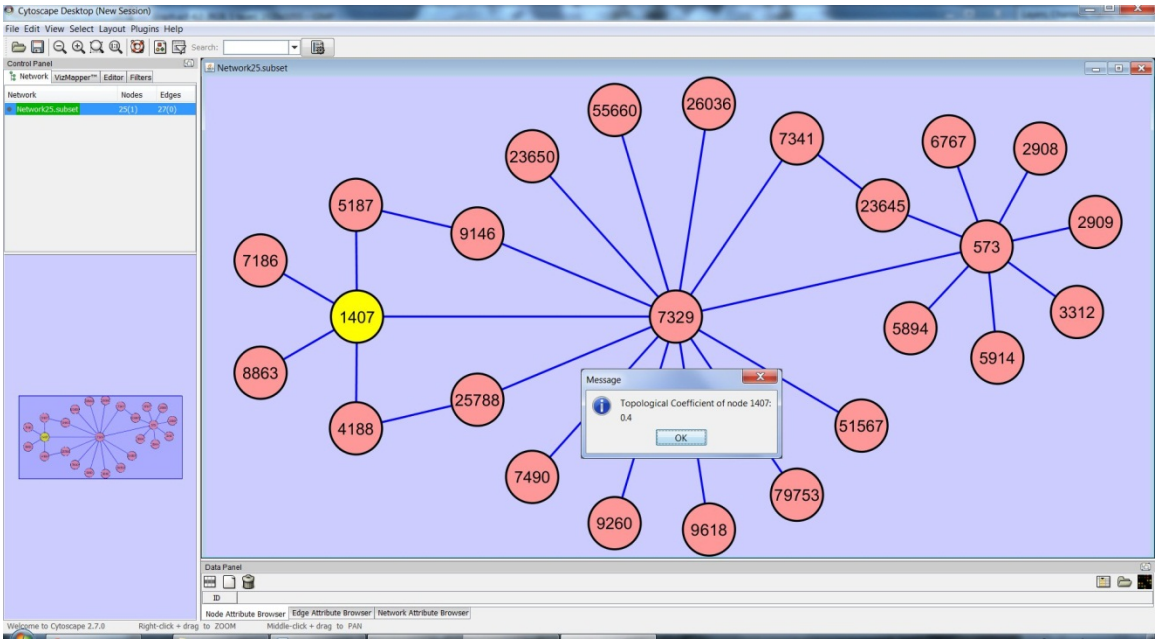

Fig. 3.3.1 - Topological Coefficient

3.4 Graph-Level Operations

3.4.1 Mean Shortest Path. The mean shortest path length property is usually utilized in analyzing connectivity of protein networks. It is calculated by finding the shortest paths

between all protein pairs in a given connected network and then averaging them over all connected components [15].

The Mean Shortest Path (MSP) operation is one of the graph-level path finding operations. It employs a modified version of the Shortest Path algorithm and iterates through every pair of nodes in the network instead of having the user to specify a single source and single sink node. MSP finds the shortest path between every pair of nodes and sums their lengths. This summation is then divided by the number of node pairs found to calculate the average. Combinations of identical source and sink nodes are excluded. The Mean Shortest Path, being a graph-level operation, does not highlight the paths it calculated since it does calculation over the entire network. It does however display a pop-up box with numerical values of the network's mean shortest path length, the number of shortest paths, and the summation of all of the shortest paths found as shown in Fig 3.4.1. Below is the code of Mean Shortest Path which is unique from Shortest Path.

```
//Iterate through all node combinations
        for (int i = 0; i < indexArray.length; i++){
           int node1 = indexArray[i];
           for (int j = 0; j < indexArray.length; j++){
              int node2 = indexArray[j];

              if(node1 == node2){}
              else {
                 int allPaths = getShortestPath(node1,node2);
                 pathCount += 1;
                 output += allPaths + " ";
                 sum += allPaths;
```

```
            }

          }
        }

        //Avoid divide by 0 error
        if(pathCount > 0){
        MeanSP = sum/pathCount; //Calculate mean
        }else{
          MeanSP = 0;
        }
        DecimalFormat df = new DecimalFormat("#.###");
        JOptionPane.showMessageDialog(view.getComponent(),
          "The network's mean shortest path: " + df.format(MeanSP) +
          "\n\nNumber of shortest paths: " + pathCount +
          "\nSum: " + sum);
```

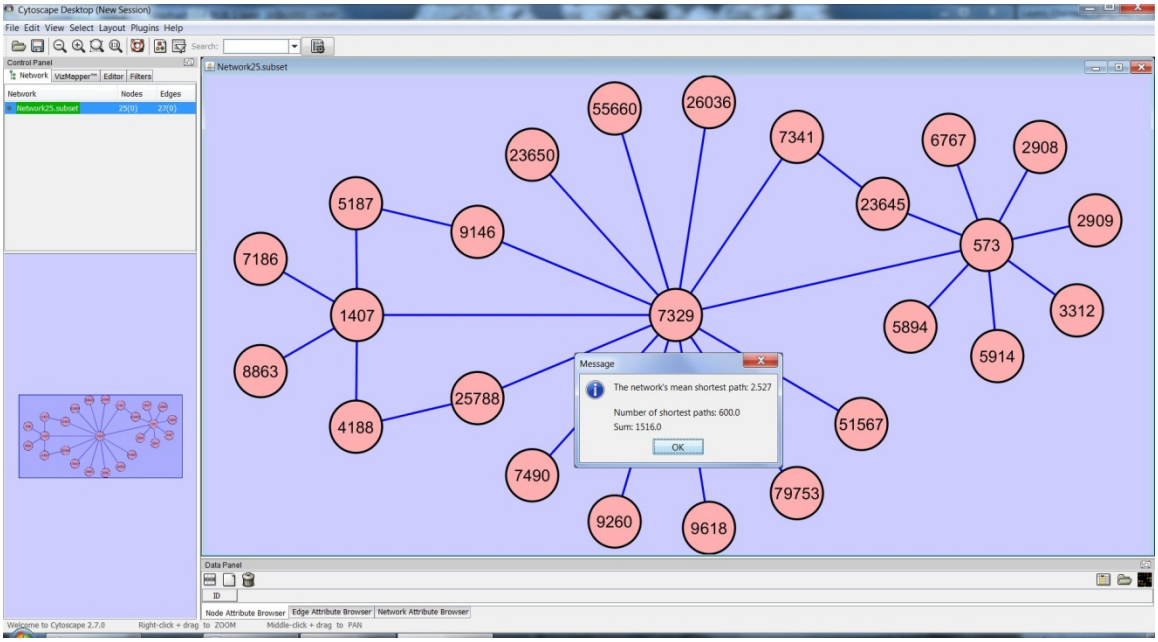

Fig. 3.4.1 - Mean Shortest Path

3.4.2 Diameter of Graph. The diameter is an indicator of the compactness of a biological network. For example, in a protein-signaling network, the diameter of a graph can provide

insight into the complexity associated with proteins communicating or influencing their neighbors. The diameter feature is also an example of functional convergence. A large protein network with a low diameter may indicate that the proteins within the network have functional co-evolution [16].

The Diameter of Graph operation is of course a graph-level operation and it functions much like the Mean Shortest Path operation. It too employs a modified version of the Shortest Path algorithm using depth first search on every node pair. However, instead of averaging the path lengths, Diameter of Graph examines the paths as they are calculated and records the longest of the resulting shortest paths. This longest shortest path in the network is called the diameter of the graph. The length of the diameter of the graph is returned by a pop-up box as shown is Fig 3.4.2.

```
//Iterate through all node combinations
        for (int i = 0; i < indexArray.length; i++){
           int node1 = indexArray[i];
           for (int j = 0; j < indexArray.length; j++){
              int node2 = indexArray[j];

              if(node1 == node2){}
              else {
                 int allPaths = getShortestPath(node1,node2);
                 output += allPaths + " ";
                 //Find the longest shortest path in the graph
                 if(allPaths > dia){
                    dia = allPaths;
                 }
              }

           }
        }
        dia = dia -1;
        JOptionPane.showMessageDialog(view.getComponent(),
           "The diameter of the graph: " + dia);
```

```
//+"\nShortest Paths: " + output);
```

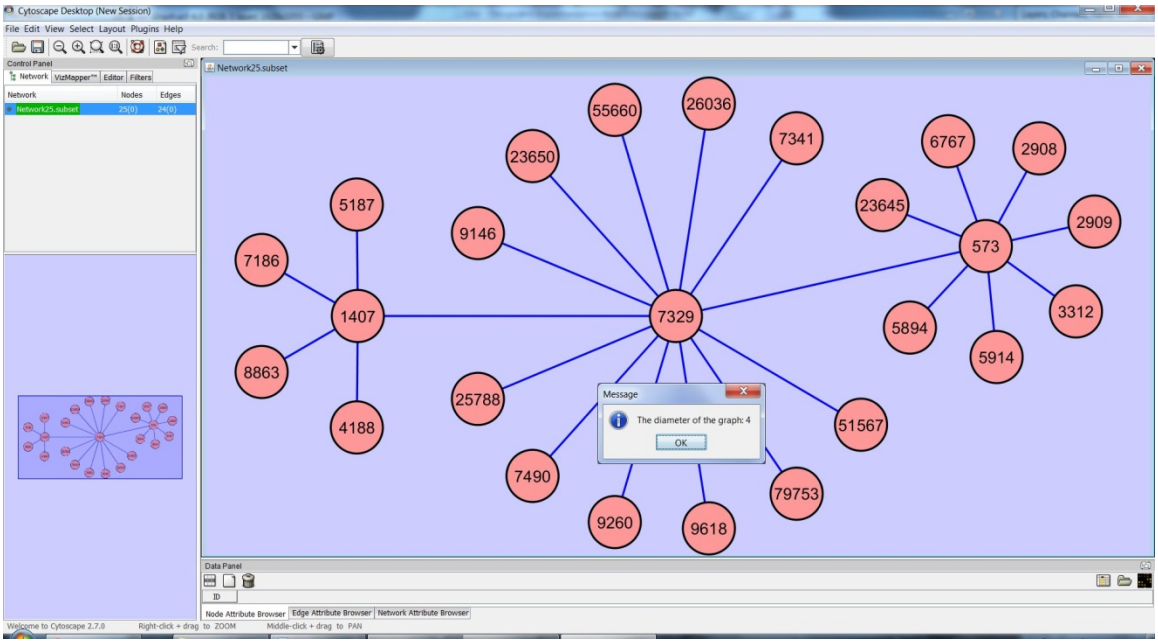

Fig. 3.4.2 - Diameter of Graph

3.4.3 Degree Distribution. The degree distribution represents the probability distribution of node degrees over the whole network. The shape of the degree distribution allows us to distinguish among different types of biological networks. In a scale free network, the degree distribution is exponential, signifying that most nodes have only one connection and few nodes are highly connected. On the other hand, protein nodes with lots of interactions tend to gain links more often.

Degree Distribution is defined as the fraction of nodes in a network that share a given node's degree. Degree is defined as the number of incident edges of a node.

Degree distribution is a graph-level operation that uses the following simple equation:

$$P(k) = \frac{n_k}{n}$$

Where k represents the degree value for which the equation is calculating. The symbol n_k is number of nodes in the network which have a degree of value k. The symbol n is the total number of nodes in the network. When the user executes the Degree Distribution operation, he is prompted to enter a value of degree for which to calculate and the result is displayed in a pop-up box as shown in Figs. 3.4.3.1 and 3.4.3.2, respectively.

```
public void DDcalc(){

    network = Cytoscape.getCurrentNetwork();
    view = Cytoscape.getCurrentNetworkView();
    root = Cytoscape.getRootGraph();
    DDSaveData = "";

    List nodeList = root.nodesList();
    double N = nodeList.size();
    int k = 0;
    int Nk = 0;
    int degree = 0;
    double Pk = 0;

    String input = JOptionPane.showInputDialog("Enter node degree");
    k = Integer.parseInt(input);

    //Iterate through all nodes and sum degrees
    for (Iterator iter = nodeList.iterator(); iter.hasNext();) {
        CyNode Node = (CyNode) iter.next();
        degree = root.getDegree(Node);
        if(degree == k){
            Nk += 1;
```

```
    }else{}
}

if(N == 0){
    Pk = 0; //Avoid divide by zero error for empty graphs
}else{
    Pk = Nk/N;
}
```

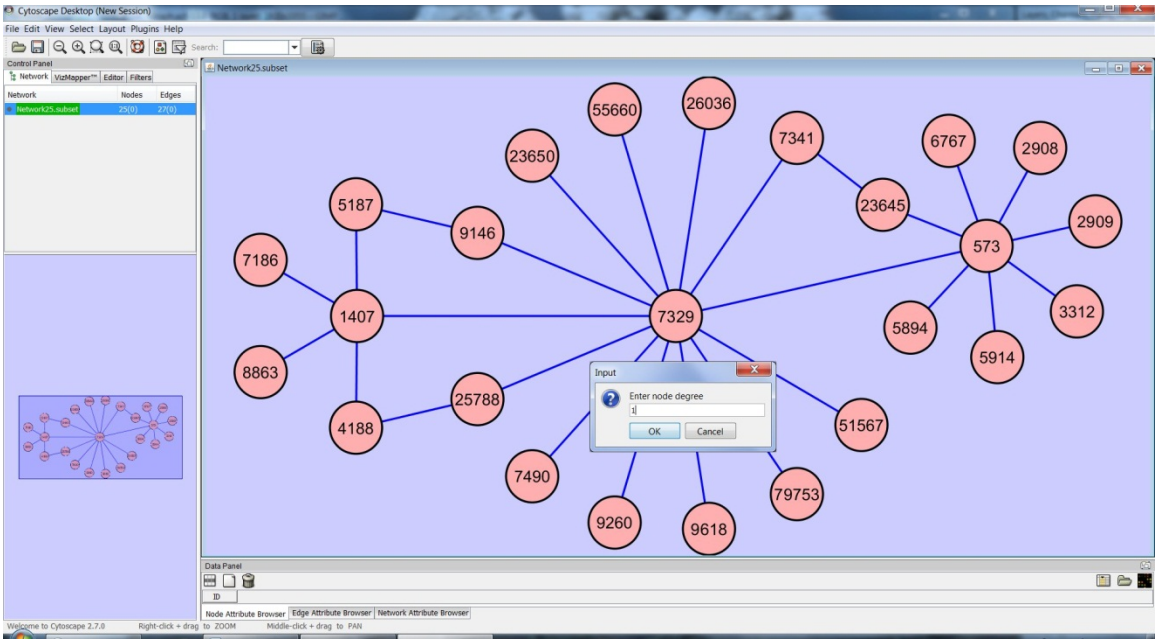

Fig.3.4.3.1 - Degree Distribution

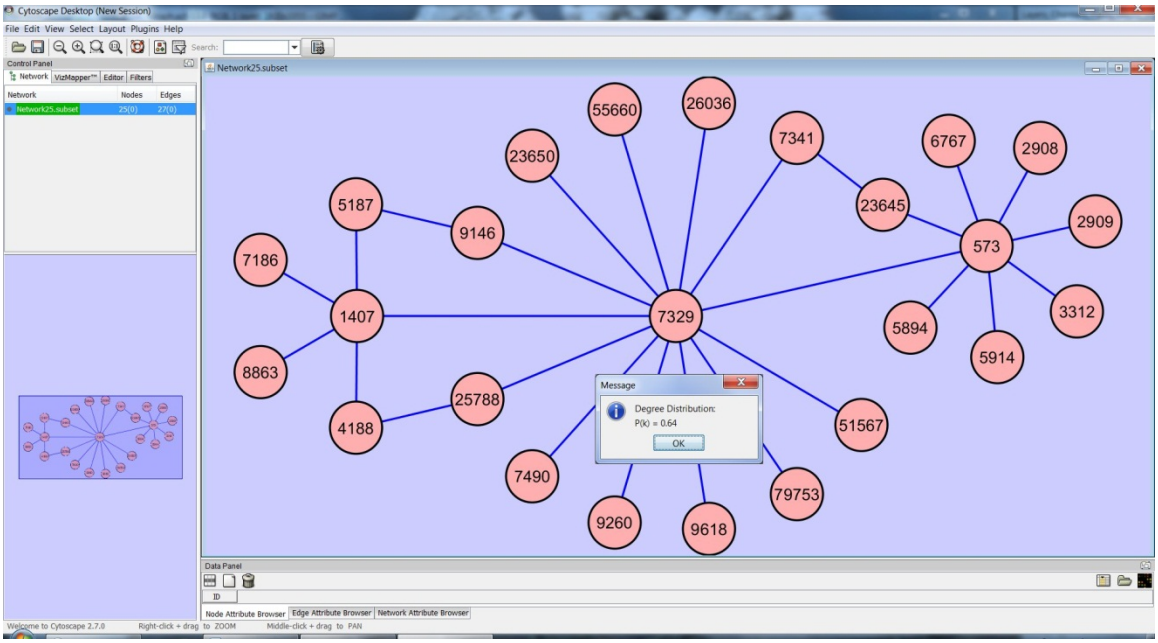

Fig. 3.4.3.1 – Degree Distribution - Continued

3.4.4 Average Number of Interaction Partners. The average number of interacting partners per protein provides insight into the dynamic interactions of a specific protein with its partners [17]. Analyzing these dynamic interactions is critical for rational design of drug molecules to modulate protein interactions. The average number of interaction partners is sometimes called the average number of neighbors. The Average Number of Interaction Partners operation is a graph-level operation that iterates through every node in the network, sums the number of neighbors each node has and finally divide the resulting sum by the total number of nodes in the network to calculate the average. The result is displayed in a pop-up box for the user as shown in Fig 3.4.4.

```
    public void ANIPcalc(){

        network = Cytoscape.getCurrentNetwork();
        view = Cytoscape.getCurrentNetworkView();
        root = Cytoscape.getRootGraph();
        ANIPSaveData = "";

        List nodeList = root.nodesList();
        double N = nodeList.size();
        int degree = 0;
        double sumDeg = 0.0;
        double avDeg = 0.0;

        //Iterate through all nodes and sum degrees
        for (Iterator iter = nodeList.iterator(); iter.hasNext();) {
              CyNode Node = (CyNode) iter.next();
              degree = root.getDegree(Node);
              sumDeg += degree;

        }
         //Calculate average degree
        avDeg = sumDeg/N;

        DecimalFormat df = new DecimalFormat("#.###");

        JOptionPane.showMessageDialog(Cytoscape.getDesktop(), "Average Number of Interaction Partners:
" + df.format(avDeg));

        //Save Data
        ANIPSaveData = "Average Number of Interaction Partners: " + avDeg;
        SaveData SDobj = new SaveData();
        SDobj.WriteFile(ANIPSaveData);
    }
```

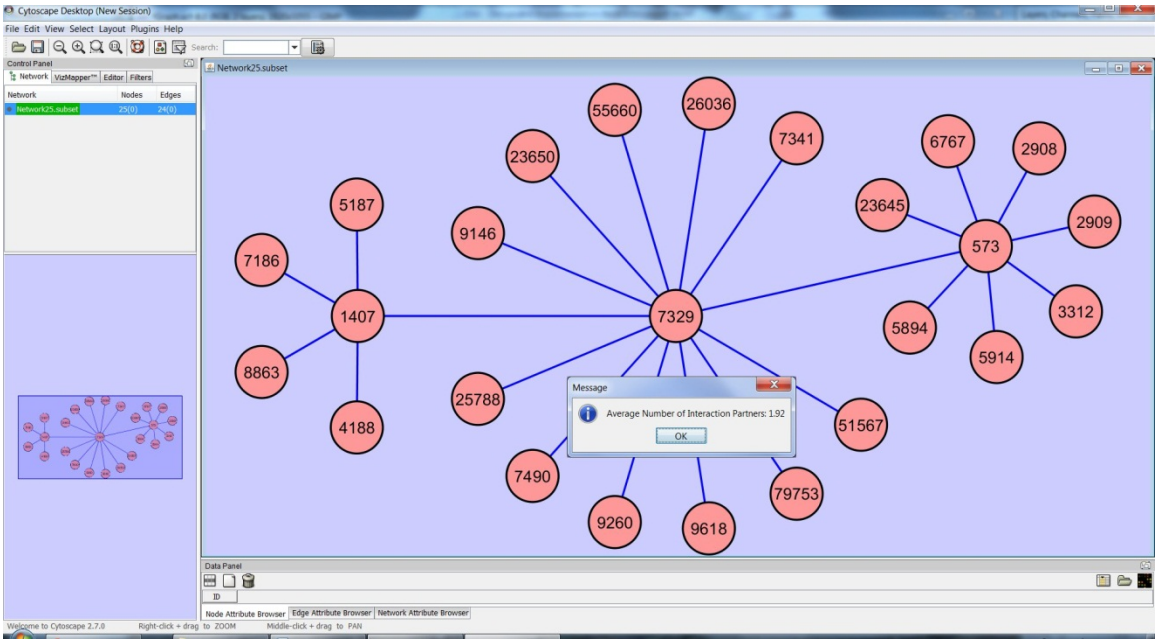

Fig. 3.4.4 - Average Number of Interaction Partners

3.4.5 Degree Centrality. The degree centrality provides an immediate evaluation of the regulatory relevance of a node in various biological networks. For example, in signaling networks, proteins exhibiting high degree interact with several other signaling proteins, thus indicating a central regulatory role. The role also indicates that these proteins are likely to be regulatory hubs. Jeong et al. (2001) demonstrated the correlation of the degree of a protein in the network with the lethality of its removal [18]. Hahn and Kern (2005) demonstrated how degree centrality has led to the identification of essential proteins in three different organisms: ***Saccharomyces cerevisiae***, ***Caenorhabditis elegans***, and ***Drosophila***

melanogaster [19]. Inall these example networks, it was shown that the mean centrality value for essential proteins is significantly higher than the centrality value of nonessential proteins. INVGA employs the Freeman Centrality [20] method to determine the relative importance of a vertex in a graph (i.e. how important is a vertex in the context of information flow on a particular path).

Degree centrality is the simplest and most direct method for calculating centrality. Degree centrality is used as an indicator of the immediate risk of a node coming into contact with whatever is flowing through the network. The Degree Centrality operation performed by INVGA is a graph-level operation. The formula for the degree centrality of a graph is below:

$$C_D = \frac{\sum_{i=1}^{|V|}[C_D(v^*) - C_D(v_i)]}{n^2 - 3n + 2}$$

where $C_D(v_i)$ is the value of the degree of the iterated node v. $C_D(v^*)$ represents the degree value of the node with the highest degree in the network. The symbol n is the total number of nodes in the network.

In order to efficiently calculate the degree centrality of a network, the numerator is algebraically rearranged into the following form:

$$n \times C_D(v^*) - \sum_{i=1}^{|V|} C_D(v_i)$$

This change approximately doubles the speed of calculating the formula from two iterations through the nodes of that network to one. The value of the graph's degree centrality is displayed to the user via a pop-up box as shown in Fig. 3.4.5.

```
public void DCcalc(){

    network = Cytoscape.getCurrentNetwork();
    view = Cytoscape.getCurrentNetworkView();
    root = Cytoscape.getRootGraph();
    DCSaveData = "";

    List nodeList = root.nodesList();
    double n = nodeList.size();
    double pMax = 0.0; //highest degree in network
    double p = 0.0; //degree of current node
    double sumDC = 0.0;
    double numDC = 0.0; //numerator of degree centrality
    double denomDC = 0.0; //denominator of degree centrality
    double Cd = 0.0; //Degree Centrality

   //Iterate through all nodes and sum degrees
    for (Iterator iter = nodeList.iterator(); iter.hasNext();) {
       CyNode Node = (CyNode) iter.next();
       p = root.getDegree(Node);
       sumDC += p;
       if(p >= pMax){
          pMax = p;
       }else{}
    }

    numDC = n*pMax-sumDC;
    denomDC = n*n-3*n+2;

    Cd = numDC/denomDC; //Calculate Degree Centrality

    DecimalFormat df = new DecimalFormat("#.####");

    JOptionPane.showMessageDialog(Cytoscape.getDesktop(), "Degree Centrality:\n" + "CD = " +
df.format(Cd));
```

```
//Save Data
      DCSaveData = "Degree Centrality: " + df.format(Cd);
      SaveData SDobj = new SaveData();
      SDobj.WriteFile(DCSaveData);
   }
```

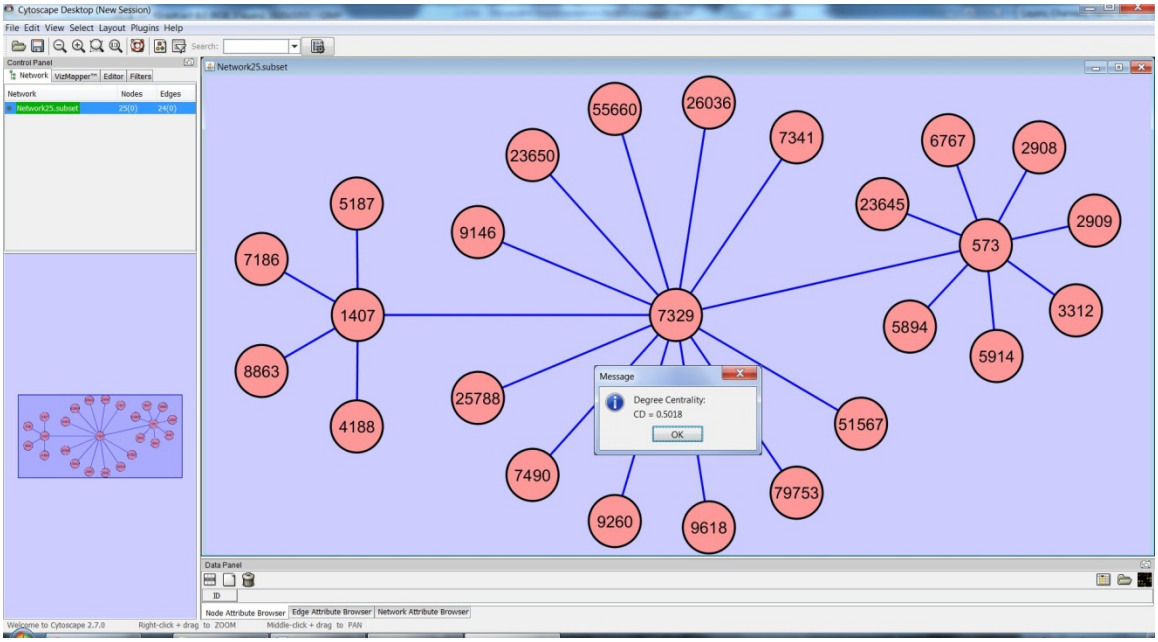

Fig. 3.4.5 - Degree Centrality

3.4.6 Clustering. Clustering of biological networks, such as protein-protein interaction network, and metabolic network, is one of the most common approaches for identifying functional modules and protein complex, predicting protein or gene functions [21]. INVGA implements MCODE [7] for intuitive visualization of clustering results and easy-to-use topological analysis.

As was stated in the previous chapter, the clustering operation of INVGA is performed by the 3rd party clustering plugin MCODE. Since INVGA is meant to supply researchers of protein-protein interactions with a complete set of basic tools for their research, the INVGA plugin requires a clustering function. The decision to employ the MCODE plugin was made after surveying the large number of clustering plugins already available in the Cytoscape platform. Given the amount and capabilities of these plugins, it was judged to be a misallocation of time to develop another clustering operation specifically for INVGA and that it would be far more practical to utilize MCODE. The jar file for MCODE is included in the download for INVGA and INVGA simply calls MCODE when the user executes the Clustering operation from the INVGA operations menu. The MCODE plugin is unchanged and the developers of INVGA take no credit for this work.

3.5 Miscellaneous Operations.

The operations in this section are not mathematical like the previous operations. The operations simply perform important functions of usability for the user.

3.5.1 Reset Colors. At the bottom of INVGA's operations menu there is an operation titled RESET COLORS. This function simply returns all nodes in the

network back to their original colors. The decision to make this function a separate operation is twofold. The first reason is efficiency. If this function were built into the beginning of every operation, so that the node colors would be reset before the next operation was run, it would require an extra iteration through the network for every operation. If an extremely large network is being analyzed, this could add a significant amount of time to each operation even if the resetting node colors isn't of concern to the researcher. The second reason is that by having the node color reset function as a separate operation, it is now possible to compare multiple paths simultaneously in a visual manner.

3.5.2 Save Data. The Save Data operation does not appear in the operations menu and cannot be manually executed. Instead, this operation functions automatically in the background every time a mathematical operation is performed. When INVGA is activated for a new session, Save Data automatically creates a log file titled in the following format: INVGA_Log_NetworkName_Year_Month_Day_Hour_Minute_Second.txt. Every time a mathematical operation is executed, the name of the operation and the corresponding values returned are appended to this file. Each operation is appended as its own line in the log. Here are a few examples of the formatting.

Dyad-Level Operation:

Format:

```
Operation Name(Source ID, Sink ID): [IDs of nodes along path] Other Data: value
```

Example:

```
Critical Path(7329, 4188): [7329 9146 1407 4188]   Length: 6
```

Node-Level Operation:

Format:

```
Operation Name(Node ID): value
```

Example:

```
Topological Coefficient(7329): 0.15385
```

Graph-Level Operations:

Format:

```
Operation Name: Primary Data     Other Data: value
```

Example:

```
Mean Shortest Path: 2.513   Number of Shortest Paths: 600.0   Sum: 1508.0
```

The log provides a convenient and clear record of everything that the user calculates to avoid the need to manually copy the output with a pen and paper after every operation. The log file is saved to the Cytoscape directory.

Chapter 4

Performance Analysis

In this chapter we will examine the performance of the INVGA plugin in various respects. Given the extremely large size of many biological networks, efficiency is of great importance to analysis in this field [2]. Both the memory performance and the theoretical efficiency of the underlying algorithms for the different operations offered by INVGA are examined here.

4.1 Memory Performance

As the size of the network increases from 25 nodes to 300 nodes, the memory usage is obtained for and compared between INVGA and a combination of four other plugins, i.e., Network Analyzer, Shortest Path Plugin, CentiScaPe and CalculateNodeDegree. Note that the combination of the four plugins has the biggest overlap with INVGA in terms of the topological analysis at the network-, node- and dyad-level. For this performance testing, a 3rd party system memory monitoring program, DTaskManager [22], is used to together with Cytoscape version 2.7. The system specifications of the computer used to run the tests are as follows: Windows 7 Home Premium 64-bit operating system with Intel Core i7 CPU Q720 @1.60Ghz and 8GB.

Using the data in Table 4.1, 95% confidence interval for the mean difference in average memory usage is calculated. The approximate 100(1-α)% confidence interval is defined as:

$$\hat{\theta} \pm g_{\alpha/2,(n-1)}\hat{V}(\hat{\theta}) \text{ or } \hat{\theta} - g_{\alpha/2,(n-1)}\hat{V}(\hat{\theta}) \le \theta \le \hat{\theta} + g_{\alpha/2,(n-1)}\hat{V}(\hat{\theta})$$

where $\hat{\theta}$ is a sample mean of θ based on a sample of size n; $\hat{V}(\hat{\theta})$ is the standard error of $\hat{\theta}$; $g = \dfrac{\hat{\theta} - \theta}{\hat{V}(\hat{\theta})}$ and $g_{\alpha/2,\,(n-1)}$ is the 100(1-α)% percentage point of a t-distributed with n-1 degrees of freedom. The value of $g_{\alpha/2,\,(n-1)} = g_{0.0025,\,3} = 3.18$ is obtained from t-distribution table [23]. A 95% confidence interval for average disassembly time is given by:

$$-8765 \pm (3.18)1474$$

or

$$-13452.32 \le \theta_1 - \theta_2 \le -4077.68$$

The 95% confidence interval for average memory usage lies completely below zero, which provides strong evidence that $\theta_1 - \theta_2 < 0$ — that is, the INVGA is better than the combination of the other four plugins, because its average memory usage is smaller.

Table 4.1 - Comparison of Average Memory Usage

Size of networks	Average Memory Usage (KB)		Observed Difference
	INVGA	Combination	
25 nodes	131,136	137,214	-6,078
50 nodes	131,262	139,521	-8,259
100 nodes	132,716	139,748	-7,032
300 nodes	132,938	146,630	-13,692
Sample mean	132,013	140,778	-8,765
Sample variance	670741	12397237	8688656
Standard error			1474

4.2 Big O Notation

Big O notation describes the performance or complexity of an algorithm. It is important to remember that Big O is a description of the worst-case scenario for the algorithm. Here we examine the performance of the operations of INVGA using Big O notation. For the operations that share the common core process, we analyze them together.

4.2.1 DFS Dyad Operations. All but one of the dyad operations of INVGA employ depth first search. Those are All Paths, Shortest Path, and Critical Path. In essence, all of these operations run the All Paths algorithm which returns an arraylist of all the unique,

noncyclical paths between the source and sink nodes. Shortest Path and CriticalPath then iterate through this arraylist in a linear fashion and return the single path that meets their individual criteria. In comparison to the complexity of All Paths algorithm, such iterations are trivial. Therefore, the analysis here focuses on All Paths algorithm and the result applies to the other two algorithms by extension.

For a given network with N number of vertices, All Paths algorithm starts with a process of building a 2 dimensional NxN capacity matrix. In this 2D matrix every row represents a node and every column corresponds to another node in the network. A value of 1 denotes an edge connecting the two nodes and a value of 0 means that they are not connected. All Paths algorithm then employs DFS. The complete implementation of DFS is presented in section 3.1.1, however, the portion of code that is relevant to calculating the efficiency of DFS is:

```
for (int k = 0; k < size; k++){
                int v = capacity[u][k];
                if (v != 0 && visited[k] != true){
                   DFS(k);
                }
```

In the *for* loop, *k* equals the size of the network, so here the algorithm must necessarily iterate through every node in the network. The next part is recursive. There is an *if* statement where if there is a neighbor node that has not been

visited, DFS is run again. Given the worst-case scenario, where each node is connected to every other node, this would result in nesting N more calculations in each previous N calculation. That gives DFS an efficiency of $O(N^2)$. Finally, the All Paths algorithm iterates through the path matrix resulted from DFS to color the nodes involved in all paths. Iterating through a 2D matrix adds another nested loop, which is $O(N^2)$. The total computational complexity of the All Paths algorithm is $O(N^2) + O(N^2) = O(2* N^2)$. After simplifying, the worst case scenario results in a complexity of:

$$O(N^2).$$

4.2.2 Maximum Flow. The only operation offered by INVGA that utilizes breadth first search is Max Flow. It is known that BFS also results in an efficiency of $O(|N|+|E|)$ [24]. This is the same as DFS for the same reasoning that given the worst case scenario, every vertex and edge would be traversed before finding the desired path. Also, just since like the DFS algorithm the highest complexity results from iterating through 2D matrices, it results in the same efficiency of:

$$O(N^2)$$

4.2.3 Degree Distribution, Degree Centrality, & Average Number of Interaction Partners.

The Degree Distribution, Degree Centrality, and Average Number of Interaction Partners are examined here together since they share the same complexity. All three of these operations are very simple and straight forward. They begin by iterating through a list of all the nodes of the network. At each node, the value of its degree is examined via the *getDegree()* function from the Cytoscape API. It is intuitively understandable that the complexity getDegree() is N as the function searches through the list of all nodes and determines the number of the nodes connected the node of concern. Depending on the individual operation, these degree values either are summed, their maximum recorded, or matches of an exact value counted. Finally, a simple arithmetic operation is performed with the results to produce the final output. There is only one significant action in these operations, which is the nested iteration through all the nodes in the network. Therefore, in all cases the complexity of these operations is:

$$O(N^2)$$

4.2.4 Topological Coefficient. The Topological Coefficient operation is a unique operation within INVGA since it is the only node-level operation. Topological Coefficient's calculation is based on how many neighbor nodes the selected node has in common with those of the

rest of the nodes in the network. This can be thought of as "Mutual Friends". The operation begins by getting an array of neighbors of the user selected node called node "A". Then it gets the neighbor arrays of the other nodes, called the "B" nodes. These neighbor arrays would normally be small sub-arrays of the network, but in a worst case scenario they can contain every other node in the network, making it *N-1*. These arrays of neighbors of the B nodes are iterated through to find matches with the neighbors of the A node. This forms a nested loop which results in a complexity of $(N-1)^2=N^2-2N+1$. After eliminating the insignificant terms, we are left with a complexity of:

$$O(N^2)$$

4.2.5 Mean Shortest Path & Diameter of Graph. Mean Shortest Path and Diameter of Graph are graph-level operations of INVGA that operate very similarly to each other. Unfortunately, they are inherently complex algorithms. Simply put, these operations run the Shortest Path algorithm, but instead just finding the shortest path between two nodes, they run Shortest Path on every combination of node pairs. It should be immediately apparent that this will greatly compound that complexity of the algorithm. Since Shortest Path already has a complexity of

$$O(N^2)$$

Since the combination of every article in a set of N is $N!$, this added complexity results in Mean Shortest Path and Diameter of Graph having a complexity of:

$$O(N!)(N^2)$$

Because of this high level of complexity, steps have been taken to reduce the calculation time. Since a path cannot be formed with the same node as both the source and sink nodes, these combinations are omitted from the calculation. However, this provides only a minor improvement.

Chapter 5

Conclusion

As mentioned previously, there has been an ever increasing amount of molecular data generated by experiments and computational methods performed on biological networks. As a result, there is a growing need to obtain insight into the organization and structure of massive and complex biological networks formed by interacting molecules. Given the large size of these networks it is necessary to employ a systematic approach to analyze the raw data in order to better understand their properties. *Integrated Network Visualization and Graph Analysis* (INVGA) is one of such contributions to help fulfill this growing need by implementing a list of network analysis functions that enables researchers to quickly and easily analyze and visualize complex bioinformatics data on the node, dyad, and network levels. Those functions, many of which aren't available for Cytoscape users anywhere else, include: All Paths, Shortest Path, Critical Path, Max Flow, Mean Shortest Path, Degree Distribution, Average Number of Interaction Partners, Clustering Coefficient, Topological Coefficient, Diameter of Graph, and Degree Centrality. In addition, the performance advantages of having a collection of basic network analysis tools combined in a single plugin

has been demonstrated to be superior, by reducing overhead, as opposed to loading many separate plugins to accomplish the same tasks. INVGA presents the additional advantage of having a consistent user interface for all of its functions. This decreases the learning curve for users since they no longer are required to learn to navigate the interfaces of multiple, dissimilar plugins. It is this author's hope that this plugin for Cytoscape will assist researchers in predicting the effects and limiting potential side effects of drugs before they begin testing on animals or humans, resulting in savings in time and money.

5.1 Future Work

The INVGA is created to provide all the fundamental graph analysis algorithms that are essential to understand protein-protein interaction networks but are not provided by Cytoscape and have not been fulfilled by the Cytoscape plugin development community. The objective is to provide researchers everywhere, such a tool to assist them in gaining better insight into the properties of complex protein-protein interaction networks. In the future, the functions of INVGA could be expanded beyond the fundamental operations that are currently implemented, such as multiple methods of measuring centrality, and the incorporation of graph that display the plotted data that results from the current operations. In addition, we plan to expand INVGA to accept bioinformatics data

encoded in different formats other than what Cytoscape natively supports. It is this author's sincere wish that researchers of protein-protein interaction networks, or researchers of less directly applicable networks, find this work beneficial in some form in the course of their efforts.

List of References

1. Radu Jianu, Kebing Yu, Lulu Cao, Vinh Nguyen, Arthur R. Salomon, and David H. Laidlaw, "Visual Integration of Quantitative Proteomic Data, Pathways, and Protein Interactions", IEEE Transaction on Visualization and Computer Graphics, Vol. 16, No. 4, pp. 1.

2. J. Kohler, J. Baumbach, J. Taubert, M. Specht, A. Skusa, A. Ruegg, C. Rawlings, P. Verrier, and S. Philippi, "Graph-based analysis and visualization of experimental results with ONDEX," Bioinformatics, Vol. 22, No. 11, 2006, pp. 1383-1390.

3. S. D. Hooper and P. Bork, "Medusa: A simple tool for interaction graph analysis," Bioinformatics, Sept. 27th, 2005.

4. Shortest Path plugin, http://chianti.ucsd.edu/cyto_web/plugins/displaypluginfo.php?name=ShortestPath%20Plugin

5. C. Y. Lin, C. H. Chin, H. H. Wu, S. H. Chen, C. W. Ho, and M. T. Ko, "Hubba: hub objects analyzer – a framework of interactome hubs identification for network biology," Nucleic Acids Research, Vol. 26, 2008, pp. 438-443.

6. G. Scardoni, M. Petterlini, and C. Laudanna, "Analyzing biological network parameters with CentiScaPe," Bioinformatics, Vol. 25, No. 21, pp. 2857-2859.

7. Bader GD and Hogue CWV, "An automated method for finding molecular complexes in large protein interaction networks," BMC Bioinformatics 2003, 4:2.

8. Y. Assenov, F. Ramirez, S. E. Schelhorn, T. Lengauer, and M. Albrecht, "Computing topological parameters of biological networks," Bioinformatics, Vol. 24, No. 2, 2008, pp. 282-284.

9. D. Croes, F. Couche, S. Wodak, and J. v. Helden, “Metabolic path finding: inferring relevant pathways in biochemical networks,” Nucleic Acids Research, 33:W326-W330, 2005.

10. D. Croes, F. Couche, S. Wodak, and J. v. Helden, “Inferring meaningful pathways in weighted metabolic networks,” J. Mol. Biol., 356:222-236, 2006.

11. Saliha Durmuş Tekir, Pelin Ümit, Aysun Eren Toku, and Kutlu Ö. Ülgen, “Reconstruction of Protein-Protein Interaction Network of Insulin Signaling in Homo Sapiens,” Journal of Biomedicine and Biotechnology, Volume 2010 (2010).

12 Ying Xu, Dong Xu, and Harold N. Gabow, “Protein domain decomposition using a graph-theoretic approach,” Bioinformatics (2000) 16(12): 1091-1104.

13 Luca Trevisan, "CS261 Lecture 9: Maximum Flow", http://lucatrevisan.wordpress.com/2011/02/04/cs261-lecture-9-maximum-flow/, February 4, 2011.

14 Kar G, Gursoy A, Keskin O, “Human Cancer Protein-Protein Interaction Network: A Structural Perspective,” PLoS Comput Biol, 2009, 5(12)

15 Rob M Ewing, “Large-scale mapping of human protein–protein interactions by mass spectrometry,” Mol Syst Biol. 2007, 3: 89.

16 D. Koschützki and F. Schreiber, “Centrality analysis methods for biological networks and their application to gene regulatory networks,” Gene Regulation and Systems Biology, 2: 193-201, 2008.

17 Shide Liang, Chi Zhang, Song Liu, and Yaoqi Zhou, “Protein binding site prediction using an empirical scoring function,” Nucl. Acids Res. (2006) 34(13).

18 Jeong H, Mason SP, Barabási AL, Oltvai ZN, “Lethality and centrality in protein networks,” Nature, 2001; 411:41–2.

19 Hahn MW and Kern AD, "Comparative genomics of centrality and essentiality in three eukaryotic protein-interaction networks," Mol. Biol. Evol. 2005;22(4):803–6.

20 L. C. Freeman, "Centrality in Social Networks Conceptual Clarification," Social Networks, 1, 1978, pp. 215-239.

21 J. Cai, G. Chen, J. Wang, "ClusterViz: A Cytoscape Plugin for Graph Clustering and Visualization," http://clusterviz-cytoscape.googlecode.com/files/ClusterViz.pdf2010.

22 DTaskManager, http://www.downloadznow.com/get/dtaskmanager.

23 Banks, J., Carson, J. S. II, Nelson, B. L. and Nicol, D. M., Discrete-Event System Simulation, Prentice Hall, 2000.

24 Cormen, Thomas H., Charles E. Leiserson, and Ronald L. Rivest. p.590