Network Tools for the Analysis and Prediction of Protein-Protein Interactions

by

Kevin T. Weston, Jr.

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2004

Author ... .
Department of Electrical Engineering and Computer Science
May 20, 2004

Certified by.. .....
Amy E. Keating
Assistant Professor
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Network Tools for the Analysis and Prediction of Protein-Protein Interactions

by

Kevin T. Weston, Jr.

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2004, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

In this thesis, we present two computational platforms for future biological research. The first, FNAC, is a flexible programmatic *Framework for Network Analysis and Comparison* that simplifies many common operations on biological networks. As a demonstration of FNAC, we investigate the properties of several prominent protein function and protein-protein interaction networks. In doing so, we uncover evidence suggesting that a recently-developed technique for annotating proteins may also have substantial value in the computational prediction of protein-protein interactions. Our second computational platform, the *Coiled-Coil Database (CCDB)*, serves as a central and easily queryable repository for information about the coiled coil protein structural motif in a variety of organisms.

Thesis Supervisor: Amy E. Keating
Title: Assistant Professor

Acknowledgments

First I need to thank my advisor Amy Keating and Taijiao Jiang, a post-doc in our lab, for their amazing ideas, drive, and constant feedback, without which this work would not have been possible. I would also like to thank two other members of the lab—Gevorg Grigoryan and Shaun Deignan—for their constructive comments throughout the development of this document.

In addition, I owe a tremendous amount of gratitude to Susan, who has consistently born the brunt of my thesis-induced misery and, yet, remained patient throughout.

Above all, though, none of this work would have been possible without the years of support and continuing encouragement I've received from my parents, Kevin and Deborah. Mom and Dad, this thesis is for you.

Contents

List of Figures

Chapter 1

Introduction

The simple network has emerged as an extremely common and powerful model for many different types of biological data. This thesis introduces FNAC, a flexible programmatic *Framework for Network Analysis and Comparison.* FNAC, described in Chapter 2, is a Java API that builds on a variety of previous work to simplify many common network operations. Use of FNAC can speed the development of future network-based computational approaches and, by doing so, drive substantial biological discoveries.

As a demonstration of FNAC, Chapter 3 describes the application of many of the framework's features to the analysis and comparison of several protein networks. This application of FNAC yields several new insights into the relationship between functional annotation of proteins and experimentally observed interactions.

This document also introduces the CCDB, a persistent, searchable *Coiled-Coil Database.* The CCDB, described in Chapter 4, is a general repository for information about the coiled coil protein structural motif and can be readily applied, either on its own or in conjunction with FNAC, to genome-wide protein-protein interaction studies.

1.1 Biological Network Analysis

While by no means a novel concept, the representation of many types of biological data as graphs has grown very popular in recent years. Such networks can be both efficient and visually illustrative models of large and complex data sets. They are also fairly easy to manipulate and can be used to represent everything from ecological food webs[21] to genetic regulation[27]. Importantly, network properties and algorithms for graph analysis have been studied by theorists for hundreds of years. The primary driving force for the adoption of graphs in biology, however, has been the large volume of data recently generated by efficient computational and experimental techniques. These data have finally enabled meaningful analysis of natural networks and the discovery that these graphs possess many unique and intriguing properties.

1.1.1 Connectivity and Clusters

Many interesting properties of biological networks involve the connectivity of their nodes; that is, the number of edges incident on each node. It has been observed that the connectivity distributions of most such networks (including dozens of metabolic networks[14], the yeast protein-protein interaction network[13], and several ecological food webs[21]) obey a power law:

$$P(k) \sim k^{-\alpha} \tag{1.1}$$

That is, most nodes are connected to only one or two edges, with the probability of a node being connected to k neighbors decreasing exponentially with k. This is in contrast with the previous assumption that biological networks are essentially random, in which case a Poisson connectivity distribution would be expected[14]. The power law distribution, illustrated in Figure 1-1, also implies that a few nodes, called "hubs," are very highly connected. Disruptions targeted at these nodes can therefore have disastrous and far-reaching consequences, leading Barabasi et al. to denote such networks as "scale-free."[4]

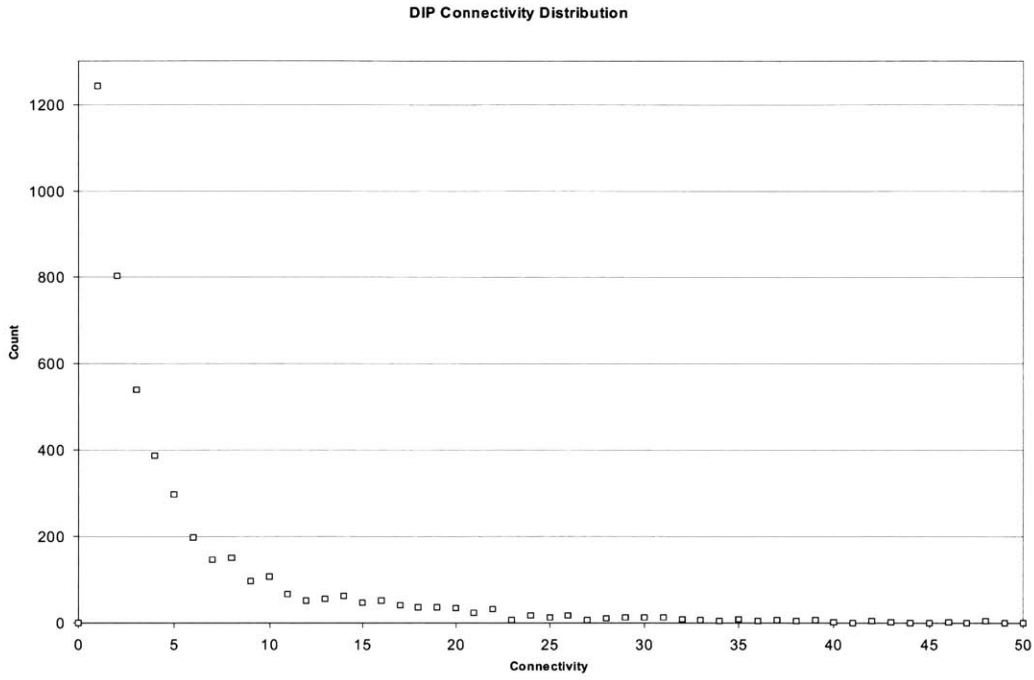


Figure 1-1: Example connectivity distribution. The connectivity distribution of the Database of Interacting Proteins' (DIP) protein-protein interaction network for the yeast *Saccharomyces cerevisiae* exhibits the exponential decrease typical of scale-free networks.

In addition to connectivity, previous researchers have also described metrics of the "clustering" of nodes[1]. The clustering of a node represents the degree to which its neighbors are fully interconnected. At the extremes, a clustering coefficient of 1 indicates that a node and its neighbors form a fully-connected subgraph, while a clustering coefficient of 0 indicates that none of a node's neighbors are connected to any other. Biological networks often simultaneously exhibit distinct regions of high and low clustering. A hierarchical pattern emerges in which many densely-connected clusters are loosely connected to one another, forming a set of less densely-connected modules which are, in turn, loosely connected among themselves[23].

1.1.2 Network Motifs

Most recently, attention has been focused on the notion of network "motifs," multi-node topological patterns that, statistically, are significantly more prevalent in a particular network than in randomly rewired versions of the same network[27, 20]. Such motifs have been identified in many different types of naturally-occurring networks and biological explanations have been proffered for several of these. The feed-forward loop motif, for example, has been particularly well characterized by Mangan and other members of the Alon group and has been suggested to be central to time-dependent transcriptional regulation[17, 18].

1.2 Protein-Protein Interaction Networks

One class of biological network that is of particular interest to the Keating lab is the protein-protein interaction network. In this type of network, each node represents a protein and each edge represents a direct physical interaction between two proteins. Preliminary protein-protein interaction networks have been generated for several species based on recently developed high-throughput interaction assays such as yeast two-hybrid assays[28, 11], co-purification/mass spectrometry[10], and protein arrays[34]. In addition, these techniques—in conjunction with classical low-throughput experiments—have been used to compile several large, public databases

of protein-protein interactions including DIP[32], BIND[3], MINT[33], and MIPS[19]. An example network, generated from the DIP-CORE database (described in Chapter 3), is shown in Figure 1-2.

1.2.1 Motivation for Computational Interaction Prediction

Despite the publicity that the above protein-protein interaction networks have received, the high-throughput experiments from which they are all derived are incomplete and unreliable (up to 50% false positive rate), with little overlap in results (less than 20% overall)[25]. Classical analyses can detect whether two particular proteins interact with very high confidence. However, such experiments are very time consuming and cannot possibly be used to test all half a billion possible interactions between the ~35,000 human proteins or even the ~18,000,000 possible interactions among the the ~6,000 proteins of the much-studied yeast *Saccharomyces cerevisiae.*

Nonetheless, the construction of complete and accurate protein-protein interaction networks is very important to modern biology and medicine. Proteins are vital to life as we know it—they form the foundation of cellular structures, facilitate transport of materials into and out of cells, and come together to regulate nearly all biological processes. Even minor alterations in the interaction pattern between proteins can lead to many critical malfunctions, including cancer[9, 6, 29, 24].

Fortunately, we believe that a growing wealth of tangentially-related information about proteins can be integrated computationally to improve the confidence of high-throughput techniques and, in some cases, suggest interactions not detected by these methods. Specifically, we expect the function, cellular location, and physical structure of a protein to be useful predictors or contradictors of interaction. For example, we would expect a higher likelihood of interaction between a phosphatase (a protein that cleaves phosphate groups off of other proteins) and a protein with many phosphate groups than a protein lacking phosphates. Similarly, we would expect proteins primarily located in the nucleus to be most likely to interact with other proteins located in the same compartment. Moreover, certain classes of structures are known to preferentially interact with certain other structures.

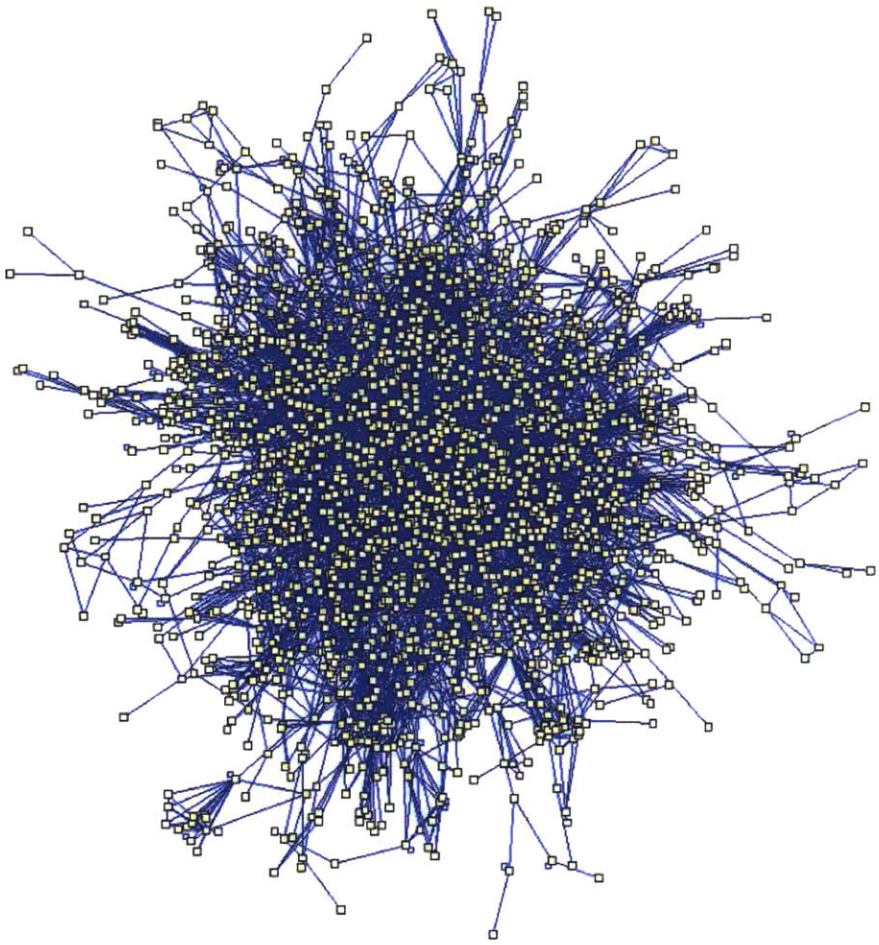


Figure 1-2: Example protein-protein interaction network. Despite its visual appearance, the network representing the protein-protein interactions in DIP-CORE (described in Chapter 3) is quite sparse, with just 5,581 edges between 2,386 proteins. Figure generated using Cytoscape[26].

Several computational interaction prediction methods have already been developed based on integration of these types of information. All of these techniques seek to merge many network representations of the above data into a single high-confidence predicted protein-protein interaction network. The simplest approaches involve computing either boolean or weighted pairwise unions of multiple networks, while others involve more elaborate graph-theoretical methods like Jansen et al.'s Bayesian network-based machine learning scheme[12]. While there is some evidenciary support that these approaches improve the confidence of high-throughput experimental data sets, no measure of general reliability has been presented. Recent work in the Keating lab has combined simple correlation analysis and a decision tree-based machine learning scheme to produce the AVID integration technique described later in this chapter[15]. Although AVID hasn't yet been applied to protein-protein interaction prediction, data presented in Chapter 4 suggests its potential utility for exactly that purpose.

1.3 Annotation Networks and Gene Ontology

In order to examine whether function and localization indeed correlate with interaction, we first need to collect such annotations and represent them in a manner well-suited to comparison with the interaction networks. One solution is to model the data as a set of networks, but, in doing so, we encounter three major challenges. First, a common language (both vocabulary and grammar) is needed to describe these protein properties. Second, a reasonably comprehensive source of such annotations is needed. Finally, a method is needed for representing the annotations of an individual protein as edges connecting it to other proteins.

Fortunately, the Gene Ontology (GO) Consortium provides a solution to the first challenge by defining a standard framework and controlled vocabulary for annotation[2]. GO divides protein descriptions into three distinct ontologies: the Molecular Function (MF) associated with a protein, the Biological Process (BP) in which it is involved, and the Cellular Component (CC) in which it is known to exist. In each of these

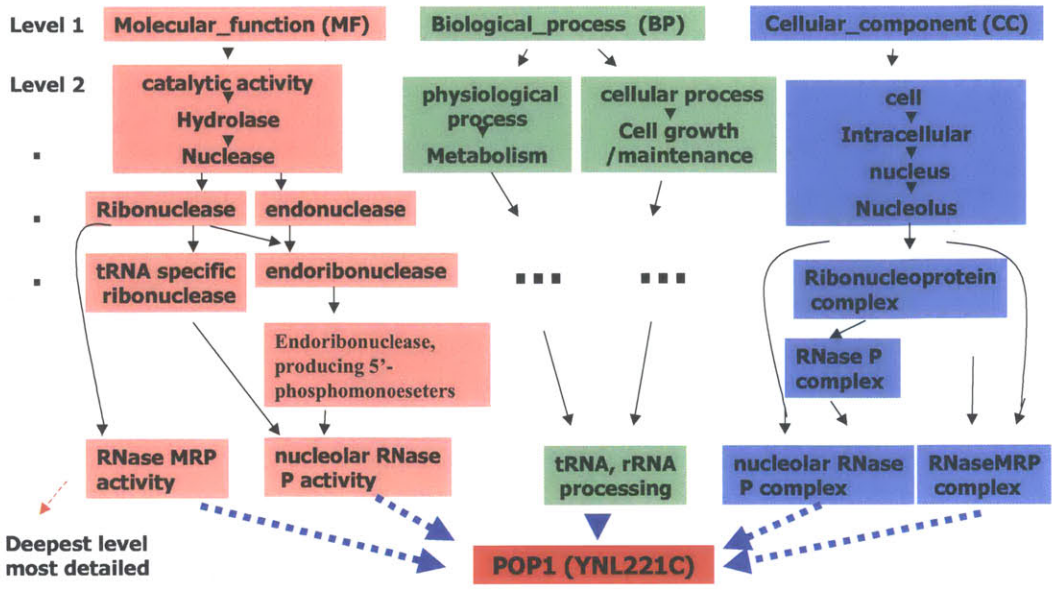


Figure 1-3: GO annotation for the yeast *POP1* protein. The GO annotations of *POP1* illustrate the distinction between the Molecular Function, Biological Process, and Cellular Component ontologies. Additionally, these descriptions exemplify the hierarchical nature of the GO vocabulary at several levels of detail. Figure courtesy of T. Jiang.

categories, GO also defines a broad hierarchical vocabulary to describe proteins in a very general or a very specific manner, depending on the degree to which the protein's operation is understood. An example of the annotation hierarchy for a single protein is shown in Figure 1-3.

In response to the second challenge, many independent groups, working in collaboration with the GO Consortium, maintain GO ontologies for approximately two dozen of the most commonly studied organisms including bacteria, yeast, nematodes, plants, fruit flies, mice, rats, and humans. All of these data sets are well-documented, regularly updated, and available for public download in a common format from the GO website.

The representation of GO ontologies as networks, however, is not a practice addressed by the GO Consortium or any of its current collaborators. The most simple and straightforward solution, which we have adopted, is to represent each ontology (MF, BP, and CC) has an independent network. In each, we connect with an edge

every two proteins that share a common annotation at the most detailed levels. This leads to fully-connected clusters of proteins that occasionally intersect at proteins with multiple annotations. Our GO networks are not scale-free and are not expected to contain meaningful network motifs. However, they do describe certain relationships between proteins and, in this thesis, we test the extent to which they correlate with protein-protein interaction networks. Such a correlation would make them useful in computational interaction prediction.

1.4 Annotation Via Integration of Data (AVID)

The GO ontologies are far from complete in their description of proteins. Proteins that have not been characterized in small-scale experiments often have no GO annotations at all, while many others are described only at the coarsest levels. In fact, the most detailed levels of the *Saccharomyces cerevisiae* MF ontology at the time of our analyses covers only 40% of the species' known proteins, with BP covering 36% and CC just 21%. To supplement these ontologies, we use a method developed by Jiang et al. called *AVID*—Annotation Via Integration of Data[15].

1.4.1 The AVID Process

AVID is a four-stage process (shown in Figure 1-4) for predicting new GO annotations by integrating several loosely-related genomic data sets. In particular, AVID incorporates information about protein pairs generated by:

- High-throughput yeast two-hybrid assays
- Large-scale affinity co-purification and mass spectrometry
- DNA microarray co-expression analyses
- Global protein localization studies
- Paralog analysis based on sequence similarity.

It is important to note that no small-scale experimental data is considered. In the first stage of AVID, each of these genomic data sets, represented as a network, is compared with the existing GO annotations to compute the conditional probability of two proteins sharing a GO annotation given that they are connected by an edge in the genomic network. In stage two, the genomic networks are weighted by their normalized conditional probabilities and multiplied to generate a new network in which each edge weight reflects the relative likelihood that an edge represents a real shared GO annotation. Edges whose weights fall below a particular threshold are eliminated completely. In stage three, the remaining edges, along with the conditional probabilities from stage one, are processed by a decision tree-based machine-learning scheme to further filter low confidence pairs. Finally, in stage four, AVID assigns a GO annotation to each protein based on the annotation of the majority of its neighbors in the finely-filtered network from stage three.

1.4.2 Results and Evaluation of the Technique

The yeast annotations predicted by AVID cover a significant percentage of currently unannotated or coarsely annotated proteins. Combining the original GO networks with the AVID-predicted networks markedly improves the annotation coverage for all three ontologies. The new MF network covers 69% of known proteins (up from 40%), BP covers 59% (up from 36%), and CC covers 57% (up from 21%). After combining AVID and GO, 80% of known proteins are assigned an annotation in at least one of the three ontologies.

Importantly, the AVID-predicted annotations also appear to be very reliable. Self-consistency tests indicate that AVID's predictions for currently unannotated proteins are correct 71%, 59%, and 71% of the time respectively for the MF, BP, and CC ontologies. Furthermore, 75%, 80%, and 87% of AVID's refined predictions for MF, BP, and CC, respectively, are consistent with the coarser existing annotations. In many instances where AVID predictions do not match existing GO annotations (and are, therefore, counted as incorrect) they are highly related to current annotations or are experimentally accurate according to literature reports.

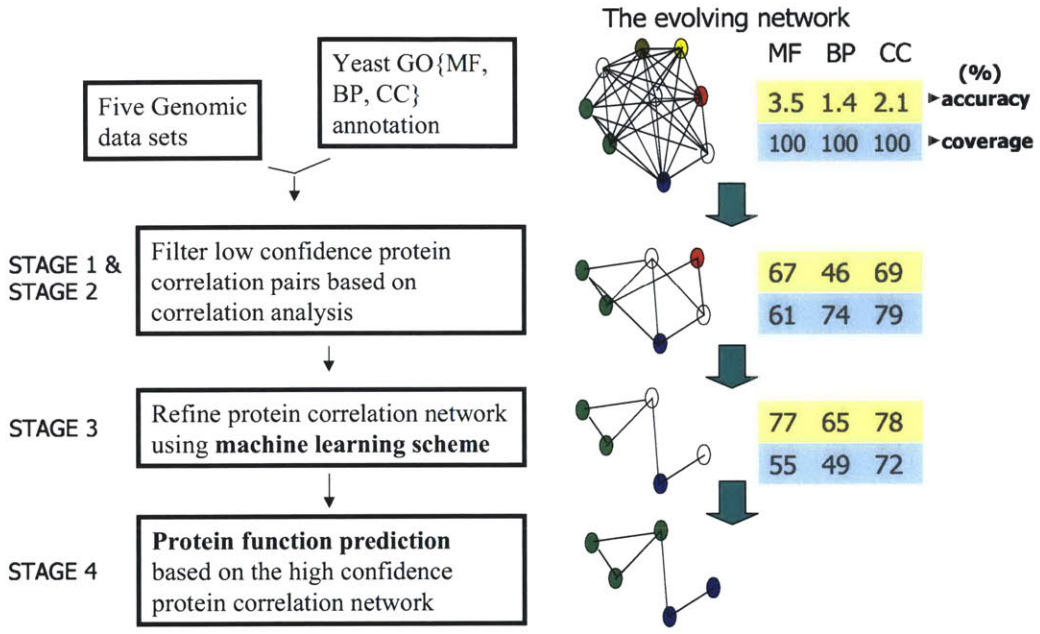


Figure 1-4: The AVID functional prediction process. AVID predicts functional annotations of proteins through a four-stage process. Colored nodes in the illustrated networks represent annotated proteins. In the first two stages, a product of conditional probabilities is used to filter weakly correlated protein pairs from the complete protein correlation network. Stage three employs a decision tree-based machine-learning scheme to further filter the lowest-confidence pairs. Finally, the remaining edges are used to transfer GO annotations (and node coloring) to all involved proteins. At each stage, the accuracy of edges improves as judged by tests on known proteins, at a modest cost in coverage. Figure courtesy of T. Jiang.

In Chapter 3, we use network comparison techniques to investigate the correlation between protein-protein interactions and (1) original GO annotations, (2) new AVID-predicted annotations, and (3) GO annotations supplemented with the AVID predictions.

1.5 Structural Motifs and the Coiled Coil

In addition to the protein properties embodied in GO annotations, we expect that certain structural properties may also correlate with protein-protein interactions. Many protein tertiary structures can be broken into modules called "motifs" or "domains". Proteins frequently contain one or more of these motifs. Furthermore, many of these motifs are known to bind specifically and directly to certain other motifs. Therefore, it is reasonable to expect that pairs of proteins containing compatible binding domains would be more likely to interact with one another than two random proteins.

While there are a wide variety of these known structural elements, one relatively simple and well-characterized motif is the coiled coil. We estimate that this motif exists in at least 4-5% of human proteins (Chapter 4). In many cases, coiled coils directly mediate physical protein-protein interactions, and aberrations in coiled-coil structure are known to cause several diseases, including a variety of cancers[9, 6, 29, 24]. Moreover, almost all coiled-coil interaction partners are other coiled coils, and inter-coiled coil interaction specificity is reasonably well-characterized. Finally, a variety of techniques (described later in this section) are available to predict the presence of the structural motif entirely from protein sequence with considerable accuracy[16, 5, 31, 8]. Together, these properties make the coiled coil an ideal candidate motif for correlating structure with interaction.

1.5.1 Physical Properties of the Coiled Coil

Coiled coils are composed of multiple α-helices wound around one another to create a superhelical twist. Most coiled coils have been shown to consist of either two helices (dimers) or three (trimers), although a few cases of higher-order coils have been

observed. Importantly, the superhelical twist of the coiled coil forms a characteristic seven-residue structural repeat, or *heptad*. The positions, or *register*, (designated as **a** through **g**) define the location of each residue with respect to the other α-helices involved in the motif. For example, positions **a** and **d** are buried within the helical bundle while **e** and **g** are exposed on the outer surface. As is generally the case with protein folding, the buried residues are usually highly hydrophobic in nature while exposed residues are predominantly charged and polar.

1.5.2 Computational Identification of Coiled Coils

It is implausible for researchers to visually inspect each of the more than 2,300 known protein structural families[22] for the presence of a coiled coil. Even if such a project were undertaken, it is often difficult to distinguish short coiled coils from similar, non-coiled helical packings by eye. As a result, several computational prediction programs have been developed to assist in coiled-coil identification.

One program, SOCKET, can fairly accurately locate coiled coils in proteins for which high-resolution structures are available[30]. SOCKET examines the α-helices in proteins and evaluates the physical packing of their amino-acid side chains. However, such structures have been solved for only a small percentage of proteins, reducing the usefulness of SOCKET in proteome-wide identification of coiled coils.

Fortunately, several other programs have been developed that detect coiled coils based solely on the amino acid sequence of the proteins in question. Such information is generally available for all proteins in the growing number of organisms whose genomes have been completely sequenced. Amino acid sequences are also known for many other proteins that have been studied experimentally, but for which no structure has been solved.

The most popular sequence-based prediction programs include COILS[16], Paircoil[5], Multicoil[31], and MARCOIL[8]. COILS, the first such program, works by comparing a sliding 28-residue window with a database of single-residue frequencies in known coiled-coil sequences to identify regions of proteins most similar to known coils. Paircoil significantly extends this concept by scoring all *pairs* of residues in a sliding

30-residue window with a similar database. Multicoil adds to the Paircoil program the ability to detect trimeric coiled coils and distinguish them from dimers. Finally, MARCOIL, the most recent program, uses a windowless hidden Markov model to elucidate likely coiled-coil regions. The performance of these methods appears highly-dependent on the databases used to train the algorithms. As a result, our internal testing indicates that COILS and MARCOIL most accurately detect very long, well-structured coils whereas Paircoil and Multicoil are more versatile and detect smaller, less-obvious coiled coils more often than the others. Additionally, we find that COILS generally produces a large number of false-positive predictions

In Chapter 4, we discuss the creation of a Coiled-Coil Database based on the results of these sequence-based predictions. Using FNAC, we can construct networks representing the presence of coiled coils in proteins. In the future, we plan to integrate these networks with protein-protein interaction and annotation networks to identify a set of previously uncharacterized interactions likely to occur and be mediated by coiled coils.

Chapter 2

A Framework for Network Analysis and Comparison

This chapter details FNAC, our computational Framework for Network Analysis and Comparison that we use to investigate correlations between a variety of protein networks in Chapter 3. Here we include discussion about the motivation for FNAC and its capabilities, the core Java interfaces and abstract classes that can be used to quickly create new network analysis and comparison routines, and the set of such routines already implemented as part of the framework.

2.1 Motivation and Capabilities

FNAC was created to provide a general, powerful, and intuitive platform for examining relationships between large numbers of proteins. Networks are a particularly suitable representation of such relationships because most properties of proteins can easily be represented in such a format and also because such a format can be readily interpreted by the human eye. For example, one of the properties we investigate is the GO Cellular Component (CC) annotation. This annotation can be represented for an entire genome by constructing a graph in which each protein is represented by a node and an edge is drawn between every protein that shares a CC annotation. The result is a graph consisting of several completely connected clusters that

occasionally intersect at proteins annotated to exist in multiple cellular components. Construction of the network is straightforward and, when intelligently layed out, its visual representation is physically intuitive.

Furthermore, the representation of many different protein properties in a common format allows FNAC to provide a substantial, shared foundation for the easy implementation of many analysis and comparison techniques. FNAC greatly simplifies creation of such operators by removing from the developer the burdens of network import and export, network representation, logging, and configuration. This allows the developer to focus on the analysis algorithm itself.

FNAC is provided as a set of Java packages that provide this foundation as well as others that implement several common analysis and comparison techniques that we found useful. In particular, it includes classes for analyzing and comparing networks on the basis of their nodes' connectivity, pairwise relationships (edges), and network motifs—all of which are demonstrated in Chapter 3. FNAC does *not* provide any functionality for visualizing networks, but its file format and internal graph representation (using the Graph INterface librarY, GINY) are compatible with the more general bioinformatics platform Cytoscape[26]. This allows FNAC networks to be visualized and further analyzed within Cytoscape. Moreover, it simplifies the creation of Cytoscape plugins based on FNAC.

2.2 Interfaces and Abstract Implementations

The core FNAC package consists only of interfaces and abstract implementations that can be used to construct network analyzers, comparators, and integrators. Details are available through the JavaDoc documentation in Appendix A, but this section aims to provide an overview of the package and some of the software engineering considerations involved with its design.

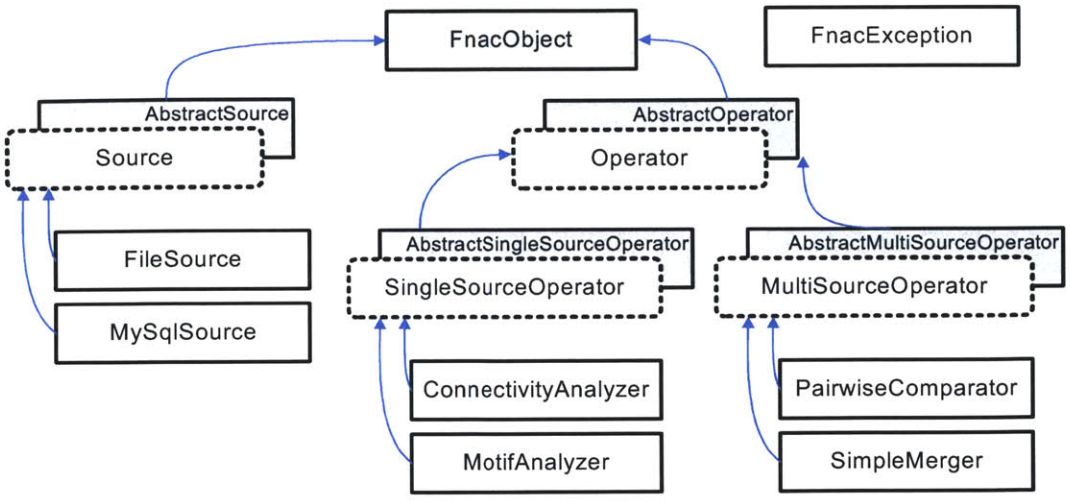


Figure 2-1: FNAC class and interface heirarchy. FNAC is designed to simplify creation of new network sources and operators by allowing inheritance of core functionality. This hierarchy illustrates how provided utility classes like `FileSource`, `ConnectivityAnalyzer`, and `PairwiseComparator` make use of this inheritance. White boxes with solid borders represent concrete classes, while dashed boxes signify interfaces and grayed boxes indicate the abstract classes that implement them.

2.2.1 Common

At the root of the inheritance trees (shown in Figure 2-1) lies the `FnacObject` and `FnacException` classes. Together, these provide the functionality shared by all FNAC-related classes. `FnacObject` currently provides only a common logging mechanism whereby output and error streams can be passed back and forth or shared between multiple FNAC classes. However, the existence of such a completely shared abstract class allows for easy future expansion of FNAC, at even the most fundamental levels. The `FnacException` class extends Java's built-in checked Exception class and provides a basis for distinguishing all FNAC-related exceptions from those exceptions arising from other codebases.

2.2.2 Data Sources

While an excellent graph representation package, the GINY library lacks many helpful features that can be added with the construction of a simple wrapper around

a GINY graph. The `Source` interface and it's associated `AbstractSource` abstract implementation act as just such a wrapper. First, they allow textual naming of each data source. Second, they provide efficient binding of protein names to nodes in the graphs. Third, they can be extended to associate networks with a persistent storage location (such as a text file or a database table) and facilitate loading from and saving to that location.

One of GINY's few major weaknesses is its lack of efficient node naming. In general, GINY clients can address and manipulate graphs either by index or by object reference. Every node and edge in a graph is assigned an integer index and, as such, can be accessed via this index quickly and with minimal memory overhead. Alternatively, a `Node` object can be created for each node and an `Edge` object can be created for each edge. These objects can then be assigned names and used thereafter to refer to each of the nodes and edges. While the latter method offers the abstraction and extensibility benefits of a traditional object-oriented approach, it also incurs a sizeable memory and performance overhead, especially when networks are as large as whole-genome protein interaction networks or when multiple networks are resident in memory simultaneously. Even more unfortunately, this object-oriented method is required to assign textual names to nodes within the GINY framework.

The `Source` wrapper offers a solution to the node-naming problem by implementing a simple name-to-index mapping. Each network is associated with a single map associating every node index with the name of the protein represented by that node. Thus, GINY graphs can be addressed by index **and** names can be assigned to nodes without the extreme overhead of the completely object-oriented approach. Moreover, the map is implemented using two unidirectional hashtable-based maps, allowing efficient lookups both by index and by name.

Finally, concrete implementations of the `Source` interface require methods for loading and saving of the network from/to persistent storage. Since these are the only two methods that need to be implemented to extend `AbstractSource`, new data sources can be supported relatively easily. Two such sources, `FileSource` and `MySqlSource` are provided as part of the `fnac.sources` package.

2.2.3 Operators

The primary function of FNAC, to facilitate analysis and manipulation of the data sources described above, is encapsulated by the `Operator` interface and other interfaces inheriting from it. This interface, along with its abstract implementation `AbstractOperator`, extends the logging functionality of `FnacObject`, providing a model in which each object implementing `Operator` can be thought of as an operation to be performed on a network or a set of networks. Every operation can be supplied a flexible set of options, started, and queried for completion. After the operation has completed, a client can interrogate the `Operator` for results in an implementation-dependent manner, but this functionality is beyond the scope of the simple `Operator` interface.

Because this interface is very general, management of an operation's target data sources is left to two child interfaces as described below.

SingleSourceOperator

The `SingleSourceOperator` interface extends `Operator` to provide a complete channel for clients to associate a single data source with an operator. Many implementations of `Operator` may be designed to support only single data sources. In such cases it is often convenient to have this restriction enforced by the interface to the class.

MultiSourceOperator

The `MultiSourceOperator` interface extends `Operator` to support the general case where many data sources can be associated with a single operation. Such support is required for all network comparators, integrators, and other operators which, by their very nature, necessitate consideration of more than one network. In addition, it is often beneficial for even simple analyzers that operate on a single network at a time to implement `MultiSourceOperator`. Doing so allows a single instance of the operator to analyze many networks at once with a common set of options and a common memory context.

2.3 Provided Packages

This section describes a set of packages we've developed on top of the FNAC core components for the types of analyses we conducted in Chapter 3. These packages include data source connectors as well as operators based on node connectivities, pairwise (edgewise) relationships, and network motifs. Overviews of these tools are provided below and detailed JavaDoc documentation is included in Appendix A.

2.3.1 `sources`

The `fnac.sources` package provides concrete implementations of the `Source` interface for two different types of data sources. The first, `FileSource`, allows networks to be loaded from and saved to what Cytoscape refers to as "raw interaction files." These are simple text files where each line represents an edge in the network and consists of two tab-delimited node names. The text files' locations are represented as Java `File` objects, extending those objects' multi-platform support to FNAC clients.

The second concrete `Source` implementation, `MySqlSource`, allows networks to be loaded from SQL queries on MySQL database tables. Queries are specified by the hostname of the server, the TCP port number of the listening MySQL service, the database to be used, the SQL `SELECT` statement to be issued, and, optionally, a username and password. The result set of the specified query is processed by treating the first two columns as names of interacting nodes, just as in the raw interaction file above. Currently, `MySqlSource` does not support saving of networks into MySQL databases and throws an `UnsupportedOperationException` when such a save is attempted. This functionality may be added in the future, however, by allowing a client to specify the SQL `INSERT` command to be executed.

2.3.2 `connectivity`

The `fnac.connectivity` package provides a concrete implementation of the `SingleSourceOperator` interface based on simple analysis of node connectivities. This class, `ConnectivityAnalyzer`, examines the connectivity of every node in its

associated network. Based on this analysis, it can provide the average connectivity, per-node connectivities, connectivity distributions, and a list of the hubs in the network (where we define a "hub" to be any node whose connectivity is more than twice the network average).

2.3.3 pairwise

The `fnac.pairwise` package provides two concrete implementations of the `MultiSourceOperator` interface based the connections between individual pairs of nodes. The `PairwiseComparator` operator compares a set of networks, providing methods for retrieving the number of overlapping nodes, the number of overlapping edges, and overlap percentages. The `SimpleMerger` class—which is actually used by `PairwiseComparator`—can generate all possible intersections among a set of networks.

2.3.4 motifs

The `fnac.motifs` package is substantially more complex than any of the other FNAC packages, as it considers graphs in the context of network motifs. While this package contains supporting classes (i.e., `MotifShape`), it currently contains only a single concrete implementation of `SingleSourceOperator` called `MotifAnalyzer` that detects and remembers every motif up to a specified size within a single network. After analysis, this operator provides methods for iterating over all motifs, querying for the presence of a motif among a specific set of nodes, and retrieving the number of occurrences of motifs of different shapes.

Unlike the previous packages, the implementation of this package is not straightforward. Below we provide a summary of the data structures, algorithms, and performance considerations involved with `fnac.motifs`.

Data Structures

Representation of the shape of network motif was the first major issue addressed in the design of this package. Previous implementations such as MFinder have assigned shapes numerical identifiers on the basis of their adjacency matrices. This required that all such shapes had to be enumerated at prior to execution and "hard-coded" in the program for the purposes of pattern matching. Obviously, this greatly restricts the size of motifs that can be detected. Moreover, the nodes of a motif can be reordered in the adjacency matrix without impacting the logical shape of the motif. It follows that each logical shape can be represented by a number of different adjacency matrices—one per linear permutation of the involved nodes. The number of these matrices for each shape grow exponentially with the size of supported motifs, hampering this method's scaling to motifs greater than four nodes.

The alternative representation we developed for this package involves the notion of sub-motifs or "motif children." Specifically, we propose that each n-node motif can be uniquely identifed by the shapes of the n-1-node sub-motifs. There are n such children, generated by independently removing each node from the original motif. It is precisely the number of occurrences of each shape child that defines the shape of the parent motif. Thus, our shape representation scales linearly, rather than exponentially, with the size of the motif.

The implementation of our representation takes the form of the `MotifShape` class. Objects of this class are instantiated through a singleton-like factory pattern that ensures only one instance of each shape to minimize memory usage and processor time required for object creation. As defined by our representation, each instance of the class contains a count of the number of motif children assuming particular shapes. However, since this structure is defined recursively, a "base case" shape is necessary. We consider three-node motifs to be such root shapes since, in the case of our undirected networks, there are only two possible three-node motifs: partially- and fully-connected. The former are colloquially referred to as "TRI" since the appear graphically as triangles, while the latter are called "ELL" since they look like the

letter 'L'.

Motif storage was a second challenge we encountered during implementation. Even relatively small *Saccharomyces cerevisiae* networks can have tens or hundreds of millions of four-node motifs. Thus, every byte of memory required to store a motif can cost scores of megabytes over an entire network. In light of this situation we store each motif in a hash-based map of node sets to `MotifShape` objects. Because each object reference is effectively a single memory address, storage of the node sets is the most expensive factor. To minimize the storage costs for node sets, we represent each set as a primitive array of `shorts` with each entry representing a node's index in the network. The use of `shorts` instead of `ints` restricts the use of `MotifAnalyzer` to networks of no more than 32,768 nodes, although this could be expanded to 65,536 by utilizing both positive and negative index values. Using `shorts` also reduces the size of each node set array, although exact savings are platform-dependent and vary with the size of motifs considered. Further savings could be achieved by reducing the overhead associated with the Java array construct, either by creating a Java wrapper around a C construct or by cleverly consolidating all of the node sets into a single, large Java array.

Motif Detection Algorithm

The algorithm we've developed for enumerating network motifs can be broken into two consecutive stages: set detection and shape identification. The former employs a simple breadth-first search, modified to prevent revisitation of node sets, to identify all sets of connected nodes up to a client-specified motif size. Then, after all node sets have been identified, the shape of each set is determined, starting with the three-node ("base case") motifs. Once all three-node sets have been assigned a shape, four-node shapes can be determined by performing a series of lookups of three-node shapes. The process continues until the largest motifs have been assigned shapes.

Realistic Performance Considerations

Unfortunately, even with careful design and implementation, the problem of motif detection still scales exponentially with network size and density. While this does lead to lengthy computations, the limiting factor for our algorithm is memory availablity. For example, some *Saccharomyces cerevisiae* networks can contain approximately 5,000 nodes and 25,000 edges. Detection of three- and four-node motifs for such a network may only take a few hours on a single currently-high-end CPU. However, the detection requires more than 3 GB of memory, much more than is often available for such a computation. One obvious (and naive) solution is to write all the motifs to disk as they are found. Unfortunately, since the algorithm relies on frequent accesses of the node-set-to-shape map, such a solution degrades performance by *many* orders of magnitude.

Our solution to this storage problem involves splitting the large motif map into many smaller maps. By intelligently dividing the map, first by motif size and then by the lowest node index in each node set, we can effectively concentrate successive map lookups within a single submap. Moreover, we store each map on disk but maintain a cache of recently accessed maps in memory. The cache grows with available memory and implements a least-recently used (LRU) replacement policy. Disk copies of the maps are only updated when the map is dropped from the cache and at the end of operator execution. Through this technique, we effectively maximize memory usage and performance without restricting support for large or dense networks.

Chapter 3

Applying FNAC to Annotation and Interaction Networks

In this chapter we apply several of FNAC's functionalities to the analysis and comparison of several protein annotation and protein-protein interaction networks introduced in Chapter 1. Specifically, we begin by investigating these networks on an individual basis to develop a better understanding of the nature of their contents. Then, we examine how different categories of annotation correlate with one another as well as with known protein-protein interactions.

3.1 Introducing the Data Sets

In total, this chapter describes the evaluation of 19 distinct networks. Logically, we separate these networks into two categories: *annotation sources* derived from functional descriptions of proteins and *interaction standards* derived from actual interaction experiments. The 15 annotation sources can be further divided into five sets of networks:

- *GO* - One network derived from each of the three GO ontologies (MF, BP, and CC) for the yeast *Saccharomyces cerevisiae*. Only proteins classified at the most detailed levels in the ontologies are included in the network, and edges

are formed in the network only between proteins that share these most-detailed annotations.

- *AVID-GO* - AVID predicts its own MF, BP, and CC annotations from a variety of data (described in Chapter 1). AVID networks consist of edges between proteins that share AVID-predicted annotations. Only proteins present in the corresponding original GO networks are included in AVID-GO. Since AVID-GO networks contain only *AVID-predicted* annotations, these often differ from GO annotations of the same proteins.

- *AVID-New* - Derived from the same AVID-predicted annotations as AVID-GO, but proteins are only included in the networks if they were *not* described in the corresponding original GO ontology. Thus, AVID-New contains predicted annotations of only previously unannotated proteins.

- *GO Plus AVID-New (GPAN)* - The union of GO and AVID-New. These networks can be thought of as the original GO networks supplemented with AVID's predicted annotations for proteins not covered by GO.

- *AVID-GO Plus AVID-New (AGPAN)* - The union of AVID-GO and AVID-New. These networks amount to the complete AVID-predicted MF, BP, and CC ontologies.

Each of these five sets consists of an MF, BP, and CC annotation network. As described in Chapter 1, nodes in these networks represent proteins and each edge represents an annotation common to both involved end points.

In addition to the annotation sources, we also consider four interaction standards. These are protein-protein interaction networks described by DIP[32], DIP-CORE[7], and both the "Gold-Standard Positives" and "Gold-Standard Negatives" of Jansen et al[12]. The DIP network represents a fairly large set (15,160) of direct interactions suggested by a variety of experimental assays, both high-throughput and small-scale. DIP-CORE is a subset of DIP consisting only of those interactions that are supported by small-scale experimental evidence, have known paralogous interactions, or appear

in at least two independent studies. The Jansen positive network is derived almost directly from the MIPS complexes catalog[19], while the negatives are constructed from proteins reported to localize to different subcellular compartments. For efficiency in our analyses, we selected a random subset of 200,000 of the 2,700,000 edges in this set of Jansen negatives.

3.2 Connectivity Analysis

Before inspecting how all of these data sets compare with one another, it is worthwhile to examine each individually in order to gain further insight into the data it represents. We begin by considering simple connectivity statistics as introduced in Chapter 1. In particular, we generate the connectivity distribution and average connectivity for each graph.

3.2.1 Methods

All of the analyses in this section were performed using the FNAC Java packages. Each of the data sets, represented on disk as raw interaction files, was loaded into memory as an instance of `FileSource` and one instance of `ConnectivityAnalyzer` was created. Each network was then passed to the `ConnectivityAnalyzer` via a call to the latter's `setSource()` method and analyzed by an invocation of `operate()`. The connectivity distribution of the network was obtained through a call to the analyzer's `getConnectivityDistribution()` method and the average connectivity through `getAverageConnectivity()`.

3.2.2 Results

The connectivity properties observed for each of the networks are consistent with the manner in which the graphs were constructed. Each network's statistics are described in detail below.

Annotation Sources

All three GO networks (MF, BP, and CC) exhibit extensive clustering, as shown in Figure 3-1(a). Instead of the exponentially decreasing distribution typical of the scale-free "biological" networks, these distributions take the form of a collection of discrete peaks. There is a slight decaying trend at very low connectivities, but this is generally dominated by spikes of linearly-increasing magnitude. Each of these large peaks represents a single, fully-connected cluster of proteins sharing a particular annotation, while the much smaller peaks at high connectivities represent proteins with multiple annotations and, therefore, connections to multiple clusters. From the average connectivities, ranging from 28.2 to 75.6, we can infer the typical number of proteins sharing an annotation. However, we can also see from the large peaks that there are a few annotations shared by many proteins (as many as 300 in MF).

In contrast to their GO counterparts, the AVID-GO networks exhibit largely decaying connectivity distributions (Figure 3-1(b)). While a few indications of clustering appear in the MF and BP peaks at connectivities as high as 80, an exponential decrease uncharacteristic of annotation networks dominates the distribution. This effect, which reduces the average connectivities to between 8.0 and 17.2, results from the generally sparse nature of the networks used by AVID to make its predictions. None of the AVID training networks are nearly as clustered as GO.

Even more atypical of annotation networks, the AVID-New graphs exhibit almost pure exponential decay (as shown in Figure 3-1(c)) and average connectivities between 8.8 and 13.1. This is consistent with the fact that these are not complete annotation networks, but rather supplements to other annotation networks. As such, these lack the majority of the edges in each shared-annotation cluster. This distribution is also consistent with the exponentially-decreasing nature of the data sets used by AVID to generate this network of novel and refined predictions.

Finally, the connectivity distributions of both the GPAN and AGPAN networks (Figure 3-1(d-e)) are, unsurprisingly, consistent with their constituent graphs. That is, the GPAN sets exhibit slightly more exponential decay than their GO components

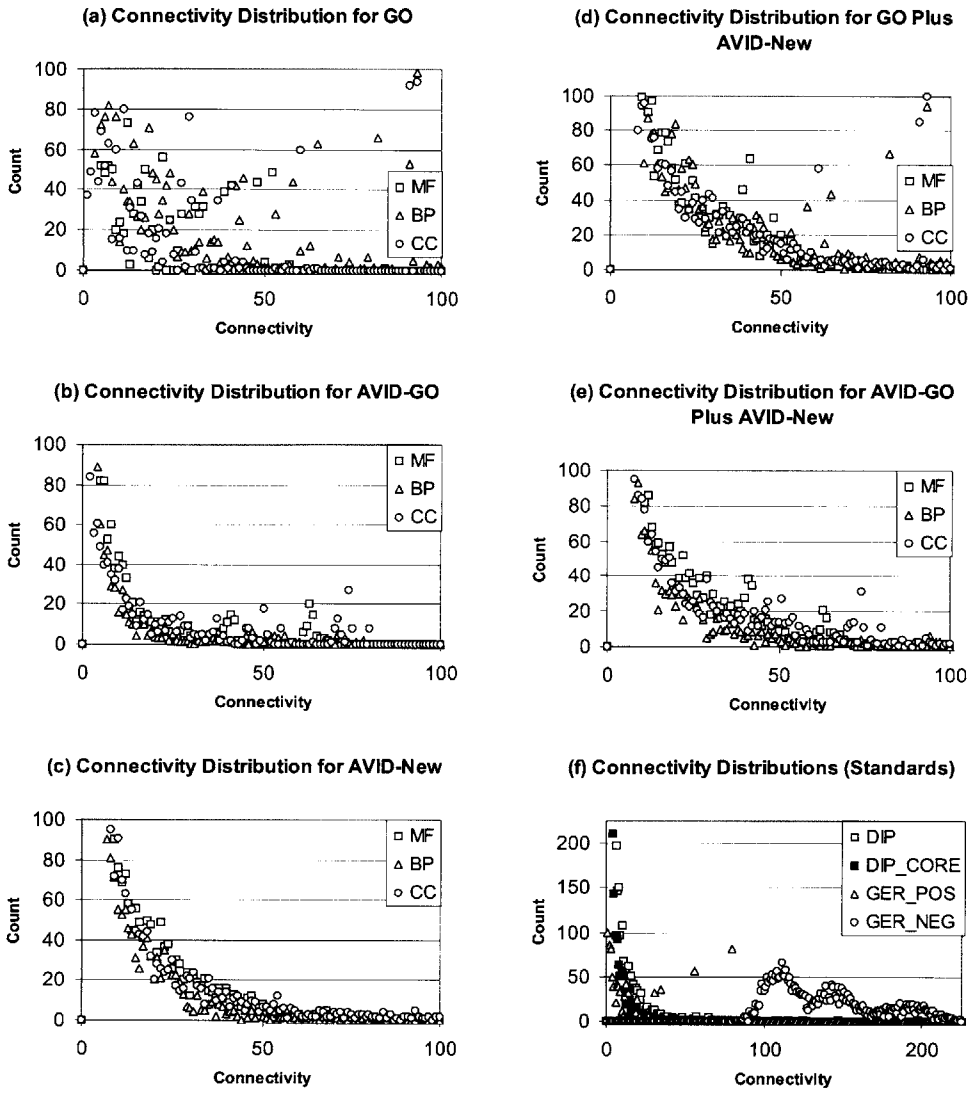


Figure 3-1: Connectivity distributions of source and standard networks. **(a)** The distribution of node connectivities for the three original GO networks (MF, BP, and CC) show sporadic peaks along the line $y = x$, indicating dense clustering. **(b)** Although significantly less clustered than their GO counterparts, the AVID-GO networks retain some element of clustering as evidenced by the substantial signal at connectivities as high as 75. **(c)** AVID-New networks exhibit purely exponential decay and minimal clustering. **(d-e)** Both the GO and AVID-GO networks retain much of their distinct appearances when supplemented with the AVID-New networks. **(f)** The interaction standards show very unique connectivity distributions. While DIP and DIP-CORE demonstrate sharp exponential decays, the Jansen gold-standard positives contain several large, dense clusters, and the Jansen gold-standard negative network is extremely dense on the whole.

but retain most of the cluster-induced peaks, especially at higher connectivities. Similarly, the distributions of the AGPAN networks largely mimic those of the AVID-New networks, with the addition of a few peaks corresponding to the most significant peaks in the AVID-GO networks. The average connectivities of both data sets also fall between the average connectivities of their components.

Interaction Standards

As illustrated in Figure 3-1(f), both the DIP and DIP-CORE interaction networks exhibit the classic exponential decay and low average connectivity expected of scale-free biological networks. The average connectivity of DIP is just 6.4 and that is further reduced to 4.7 for DIP-CORE. Moreover, both distributions peak with more than 25% of their proteins carrying a connectivity of one, while just 3.10% of DIP and 0.80% of DIP-CORE proteins have connectivities above 30.

Unlike the previous two networks, the Jansen Gold-Standard Positive graph more closely resembles a clustered annotation network than an interaction network. The average connectivity (18.9) is significantly higher than that for DIP and, although the most common connectivity is still just one, only 11% of proteins are connected in such a fashion. Furthermore, the connectivity distribution shows a pattern of increasingly large peaks, not unlike those seen in the GO Annotation Source. All of these observations are consistent with the fact that, in essence, this interaction standard actually is an annotation network similar to GO CC. It doesn't represent known direct interactions, but rather a shared complex assignment which is inherently clustered in nature. Connectivity analysis alone demonstrates this network's poor suitability as our "gold-standard" for *direct* protein-protein interactions.

Our subset of the Jansen Gold-Standard Negative network is fundamentally different than all of the previous networks. This graph is not scale-free or even highly clustered. It is simply dense, as expected, since most protein pairs do not interact. The average connectivity is 137.8 and even the *least* connected of the 2,702 nodes neighbors 81 other nodes. Moreover, the distribution resembles nothing we have previously encountered. Instead, it appears tri-modal, with three very broad peaks

occurring around the connectivities of 100, 145, and 195.

3.3 Correlation Between Sources

Now that we have established a firmer understanding of the inherent properties of the individual networks, we can begin examining how they compare with one another. In the next section we will discuss how the annotation sources correlate with each interaction standard, but this section focuses on the relationships among the MF, BP, and CC networks within each of the five annotation sources. Although these three networks are purported to describe distinct properties, we expect them to be biologically related. For example, intuition suggests that proteins involved in the same biological process should have a higher probability of localizing to the same cellular component than two random proteins. Moreover, the three ontologies may well be derived from the same data sources and, thus, not independent. This codependence is certainly true for the AVID predictions since the same training networks were incorporated into the prediction of all three ontologies, albeit with different weights.

3.3.1 Methods

For each pair of networks we examined, we computed the number of nodes common to both networks, the number of edges in each network completely incident on the common nodes, and the percentage of each network contained in the other—all with a few very simple calls to FNAC.

As in the connectivity analysis, each source network is first loaded into memory as an instance of `FileSource` and a single instance of `PairwiseComparator` is created. We then loop over each set of annotation networks and, within each, all three possible pairs of networks (MF/BP, BP/CC, and MF/CC). At each iteration, `PairwiseComparator`'s `setSources()` method is used to specify the current pair of networks to be analyzed and the `operate()` method is invoked to perform the pairwise comparison. Upon completion, the intersection of the two source networks is obtained through a call to `getOverlappingSource()` and we count the nodes and

edges (E_{both}) in the resulting graph, yielding the number common to both networks. The number of edges, $E_{i,comm}$, in each source incident on the common nodes is then obtained through two calls to `getEdgeCountOnCommonNodes()`. From the count of the overlapping edges and edges on common nodes, we compute the percentage, P_i, of network i's common node edges in the overlap as follows:

$$P_i = \frac{E_{i,comm}}{E_{both}} \tag{3.1}$$

3.3.2 Results

The observed correlations within each set of annotation sources vary from less than 20% (in the case of the AVID-New CC and BP networks) to nearly 100% (for the AVID-GO MF and CC). Among the data (shown in Figure 3-2), we believe several results are particularly helpful in understanding the nature of the networks and discuss these below.

First, we observe a very strong dependency of the GO CC network on GO MF. That is, for the 728 proteins described in both ontologies, the CC network is almost entirely a subset of MF. Interestingly, though, these CC edges comprise less than a third of the MF network. These results seems to indicate that, for the proteins described in both ontologies, almost all CC annotations are derived directly from the MF annotatons. Notably, AVID discards 91% of MF edges but just 55% of CC edges when constructing the AVID-GO networks. Among the 589 proteins that are still annotated in both of these ontologies, the MF network is now almost entirely a subset of the CC! One likely explanation for this role reversal is the emphasis AVID places on MIPS complex formation when predicting both MF and CC. Intuitively, one would expect MIPS complexes to correlate well with GO CC annotations since both are purported to represent the same type of data. Thus, MF and CC edges that overlapped would have a higher retention rate than those that didn't. Indeed, the data shows just such a trend.

Second, we notice that in GO and, to a lesser extent, AVID-GO, the BP and CC networks are surprisingly similar. Even though, on the whole, the GO BP network

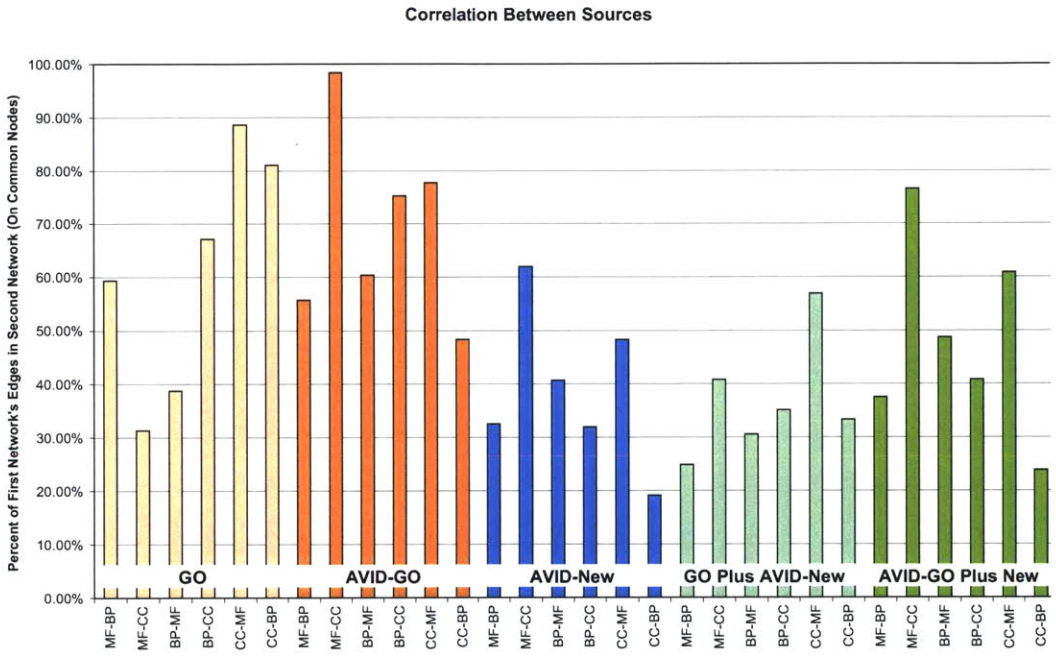


Figure 3-2: Correlation between the three networks (MF, BP, and CC) of each source category. Each of the five source categories shows varying degrees of correlation among its three networks. Interestingly, both GO and AVID-GO exhibit significant similarity betweeen their MF and CC networks.

consists of nearly twice as many nodes and edges as CC, edges between those proteins in both ontologies are highly correlated. To be precise, 6,398 edges overlap among the 9,526 in BP and 7,888 in CC between common nodes. This unexpected degree of similarity obviously indicates a dependence between the annotations in each ontology, although the directionality of this dependence is less clear.

Third, we observe generally decreased correlation among the AVID-New networks when compared to their GO counterparts. Consistent with the goal of AVID to expand the coverage of GO, we observe significantly higher numbers of proteins involved in each of the AVID-New networks than the raw GO networks. This, in turn, leads to an increase in the number of overlapping nodes and edges between those nodes. On average, each pair of AVID-New networks had 205% more nodes in common than the corresponding GO pair and each network had 58% more edges incident on these common nodes. The number of edges overlapping each pair of networks, however, did not increase proportionally, leading to the observed decrease in correlation. This is not surprising or a sign of low accuracy on the part of AVID, however—AVID networks are significantly less dense and clustered than those of GO, meaning that each AVID-New edge is much more independent of other edges.

Finally, the GPAN and AGPAN combination networks are clearly influenced by both of their constituents. This influence is most lucid in AGPAN, where the correlation between each pair of networks actually lies between the associated correlations in AVID-New and AVID-GO. Although less clear, this dual-influence is also present in GPAN, where the relative rise and fall of correlations roughly mirror the peaks observed in AVID-New and GO. Neither of these results is surprising given that GPAN and AGPAN are each the union of two disjoint data sets.

3.4 Correlation of Sources With Standards

In this section, we move a step closer to predicting new protein-protein interactions by investigating the relationships between our annotation sources and interaction standards. To this point, we have focused on understanding the nature of each of our

19 networks, both through connectivity analysis of individual networks and pairwise comparison of annotation sources. Here, however, we begin to asses the value of each annotation network in predicting known protein-protein interactions. To this end, we correlate each of the networks (MF, BP, and CC) and network intersections (MF∩BP, MF∩CC, BP∩CC, and MF∩BP∩CC) of the annotation sources with each of the interaction standards, using much of the same FNAC functionality as in the previous section.

3.4.1 Methods

As in both previous analyses, all of our networks are represented in memory as `FileSources`. For each set of annotation networks, we create an instance of `SimpleMerger` and pass it all three of the relevant networks. We then invoke the `operate()` method of the `SimpleMerger` to generate the seven possible intersections which we then retrieve through `getAllIntersections()`. We iterate over these intersections and, for each, use a `PairwiseComparator` just as in the previous section to generate overlap statistics with each of the interaction standards. The statistics generated for each comparison include:

- *Common Nodes* - the number of nodes present in both the source and standard networks
- *Trimmed Edges* - the number of edges in each network among the common nodes
- *Overlap Edges* - the number of edges present in both networks
- *Accuracy* - the percentage of trimmed source edges in the set of overlap edges
- *Coverage* - the percentage of trimmed standard edges in the set of overlap edges

3.4.2 Results

The most obvious result of this analysis is that the correlation between our annotations sources and interaction standards varies quite widely by both source and

standard. To simplify presentation, we divide discussion of our observations by interaction standard below.

DIP and DIP-CORE

Consideration of just the GO annotations and DIP interactions is a good starting point for understanding how correlation varies by combination of source networks used. As shown in Figure 3-3, the GO CC network alone is more than twice as accurate in its prediction of DIP as either MF or BP. This is most likely a statistical embodiment of the fact that, at the detailed levels included in our analysis, GO CC effectively represents protein complexes where at least several of the proteins are known to physically interact. Moreover, correlation of any of MF, BP, or CC can be improved, often dramatically, by filtering it with either of the other networks. For example, only 6.79% of trimmed MF and 7.18% of trimmed BP edges are present in DIP. However, this accuracy is increased to 26.79% when edges in both MF and BP are considered. Similarly, an accuracy of 34.52% is seen in the overlap between BP and CC, and the intersection of all three networks yields an impressive 42.61%. Given the incompleteness of DIP, such a high accuracy is remarkable and strongly suggests that GO alone may be a very powerful predictor of interaction.

Also intriguing are the differences in correlation between the GO and AVID-GO annotation sources. First, all seven AVID-GO networks correlate better with DIP than their GO counterparts. This is expected since AVID-GO is predicted, at least in part, based on some of the same high-throughput experiments used to generate DIP. However, the AVID-GO networks also all correlate better with DIP-CORE than those in GO. This result is less expected since, in its prediction, AVID considers none of the additional evidence required of edges in DIP-CORE. One explanation for the improved performance under DIP-CORE is the simple fact that DIP-CORE is a subset of DIP, under which AVID-GO also performs better. This hypothesis does not directly address the fact that AVID-GO accuracy is actually *better* under DIP-CORE than DIP (unexpected if DIP performance is the primary contributor to DIP-CORE performance).

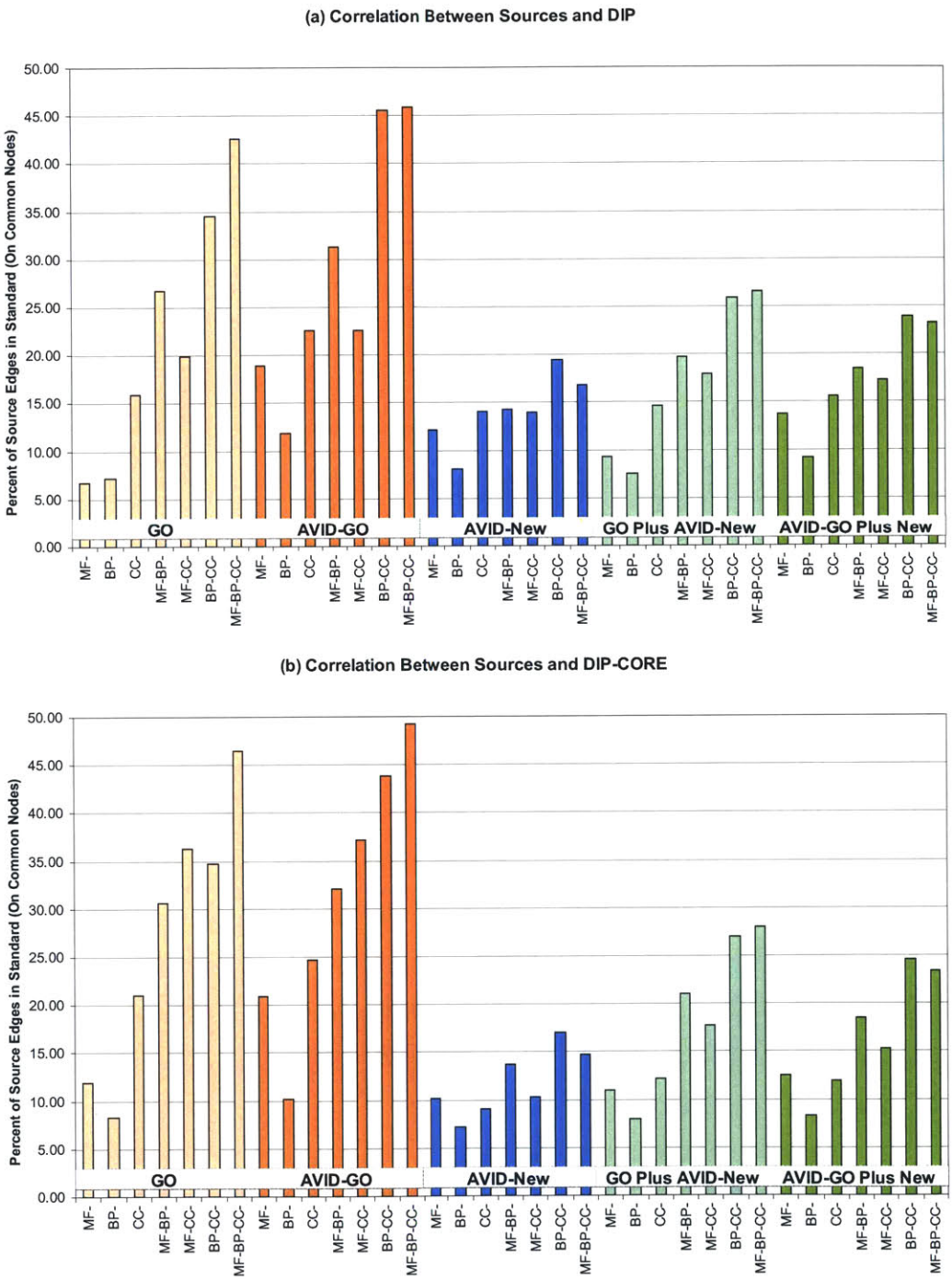


Figure 3-3: Correlation between source networks and both DIP and DIP-CORE. The GO and AVID-GO CC networks correlate particularly well with both DIP networks. In general, the correlation of a source network with the standards is improved by intersection with another source network (e.g., MF∩CC is more accurate than CC alone). Also, accuracy of GO and AVID-GO is higher in DIP-CORE than DIP, while that of AVID-New is actually lower.

This latter observation, however, is consistent with the improved correlation of the original GO networks with DIP-CORE over DIP. The cause of this improvement, though, is uncertain. Since the overlap between each GO network and DIP-CORE is a subset of the overlap between the same network and DIP, the former must be selectively enriched for correlation. This is particularly true for MF and CC, which are enhanced much more in DIP-CORE than BP. One possible explanation is that because DIP-CORE proteins are generally more well-studied, their GO annotations are more likely to be based on high-confidence small-scale experimentation rather than a single, error-prone high-throughput assay. Another possible explanation, particular for the CC networks, is that the reduction in the number of overlapping proteins between CC and DIP-CORE eliminates many of the non-interacting members of the CC complexes. This node reduction, in turn, removes an exponential number of edges from each of the remaining nodes, shrinking the accuracy denominator, and significantly improving correlation.

Despite the obvious ability of AVID to enhance correlation between GO and both DIP standards, the AVID-New predictions exhibit notably less correlation. It is certainly possible that this poor correlation is due to incorrect predictions on the part of AVID. However, given the previous analysis of the method's performance, it seems much more likely to be an embodiment of the fact that proteins newly annotated by AVID are much less well-studied. Since the relationships between these proteins were not examined closely enough to be described in GO, it would follow that they are also unlikely to be described in DIP and, especially, DIP-CORE. Indeed, unlike the GO and AVID-GO sources, the accuracy of AVID-New decreases substantially in DIP-CORE over DIP.

Jansen Gold-Standard Positives

Even the most cursory first glance at the results in Figure 3-4 reveals that *all* of our networks correlate extremely well with the Jansen Gold-Standard Positives network. In particular, GO CC alone displays 97.56% accuracy and 95.15% coverage in predicting the edges of this standard on the 575 overlapping proteins (the standard itself

only contains 871 proteins). Combining GO CC with GO MF further improves accuracy to 99.54% and coverage to 97.04%. These observations are certainly consistent with the fact that the standard was generated by filtering the MIPS complex data, as MIPS and GO CC essentially represent the same data. However, they do also clearly demonstrate that this Gold-Standard Positive network cannot be used as a reliable metric of predictive value.

Jansen Gold-Standard Negatives

As expected and illustrated in Figure 3-4, none of our annotation sources correlate appreciably with the Jansen Gold-Standard Negatives network, connecting proteins previously reported to localize to different subcellular compartments. However, several networks (namely CC and MF∩CC of both GO and AVID-GO) show exceptionally little correlation with the negative standard. This is particularly expected since the standard consists of pairs of proteins localized to wholly different parts of the cell while the CC networks consist of pairs known to group together at the finest level.

3.4.3 Predictive Value

As an extension of the correlation data described above, compute a Bayesian "likelihood ratio," as in Jansen et al., to provide a single measure of the predictive value of our annotation sources[12]. For a given network feature f, this ratio, L, is defined as:

$$L = \frac{\Pr(f|positive)}{\Pr(f|negative)} \tag{3.2}$$

Based on the previously detailed nature of our interaction standards, we define DIP-CORE to be our "gold-standard positive" set and the Jansen Gold-Standard Negatives to be our "gold-standard negative." Therefore, the likelihood ratio for a given network is its previously computed coverage of DIP-CORE divided by its coverage of the Jansen Gold-Standard Negatives. Note the usage here of *coverage* instead of *accuracy*.

Figure 3-5 shows the likelihood ratios for each of our annotation network intersec-

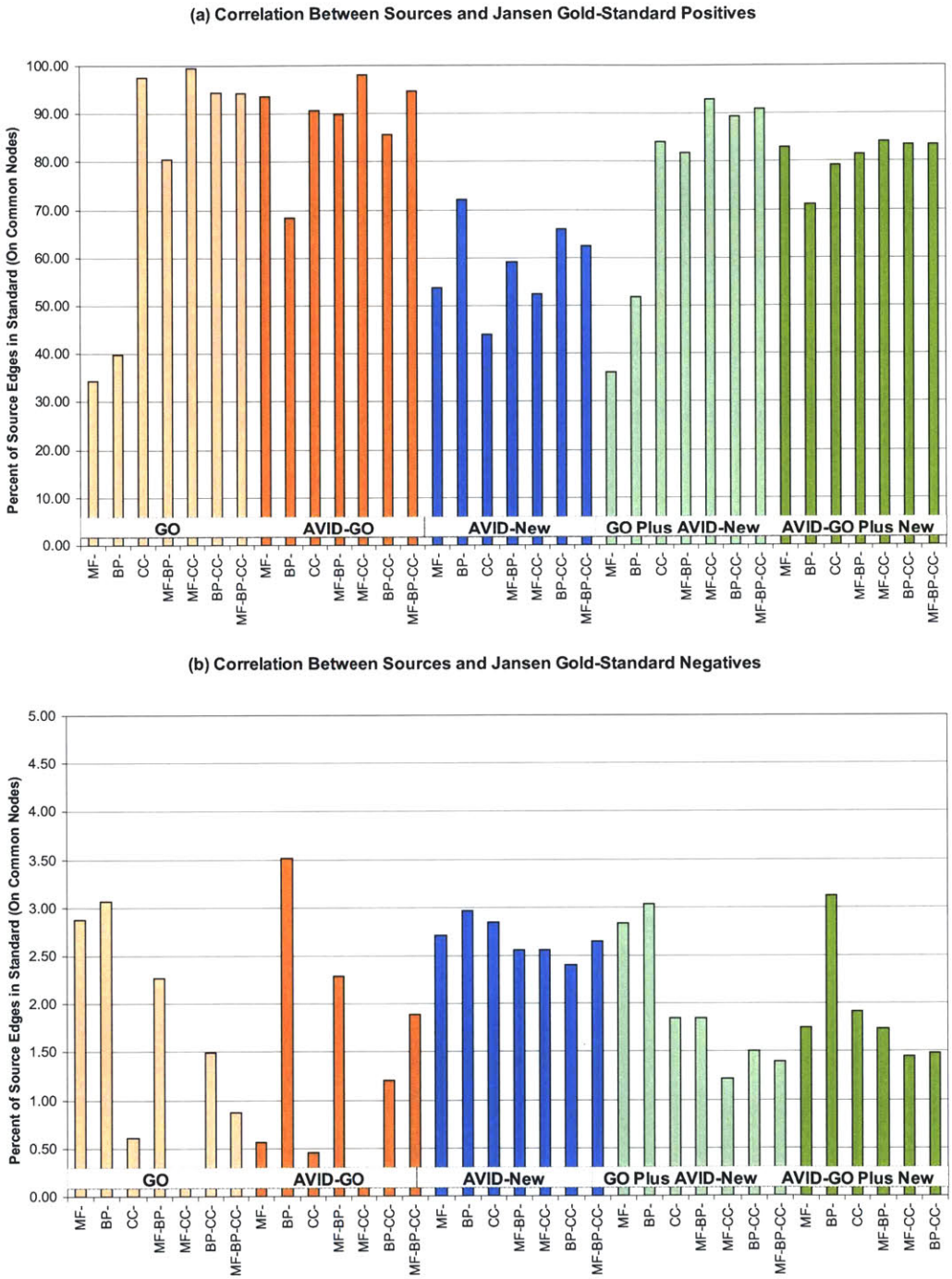


Figure 3-4: Correlation between source networks and the Jansen gold-standards. **(a)** Since the gold-standard positives are derived from MIPS complex data, it is not surprising that they correlate almost perfectly with GO CC. **(b)** As expected, none of the source networks correlates appreciably with the gold-standard negative network.

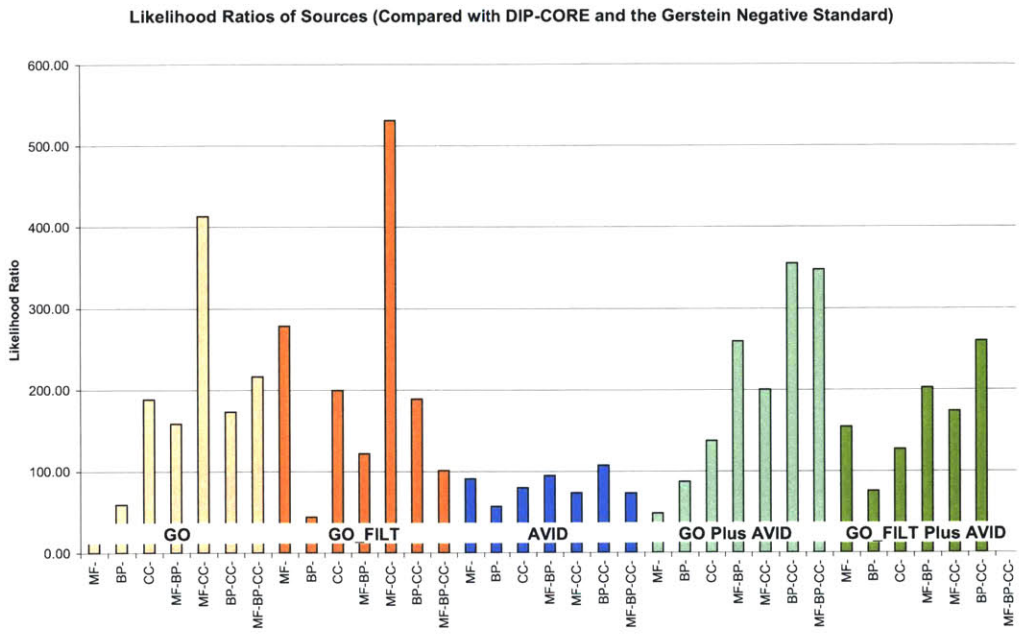


Figure 3-5: Bayesian likelihood ratios for each annotation source's prediction of interaction. The high likelihoods of several GO and AVID-GO networks suggest that these sources may be very useful in the prediction of unknown protein-protein interactions.

tions. Clearly, given our definitions, the best predictor of interaction is the AVID-GO MF∩CC network with a ratio of 531. Intuitively, this means an edge found in this network is 531 times as likely to represent an interaction as not. Even if we discard all CC-based networks because of their known anticorrelation with the gold-standard negatives, significant likelihood ratios are still observed. The GO MF∩BP network, for example, exhibits a ratio of 159, and edges in the AVID-GO MF network are 279 times as likely to represent interactions as not. Although below most of those associated with GO and AVID-GO, the likelihood ratios for the AVID-New networks are still significant, varying between 57 and 107.

From all of the data described above, we conclude both that GO annotations can be highly suggestive of protein-protein interactions and that AVID can successfully enhance the predictive value of the GO MF and CC annotations.

Chapter 4

The Coiled-Coil Database (CCDB)

This chapter describes the Coiled-Coil Database (CCDB), our central repository for information we collect regarding coiled coil containing proteins. It includes discussion on the requirements that led to its creation, the design of database itself, our approach to automating database generation and maintenance, the web service programmatic interface, and the user-friendly HTML front-end.

4.1 Motivation for a Database of Coiled Coils

Because the coiled-coil structural motif is studied by many different research groups around the world and it is also commonly encountered by scientists who aren't expressly studying it, information about proteins containing the motif is often scattered throughout many different resources. Since the Keating lab concentrates many of their efforts on coiled coils, it became clear that a single, easily searchable repository of relevant information would be very useful in supporting a variety of sequence-, structure-, and function-based calculations. In particular, such informations includes textual annotations, Gene Ontology classifications, and cross-references to other online resources for coiled coil-containing proteins.

At the core of the database is information about which proteins contain coiled coils in the first place. Unless a protein's structure has been solved by nuclear magnetic resonance (NMR) spectroscopy or x-ray crystallography, we cannot be absolutely cer-

tain whether a protein contains a coiled-coil motif. In these majority of cases, we rely on sequence-based prediction programs such as COILS[16], Paircoil[5], Multicoil[31], and Marcoil[8]. None of these programs perfectly predicts coiled coils, however, but instead generates scores for the probability of a coiled coil existing in different regions of the protein in question. For a variety of genomic analyses, it can be extremely useful to have quick access to the scores of every region of every protein.

The CCDB allows us to study the properties of coiled coils across entire genomes, rather than focusing on a few particular well-understood families as has been done in the past. Statistics regarding the length of coiled coils, their distribution across different functional families, and their involvement in known biological pathways can be obtained through quick and simple SQL queries. Moreover, computational or experimental studies of protein interactions can potentially be enhanced by extracting coiled-coil information for the involved proteins. As an example, we plan to use the CCDB in conjunction with our network analysis framework to correlate the presence of coiled coils with interaction and to identify potential—but currently unknown—interactions mediated by coiled coils. In the future, this approach can also be extended to other protein structural motifs.

4.2 Database Design

Before we could begin design and construction of the database itself, we needed to select a relational database management system (RDBMS) to serve as our primary platform. Although there are many more sophisticated and higher-performing databases available, we chose MySQL 4.0. MySQL provides good support for most of the recent Structured Query Language (SQL) standards and is well-documented. Compared to other packages, it is also straightforward and easy to learn. Above all, though, it is a very commonly used, freely available, and open-source product, making it ideal for academic use.

4.2.1 Contents

Another key design decision involved in construction of the CCDB was the choice of exactly what content it should contain. On the conservative extreme, we could mine from external databases only information specifically pertaining to coiled coils—if a protein doesn't contain the motif, we wouldn't keep any information about it. Conversely, we could extract (and consolidate locally) every bit of relevant information about every protein we encounter. The former approach would require minimal local storage and creation effort. However, it would also provide only the barest minimum of functionality, precluding complex queries and requiring additional searching of public information repositories for any details. The latter approach, on the other hand, would require significant local storage and extensive maintenance, but would allow for easy queries involving almost any property of a protein. In the end we chose to compromise between the two extremes and include all information that would be regularly queried for large datasets and nothing more.

The data type most central to our work is the protein. We uniquely identify each protein by the species in which it is found and its amino acid sequence. Many websites maintain detailed information for each protein, but much of this data is often duplicated between sites or irrelevant to our research. Therefore, from each site of interest, we chose to incorporate into the CCDB only a simple textual description from each data source as well as enough information to query the website at a later time for more details—namely the "ID" or "accession number" given to the protein by each website.

A second data type critical to the success of the CCDB is the coiled coil itself. Each coiled coil has many associated properties: the protein within which it is contained, its location in the protein, the program that predicted its presence, the program's confidence in its own prediction, and the predicted register of the coiled coil.

Additionally, supporting information about each data source is also incorporated into the CCDB. This includes a textual description of each data source and the URLs required to query each website for more information.

4.2.2 The Table Structure

The schema for this database (shown in Figure 4-1) is designed to maximize flexibility, expandability, and performance while minimizing redundancy and wasted storage. In general, unsigned integers are used as keys for indexing into tables containing more information. For example, in the `proteins` table, we do not store the full species name "Homo Sapiens" (12 bytes) for each entry, but rather just the single-byte numerical identifier of the species. This reduces storage and increases performance, each by an order of magnitude.

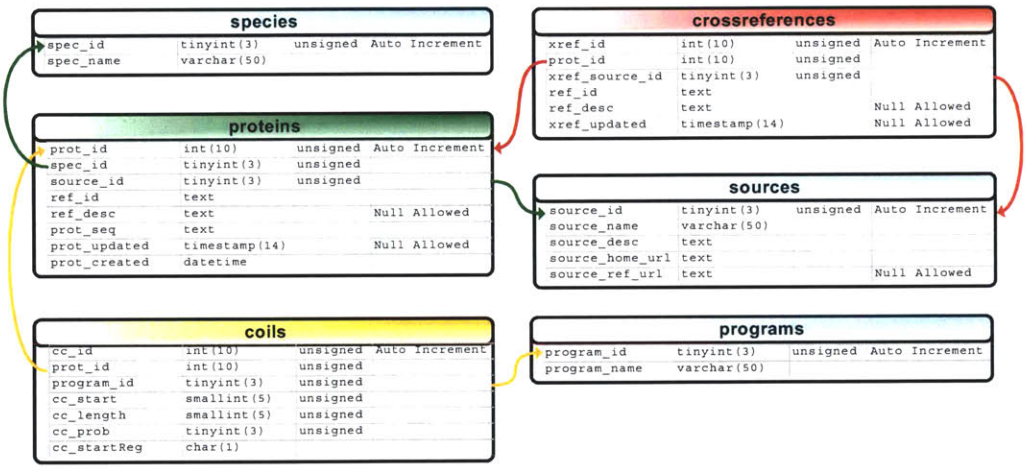


Figure 4-1: The CCDB's table layout and dependency diagram. In general, fields are represented by the most efficient data type that will accommodate foreseeable expansion. Also, references (represented by arrows) are made to other tables wherever possible to minimize redundancy and improve query performance.

The `species` Table

The `species` table is the simplest of all the tables in the CCDB and contains just two columns. The first, `spec_id`, contains a unique integer identifier for each species. The second, `spec_name` provides the scientific name for each species (e.g., "Homo sapiens" or "Escherichia coli").

The sources Table

This table contains information about each data source supported by the database. Currently, this includes only Georgetown's non-redundant protein reference, PIR NREF and NCBI's similarly non-redundant data set, RefSeq, but the design of the database facilitates future additions. Information stored in this table includes a unique integer identifier of the source (`source_id`), the textual name of the source (`source_name`), a textual description of the source (`source_desc`), the URL of the source website's home page (`source_home_url`), and the URL used to obtain more information about a particular protein (`source_ref_url`).

The proteins Table

The `proteins` table is more complicated than either of the previous two tables and relies on them both for proper function. Each protein has a single entry in this table and, as usual, is associated with a unique integer identifier (`prot_id`). The `proteins` table also contains two integer columns (`spec_id` and `source_id`) that reference the `species` and `sources` tables respectively and are used to identify both the species in which the protein has been found and the data source in which we first encountered it. The `ref_id` column contains textual strings corresponding to the identifier of the protein in the associated data source and `ref_desc` contains the source's textual annotation of the protein. Furthermore, the `prot_seq` column encompasses the amino acid sequence of each protein, represented as a string of single-letter codes. This, along with the species identifier, is the piece of information that logically defines a "unique" protein. Finally, the `prot_created` and `prot_updated` fields are used by our generation and maintenance scripts to keep track of which proteins need to be analyzed for the presence of coiled coils.

The crossreferences Table

Since there may be several data sources that describe the same protein (that is to say, the protein from a particular species with a particular amino acid sequence), we cre-

ated a `crossreferences` table to collect all references to a protein other than the original data source in which it was encountered. This table is very similar in structure to the `proteins` table since it embodies very similar types of information. Each entry in the table has a unique integer identifier (`xref_id`) as well as reference (`prot_id`) to the entry in the `proteins` table for the associated protein. Moreover, `crossreferences` contains fields called `xref_source_id`, `ref_id`, `ref_desc`, and `xref_updated` that are completely analogous to the similarly-named columns in `proteins`.

The `programs` Table

The `programs` table is identical to the `species` table in complexity and structure. Here, each coiled-coil prediction programs (e.g., COILS or Paircoil) has an entry consisting of a unique integer identifier (`program_id`) and textual name (`program_name`).

The `coils` Table

The `coils` table is designed to collect the results of all the coiled-coil prediction programs on the set of proteins contained in the `proteins` table. Each program may predict multiple coiled coils of varying length, location, register, and confidence in each protein. Therefore it is important that this table be very flexible and capable of accommodating many coiled coils per protein. Given this flexibility requirement, we decided to allow multiple rows per protein where each row represented a single predicted coiled coil.

Each entry consists of a unique integer identifier (`cc_id`), the identifier of the protein in which the coiled coil is predicted (`prot_id`), the identifier of the prediction program that generated the entry (`program_id`), the offset of the start of the coiled-coil from the beginning of the protein's amino acid sequence (`cc_start`), the length (in amino acid residues) of the coiled coil (`cc_length`), the probability score generated by the prediction program (`cc_prob`), and the predicted register of the first residue in the coiled coil. Additional information, such as the amino acid sequence of the coiled coil or the percentage of the associated protein occupied by the coiled coil, can be quickly generated by basic queries of both the `coils` and `proteins` tables.

4.2.3 The Indexing Strategy

Because of the large number of rows in some of our tables (hundreds of thousands of proteins and millions of predicted coiled coils) and the complex nature of queries we need to support, a good indexing strategy is imperative to achieve reasonable performance. Without indexing, RDBMSs must execute a query on a table by examining every row of that table to ensure the completeness of the results. By constructing well-designed indices of the table in question prior to query execution, however, the RDBMS can quickly determine which rows (if any) are relevant to the query. For example, if a user wanted to obtain the list of proteins associated with humans (assume in this case that the `spec_id` for "Homo sapiens" is five), they would issue the following SQL query:

```
SELECT * FROM proteins WHERE spec_id=5
```

If no indexing were used on the proteins table, every row would be checked to see if its `spec_id` column were equal to five. However, if the column were indexed, the RDBMS would already have built a list of all the rows whose `spec_id` column equals five and could return them very quickly.

The importance of indexing in our case is heightened by our decision to allow many entries in `coils` and `crossreferences` for each protein. We could have chosen to "hard-code" information about each referring data source as columns in the `proteins` table, but this would have required modifying the table structure for each new source. As usual, this decision was a trade-off and the cost of the flexibility is query performance.

Details of index implementation vary for different RDBMSs but the methods employed by MySQL 4.0 are sufficient for our needs. In MySQL, indices can (and often should) span multiple columns, but the order of these columns in the index matters greatly. To create the index, the RDBMS constructs a B-Tree consisting of all the column values occurring in the table (where a maximum depth can be specified to prevent unnecessarily large trees). For each leaf in this tree, MySQL constructs another B-Tree consisting of all the values of the second column occurring in rows in which

the first column matches the leaf. This process continues for each column in the index. To execute a query involving the first two columns of the index, MySQL simply walks down the first B-Tree to find the leaf node associated with all rows matching on the first column. Next, MySQL traverses the second-column B-Tree associated with the first leaf node to find all rows matching on both of the first two columns. It is important to note that, because of this construction, MySQL can only benefit from a multiple-column index when all of the leading index columns are restricted in the query. For example, an index of columns one, two, and three cannot be used to find all rows in which column three matches a particular value—MySQL doesn't have the information it needs to traverse the first two sets of B-Trees. The index can, however, be used to find all rows in which column one matches a particular value since column one was the first column specified in the creation of the index. Sections 7.2 ("Optimizing SELECT Statements and Other Queries") and 7.4 ("Optimizing Database Structure") of the MySQL Reference Manual describe more detailed use of indices to optimize query performance.

In general, designing a good indexing strategy requires understanding the scenarios in which the database is utilized. Common queries that are executed very frequently or in a time-sensitive manner should be optimized while uncommon, non-time-critical queries can be left unindexed. Table 4.1 summarizes the non-primary indices used in the CCDB and the indices for each table are described in more detail below.

Table	Column 1	Column 2	Column 3
proteins	spec_id	source_id	
proteins	ref_id		
coils	cc_prob	program_id	cc_length
coils	prot_id	cc_prob	program_id
crossreferences	prot_id	xref_source_id	
crossreferences	ref_id		

Table 4.1: Multiple-column indices in the `coiled_coils` database.

Indices on `proteins` and `crossreferences`

Because the `proteins` and `crossreferences` tables contain the same type of information, it is likely they will be queried with similar restrictions and, therefore, it is appropriate for them to include similar indices. Both include a primary index on their unique identifiers (`prot_id` and `xref_id` respectively) that allows individual rows to be extracted rapidly based on these identifiers. This primary index is particularly important when extracting information from multiple tables simultaneously.

After extraction by identifier, we expect the most common query to involve the accession number of the protein in a particular data source. For example, we expect users will likely want information related to the protein whose "id" is `NF00499275` (an identifier in the PIR NREF database). Thus, we have created single-column indices on the first ten characters of the `ref_id` column of each table.

We expect the next most common protein query to involve the species in which the protein exists and, perhaps additionally, the source of the protein information. Requests covered in this category include those for "all proteins in E. coli" and "all human RefSeq proteins." In order to support such queries on the `proteins` table, we create a two-column index spanning `spec_id` and `source_id`. Since the `crossreferences` table doesn't contain the species information directly, we substitute the column linking each row to its associated entry in `proteins`. That is, we create a two-column index spanning `prot_id` and `xref_source_id`. This substitution has the added benefit of allowing requests for "all crossreferences to protein 2236233".

Finally, in order to quickly find perfect matches of a particular sequence in the database, the first six characters of the `prot_seq` column of `proteins` are also indexed. Currently, there are 349,016 unique entries in the `proteins` table with 174,508 unique six-character sequence prefixes. Thus, for an arbitrary query sequence, this index reduces the number of potential matches from the number of rows in the table to an average of just two. However, it is important to note that this index does *not* provide any improvement in regular expression or substring matching of the `prot_seq` column since only the first six characters are indexed. Such queries require MySQL

to examine each row of the table.

Indices on `coils`

Much like for the `proteins` and `crossreferences`, several indices are required to support fast queries of the predicted coiled coils in the `coils` table. As in the previous examples, a primary index is created on the uniquely identifying column `coil_id` to facilitate single-row extraction.

Additionally, to support queries based on prediction program and confidence score, indices on the `program_id` and `cc_prob` columns are necessary. However, we do not want to index solely these columns as we expect queries to often combine these fields with the length of the predicted coiled coil or the protein in which its contained. To support both scenarios simultaneously, we create *two* three-column indices. The first spans `prot_id`, `cc_prob`, and `program_id`. The second spans `cc_prob`, `program_id`, and `cc_length`. Between the two, users can quickly retrieve all coiled coils in a particular protein, all coiled coils in the protein with a particular likelihood, all coiled coils in the protein predicted by a particular program with a particular likelihood, all coiled coils of a particular length predicted by a particular program with a particular likelihood, and many other similar data sets.

Indices on Other Tables

All tables other than `proteins`, `crossreferences`, and `coils` are simple enough that they do not require any indexing other than the simple primary indices on their unique identifiers. These primary indices allow the tables to be efficiently joined to the more substantive tables during queries.

4.3 Automatic Generation and Maintenance

A primary concern when constructing a database of this nature is keeping it up-to-date and synchronized with the myriad data sources from which its contents are derived. Some of these sources are updated sporadically (for example, once a quarter) while

others are updated continuously, making it extremely challenging, if not impossible, to maintain any reasonable level of synchronization by hand. Since the purpose of this database is to simplify, rather than complicate, our data mining, an automated maintenance system is the only viable approach.

In general, this system must regularly download the latest protein information from the external data sources, parse it and insert it into the `proteins` and `crossreferences` tables, run the coiled-coil prediction programs on new proteins, and store the results in the `coils` table. Moreover, it must accomplish these tasks in a predictable, auditable, and reasonably efficient manner.

4.3.1 Retrieving Protein Information

The first step in the maintenance process is to download the latest protein information from each of the data sources of interest. Currently, these data sources only include Georgetown's non-redundant protein reference PIR NREF and the NCBI's similarly non-redundant RefSeq database. However, additional sources can be incorporated with minimal extra effort, assuming their entire contents can be downloaded in the common FASTA format.

The retrieval process starts with the execution of a script for each data source that downloads and condenses the source's protein information into the FASTA format. In the simple (and common) case where the protein information is available directly from the source in FASTA format, this retrieval script is trivial. However, if the data is not available in a single FASTA file—some websites use a custom format or separate their data into many smaller FASTA files—these scripts are responsible for parsing the data in a source-specific fashion to generate a FASTA file for use in the next step of the process.

After the data has been downloaded and parsed into one FASTA file per source, a common insertion script is invoked for each file. This script (`insertFastaSeqs.pl`) is passed the name of the file to be parsed as well as the name of the associated data source. It then loops through each entry in the file and considers only those proteins derived from a species of interest (i.e., a species with an entry in the `species` table).

For each such protein, the script checks whether a protein with the exact same sequence exists in the `proteins` table. If not, it creates an entry and populates the columns with the relevant information. If there is already an entry for this protein, the script investigates whether there is already a reference to this protein *from the current data source.* If not, it creates an entry in the `crossreferences` table and populates the relevant fields. If a reference from the current data source already existed, the script checks whether the reference should be updated and updates columns as necessary. The end result of the execution of `insertFastaSeqs.pl` is the freshening of the `proteins` and `crossreferences` tables with the latest information from all of the external data sources.

4.3.2 Predicting Coiled Coils

Once the protein information has been downloaded from all relevant sources and inserted into the database, the four coiled-coil prediction programs (COILS, Paircoil, Multicoil, and Marcoil) must be run on each of the new proteins. Rather than support up to 100 different confidence scores for each predicted coiled coil, we have decided to create 14 common "views" of the coiled coils. That is, we have selected 14 confidence scores of interest (0.10, 0.20, 0.30, 0.40, 0.50, 0.60, 0.70, 0.80, 0.90, 0.92, 0.94, 0.96, 0.98, 0.99) and taken a snapshot of the predicted coiled coils at each cutoff. Because each program has its own unique input requirements and output formats, we have created separate scripts for the execution of each as described below.

COILS

Prediction by COILS is divided into two phases: program execution and output parsing. The execution step is embodied in `runCoils.pl`, a simple script that queries the database for new proteins, writes these proteins to a temporary FASTA file, and invokes the COILS executable with the temporary file as input. The parsing step is handled by a separate script, `parseCoils.pl`, which is invoked once per confidence cutoff. The COILS program generates confidence scores for every amino acid

residue. Therefore, `parseCoils.pl` parses the output of COILS, looking for contiguous residues with scores exceeding the cutoff of interest. It considers each such region as a separate coiled coil and creates an entry in the `coils` table for each.

Paircoil

Much as with COILS, using Paircoil to predict coiled coils in a large number of proteins is best accomplished in two stages. The first stage (implemented in `runPaircoil.pl` consists of program execution and trivial output parsing. This script collects all protein sequences from the database and compiles them into a single large FASTA file. Importantly, the Paircoil program itself is unable to accommodate sequences larger than about 20,000 residues so proteins larger than this size are currently skipped (although we plan to work around this problem in the future). Once the FASTA file has been generated, `runPaircoil.pl` passes it to Paircoil with a specified probability threshold of 0.05. This low cutoff causes Paircoil to output to a file all coiled coils whose probability is *at least* 0.05. Finally, this output file is read by `runPaircoil.pl` and each predicted coiled coil is inserted into a temporary table (`_coils_intermediate`).

The second stage of Paircoil prediction involves the condensation of the temporary results in `_coils_intermediate`. Each row in this table corresponds to a coiled region of a particular confidence. However, a single coiled coil may consist of several regions of different probability. For example, Paircoil might assign the first 20 residues probability 0.44 and the next 4 resides probability 0.62. In this case, only the last 4 residues of the coil would be visible with a confidence cutoff of 0.5 or 0.6. However, all 24 residues would be visible at a cutoff of 0.4. The `condenseResults.pl` script, however, accounts for these possibilities by merging all relevant coiled coils at each supported confidence level. The script begins with the highest confidence level (99) and finds all coiled coils predicted with at least this score. If any of these proteins are adjacent to or overlap one another, they are merged together. At the next confidence level (98), only the coils resulting from the previous round of parsing and coils scoring between 98 and 99 need be examined for adjacencies or overlaps. The process contin-

ues iteratively, merging new coiled coils with higher-scoring ones at each step. Once all confidence levels have been considered, `condenseResults.pl` writes the results to the permanent `coils` table.

Multicoil

Multicoil prediction and processing proceeds in much the same manner as Paircoil, with only a few additional quirks. First, just as Paircoil could not accommodate large sequences, Multicoil cannot accommodate large *numbers* of sequences. As a result, `runMulticoil.pl` divides the proteins to be analyzed into a series of FASTA files, each consisting of no more than 25,000 sequences. The same script then feeds each file to Multicoil and examines the program's output. Unlike `runPaircoil.pl`, however, `runMulticoil.pl` must recognize Multicoil's distinction between coiled coils likely to exist as dimers and those likely to exist as trimers. Once all coil information has been collected, it is written to the same temporary table used by Paircoil (`_coils_intermediate`) and the previously-discussed `condenseResults.pl` script is invoked.

Marcoil

Because Marcoil has the useful feature of accepting a set of confidence cutoffs as input and generating coiled coils according to the per-residue cutoff scheme employed by `parseCoils.pl`, no extensive processing of Marcoil's output is required. Instead, `runMarcoil.pl` simply queries the database for all new proteins, writes one to a temporary file, invokes Marcoil on that file with set of supported confidence cutoffs, inserts the results into the `coils` table, then proceeds to the next new protein.

4.3.3 Automation: `cron` and the Beowulf Cluster

The collection of maintenance scripts described above is coordinated by a master shell script called `updateDatabase.sh` that is scheduled (via the `cron` utility) to run on a weekly basis. The script is executed on an Internet-connected server that begins

the process by downloading the relevant data sets as described in Section 4.3.1. This master script then takes advantage of an available Beowulf cluster (effectively isolated from the Internet) and the MySQL server for parallel execution of the remaining tasks. Care has been taken to define the dependencies between the jobs so that, for example, `runCoils.pl` completes prior to the execution of `parseCoils.pl` on its output.

4.4 The Web Service API

At least as critical to the success of the CCDB as content and performance is a clean and useful programmatic interface. A core requirement of the interface is that it be accessible from many common programming languages. Additionally, it must be extensible for use across the Internet and, in doing so, serve a buffer protecting the types of queries executed against the internal database. To meet both of these requirements, we decided to implement an XML web service that conforms to the relevant World Wide Web Consortium (W3C) standards. In particular, adherence to the working drafts of the Simple Object Access Protocol (SOAP) and the XML Protocol (XMLP) is important for portability across languages and platforms.

Rather than deal directly with the details of these complex protocols, we chose to create our service with the Java Web Services Developer Pack. This platform conforms to the standard Web Services Interoperability (WS-I) Basic Profile and offers easy implementation of XML-RPC (through its JAX-RPC package).

4.4.1 Data Structures

The CCDB web service relies on five core data structures, mirroring the internal database structures, as described below:

The `Species` and `Program` Classes

Two of the five core data structures are extremely simple, consisting of just two fields—an `ID` and a `Name`. These two, `Species` and `Program`, are provided as simple, extensible representations of their logical counterparts. In addition, by providing both

numerical and textual identifiers, these classes allow an element of client flexibility. That is, the programmer of an application that uses the service can decide whether his requirements dictate the use of computationally-efficient numerical identifiers or human-friendly textual names.

The `Source` Class

The `Source` data structure is only marginally more complex than the two previous structures. Like the others, `Source` provides both a numerical `ID` and a textual `Name`, but this class also provides three other textual fields: `Description`, `HomeURL`, and `RefURL`. The first, as its name implies, consists of the brief description of the data source as stored in the `sources` table of the database. Similarly, the URL members consist of the addresses of the data source's homepage and the page to be visited for more information about a particular protein (provided a reference identifier is appended to the end of the `RefURL` field).

The `Protein` Class

Unlike the previous structures, the `Protein` class consists of more than just simple data types. Like the others, it contains a identifier (`id`); however, unlike the others, this identifier is a string and represents the name or accession number assigned to the protein by the associated data source (i.e., the `ref_id` column of the `proteins` or `crossreferences` tables. The structure also contains a textual description (`desc`), an instance of the `Source` data type (`source`), a list of `Protein` data types (`xRefs`), an instance of the `Species` data type (`species`), and a textual representation of the protein's amino acid sequence (`sequence`).

The `Coil` Class

Much like `Protein`, the `Coil` class consists of both simple and complex data types. In particular, it contains the string identifier of the protein in which it is contained (`proteinID`), an instance of the `Program` data type representing prediction program that identified this particular coiled coil, the numeric offset of the coiled coil in the

protein's sequence (`startIndex`), the numeric length of the coil (`length`), the numeric confidence score (`likelihood`), the character representation of the start register of this coil (`startRegister`), and the amino acid sequence of the coiled coil (`sequence`). Notably, there is no unique identifier for each coil. This is because coiled coils are never passed as arguments to the web service (this is discussed in more detail in the next section). Also important to note is the fact that the containing protein is referenced in this structure only by it's ID—not as an entire `Protein` structure as might be expected. This simplification is intended to dramatically reduce the size of the result set produced by queries that return many predicted coils. Clients can obtain an instance of the `Protein` structure at a later time through a separate call to the web service. The final unusual property of the `Coil` structure is that it contains the amino acid sequence of the coil—a property *not* recorded in the backing `coils` table. This property is expected to be very commonly used and, by adding it to this structure, we expect to reduce the number of `Protein` structures that must be retrieved for common web service calls.

4.4.2 The CCDB Interface

The web service interface itself is defined as a set of six functions that depend heavily on the data structures described in the previous section. This interface includes three functions for discovering supported parameters, two functions to query for matching coiled coils, and one function for retrieving detailed information about particular proteins. For performance and ease of input validation, all functions accept only simple data types and, where appropriate, these inputs correspond to the unique identifiers of their respective objects. For example, integer source identifiers are used to specify which data source to query and textual protein identifiers are used to specify which protein to consider. These functions are described in more depth below:

Parameter Discovery Functions

Before clients can search for coiled coils, they need to have some information about what values are acceptable as input for different parameters, and they can obtain this information through the web services three "parameter discovery functions." These include `getSupportedSources()`, `getSupportedPrograms()`, and `getSupportedSpecies()`. Each accepts no arguments and returns a list of instances of the corresponding data type. For example, `getSupportedSources()` returns a list of instances of `Source` with one entry per data source supported by the backing database. Taken together, these three functions provide enough information to perform a wide variety of additional queries through the CCDB web service.

Coiled-Coil Query Functions

Once a client has used the parameter discovery functions to familiarize itself with the CCDB's supported input values, it can begin to search for coiled coils through two distinct query functions. The first, `countCCs()`, simply returns the number of results that match a given set of input values while the second, `findCCs()`, actually returns a list of `Coil` instances representing all the coiled coils matching the input criteria. Both functions accept the same first seven input arguments:

- **int iSourceID** - the protein source to consider
- **String sProteinID** - the name of a particular protein to consider
- **int iSpeciesID** - the species to consider
- **int iProgramID** - the prediction program to consider
- **int iLikelihood** - the coil confidence score to consider
- **int iMinLength** - the minimum coil length to consider
- **String sDesc** - a regular expression to be matched against textual protein annotations

In addition, `findCCs()` accepts two other arguments: an integer `iStartIndex` indicating the offset into the result set at which to start returning results and an integer `iResultLimit` specifying the maximum number of results to be returned by the query. A null (zero) value can be provided in place of any argument to ignore that input criteria. For example, passing zero as the `iSourceID` argument searches all supported sources and passing zero as the `iResultLimit` argument defers to the default bound on the size of result set.

The Protein Retrieval Function

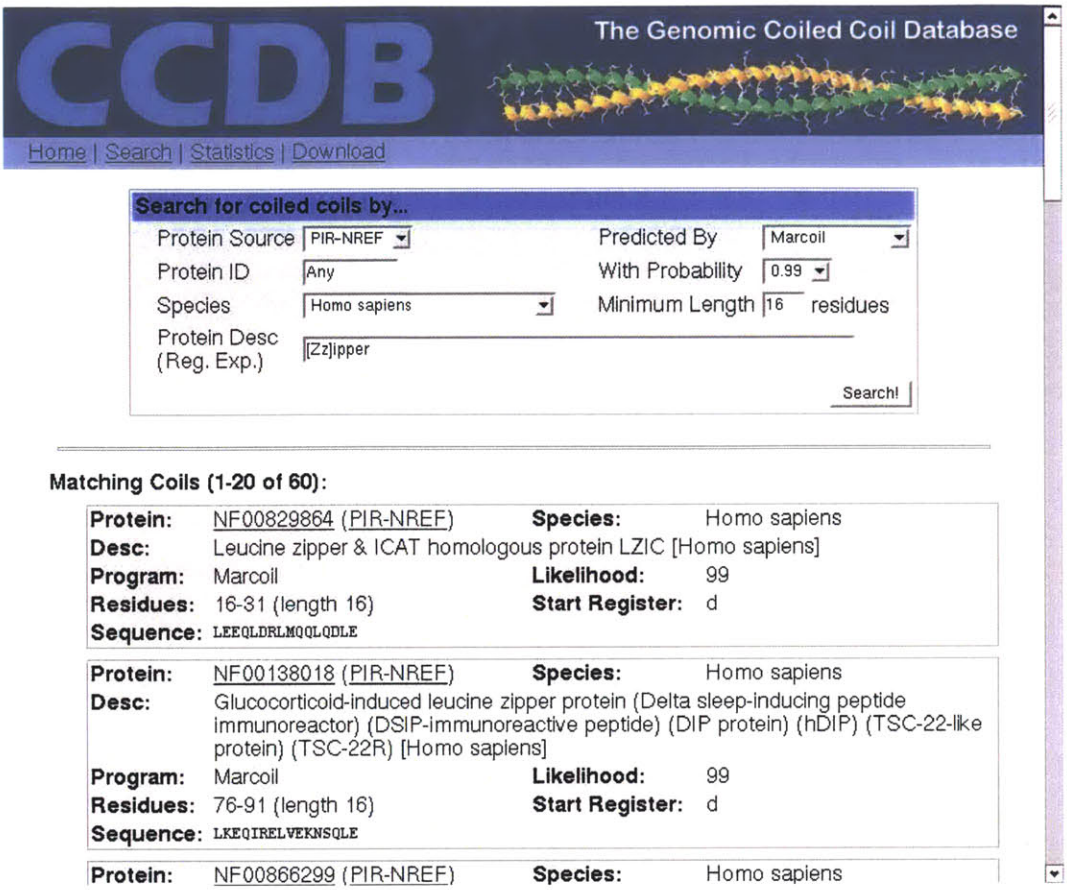
The final function provided by the CCDB web service interface, `findProtein()`, can be used to obtain detailed information about any protein referenced by a `Coil` object. A list of such objects are returned by a call to `findCCs()`, but, in order to minimize the size of the result sets, each contains only a string identifier of the protein in which the coiled coil is contained. A complete `Protein` structure can be obtained by passing one of these protein identifiers as the only input to `findProtein()`.

4.4.3 Implementation

The implementation of the CCDB web service is not complicated and requires no innovative algorithms. The parameter discovery functions need only issue a simple SQL query to the RDBMS and convert the results to a list of the appropriate data types. Even the apparently complex coiled-coil query and protein retrieval functions need only validate their inputs, construct a SQL query statement, issue that statement to the RDBMS, and convert any results to the correct data types. The bulk of the work in every case is performed by the RDBMS.

4.5 The Web Front End

While the web service is a convenient interface for developers creating client applications, the CCDB also needs a convenient interface for non-developers. To this end, we created a website that allows visitors to query the CCDB just as if they were using



[bth]

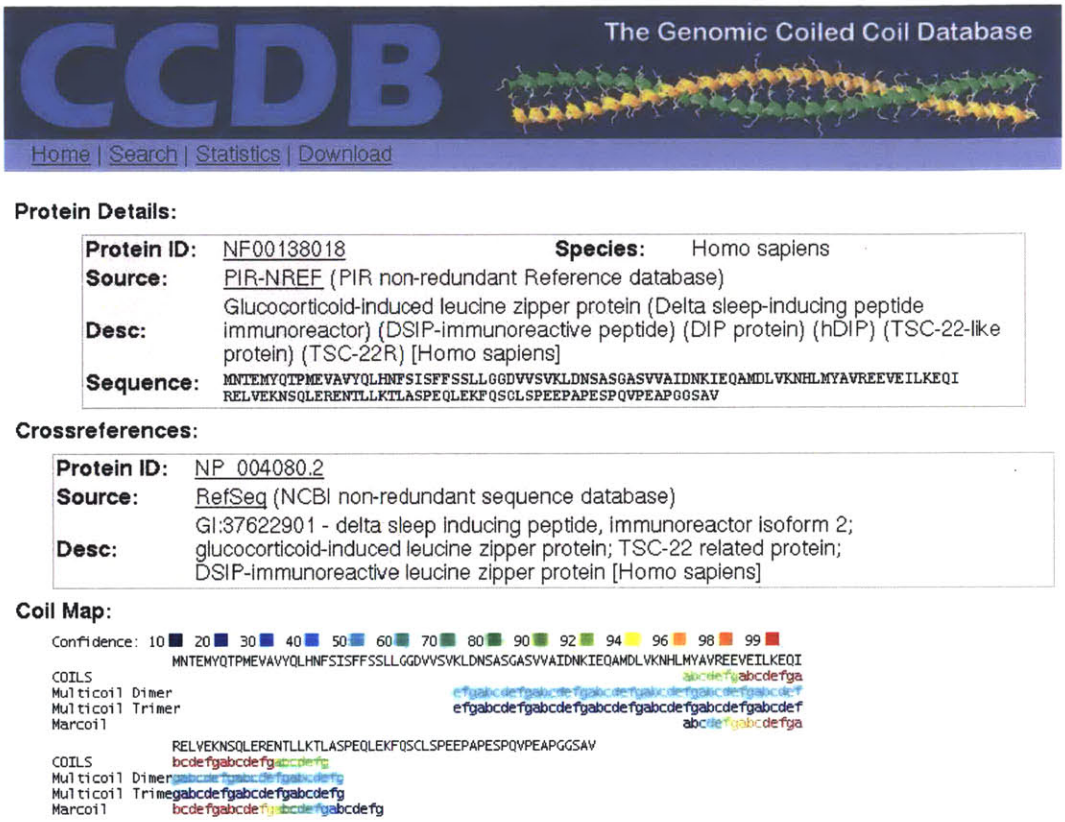
Figure 4-2: The query results summary page of the CCDB website. This particular screenshot depicts a successful regular expression search for coiled coils in proteins described as consisting of a "zipper." Summary information about 20 matching coiled coils is displayed per page.

the web service API. In fact, the CCDB website is actually implemented using calls to that API.

The heart of the CCDB website is a Java Server Pages (JSP) form (called `search.jsp`) that allows a user to query the database for coiled coils by protein source, protein name, species, prediction program, prediction probability, minimum coil length, and protein description. To minimize invalid inputs and improve the user experience, the protein source, species, prediction program, and prediction probability fields are all implemented as drop-down boxes preloaded with valid inputs. When

the search button is clicked, the form is submitted to the server for processing and execution. The JSP application server performs only minimal input validation before generating a request to the CCDB web service which performs its own validation. The JSP application server then formats the results of the web service query as an HTML table (shown in Figure 4-2 and sends the page to the client browser.

From this result page, the user can link to another JSP page (`protein.jsp`, shown in Figure 4-3) that displays detailed information about an individual protein. In addition to the basic protein information (source, name, species, description, cross-references, and sequence), the page also includes an image representing the location, likelihood, and register of every coiled coil predicted to occur in the protein. Rather than keeping a library of images, we implement this functionality through a Java Servlet (called `coilImage`) that generates the image dynamically based on information it too retrieves from the CCDB through the web service API.



[tbh]

Figure 4-3: The protein detail page of the CCDB website. In addition to providing full sequence information and textual annotation for proteins, this page identifies cross-references of the protein in other databases. Most notably, the page also includes a dynamically-generated "map" of all the protein's predicted coiled coils. The lower-case letters correspond to the heptad register of the predicted coils and the coloration indicates the confidence with which the coiled coil is predicted.

Appendix A

FNAC API Documentation

This appendix documents details of the FNAC packages using JavaDoc generated from source comments.

A.1 Class Hierarchy

A.1.1 Classes

- java.lang.Object
  - fnac.FnacObject (in A.5.2, page 124)
    - fnac.AbstractOperator (in A.5.2, page 114)
      - fnac.AbstractMultiSourceOperator (in A.5.2, page 112)
        - fnac.pairwise.PairwiseComparator (in A.6.1, page 130)
        - fnac.pairwise.SimpleMerger (in A.6.1, page 135)
      - fnac.AbstractSingleSourceOperator (in A.5.2, page 117)
        - fnac.connectivity.ConnectivityAnalyzer (in A.3.1, page 87)
        - fnac.motifs.MotifAnalyzer (in A.2.1, page 80)
    - fnac.AbstractSource (in A.5.2, page 119)
      - fnac.sources.FileSource (in A.4.1, page 93)
      - fnac.sources.MySqlSource (in A.4.1, page 96)
  - fnac.motifs.MotifShape (in A.2.1, page 84)

Interfaces

- fnac.FnacObjectInterface
  - fnac.Operator (in A.5.1, page 102)
    - fnac.MultiSourceOperator (in A.5.1, page 100)
    - fnac.SingleSourceOperator (in A.5.1, page 105)
  - fnac.Source (in A.5.1, page 107)

Exceptions

- java.lang.Object
  - java.lang.Throwable
    - java.lang.Exception
      - fnac.FnacException (in A.5.3, page 127)

A.2 Package fnac.motifs

Package Contents *Page*

Classes

A.2.1 Classes

Class **MotifAnalyzer**

Utility class for analyzing FNAC networks on the basis of topological network motifs. Given enough time, this class is capable of detecting motifs of arbitrary size and keeping track of individual motifs even after the call to operate() has completed. Logically, this class treats motifs as a pair of objects: the nodes between which the motif occurs; and the topology or "shape" of the motif. All node sets are represented as short[]s and all shapes are represented as MotifShapes.

Declaration

public class MotifAnalyzer
extends fnac.AbstractSingleSourceOperator (in A.5.2, page 117)

Constructor summary

MotifAnalyzer()
MotifAnalyzer(Source)

Method summary

getMaxMotifSize() Retrieve the maximum motif size currently being considered
getShape(short[]) Retrieve the shape of the network topology involving a particular set of nodes.
iterator() Get an iterator over all motifs.
operate()
setMaxMotifSize(int) Specify the maximum motif size to consider during a call to operate().

Constructors

- *MotifAnalyzer*

  `public` **MotifAnalyzer**`( )`

- *MotifAnalyzer*

  `public` **MotifAnalyzer**`( fnac.Source source )`

Methods

- *getMaxMotifSize*

  `public int` **getMaxMotifSize**`( )`

  - **Description**

    Retrieve the maximum motif size currently being considered

  - **Returns** – the current maximum motif size

- *getShape*

  `public MotifShape` **getShape**`( short[] nodeSet ) throws fnac.FnacException`

  - **Description**

    Retrieve the shape of the network topology involving a particular set of nodes.

  - **Parameters**

    * `nodeSet` – - The node indices for which to retrieve the MotifShape

  - **Returns** – - the MotifShape representing the topology between the specified edges if the edges form a connected motif and null otherwise.

  - **Throws**

    * `fnac.FnacException` – if operate() has not been called or another exceptional condition has been encountered

---

- *iterator*

  `public java.util.Iterator` **iterator**`( ) throws fnac.FnacException`

- **Description**

  Get an iterator over all motifs. Each item is actually an instance of Map.Entry where the key is the short[] representing the nodes involved in the motif and the value is the MotifShape representing the topological shape of the motif.

- **Returns** – an iterator over all detected motifs

- **Throws**

  * `fnac.FnacException` – if operate() has not been called or another exceptional condition has been encountered

---

- *operate*

  `void operate( ) throws fnac.FnacException`

  - **Description copied from fnac.Operator** **(in A.5.1, page 102)**

    Execute the network operation. This is almsot always required prior to retrieval of operation results.

  - **Throws**

    * `fnac.FnacException` – if Source(s) have not been specified for this Operator or another exceptional condition was encountered.

- *setMaxMotifSize*

  `public void setMaxMotifSize( int maxMotifSize ) throws fnac.FnacException`

  - **Description**

    Specify the maximum motif size to consider during a call to operate().

  - **Parameters**

    * `maxMotifSize` – - the integer (greater than 2) to be the maximum motif size

  - **Throws**

* `fnac.FnacException` – if maxMotifSize is less than 3 or another exceptional condition is encountered.

Members inherited from class `fnac.AbstractSingleSourceOperator` (in A.5.2, page 117)

- `public Source getSource( )`
- `public void setSource( Source source )`

Members inherited from class `fnac.AbstractOperator` (in A.5.2, page 114)

- `public File getDataDir( )`
- `public Properties getOptions( )`
- `public boolean isOperationComplete( )`
- `public abstract void operate( ) throws FnacException`
- `public void setDataDir( java.io.File dataDir ) throws FnacException`
- `public void setOptions( java.util.Properties options )`

Members inherited from class `fnac.FnacObject` (in A.5.2, page 124)

- `public PrintStream getErrorStream( )`
- `public PrintStream getOutputStream( )`
- `public void setErrorStream( java.io.PrintStream errorStream )`
- `public void setOutputStream( java.io.PrintStream outputStream )`

Class **MotifShape**

Object representation of a motif shape.

Public clients cannot create instances of this class. Internally, this class utilizes a Factory pattern to consolidate many occurrences of a shape into many references to a single MotifShape instance.

Furthermore, each n-node motif shape is uniquely identified by the shapes of all possible submotifs of size n-1. For example, a 4-node motif connected as a square can be identified by 4 occurrences of the 3-node motif connected as an 'L'. A 4-node motif connected as a 'U' can be identified by 2 occurrences of the same 3-node 'L' shape.

Declaration

public class MotifShape
extends java.lang.Object
implements java.io.Serializable

Method summary

toString()

Serializable Fields

- private MotifShapeChildren **_children**
- private short **_nodeCount**
- private boolean **_rootConnected**

Methods

- *toString*

  `public java.lang.String` **toString**`( )`

A.3 Package fnac.connectivity

Package Contents *Page*

Classes

A.3.1 Classes

Class **ConnectivityAnalyzer**

Utility class for analyzing FNAC networks on the basis of their connectivity. Can compute average connectivity, per-node connectivities, connectivity distributions, and hub nodes.

Declaration

public class ConnectivityAnalyzer
extends fnac.AbstractSingleSourceOperator (in A.5.2, page 117)
implements fnac.SingleSourceOperator

Constructor summary

ConnectivityAnalyzer()
ConnectivityAnalyzer(Source)

Method summary

getAverageConnectivity() Retrieve the average connectivity of the analyzed network

getConnectivityDistribution() Retrieve the connectivity distribution of the analyzed network.

getConnectivityOfNode(int) Retrieve the connectivity of a particular node.

getConnectivityOfNode(String) Retrieve the connectivity of a particular node.

getHubIndices() Retrieve the indices in the GINY GraphPerspective of all hub nodes.

getHubNames() Retrieve the names of all hub nodes.

operate()

Constructors

- *ConnectivityAnalyzer*

  `public` **ConnectivityAnalyzer**`( )`

- *ConnectivityAnalyzer*

  `public` **ConnectivityAnalyzer**`( fnac.Source source )`

Methods

- *getAverageConnectivity*

  `public float` **getAverageConnectivity**`( ) throws fnac.FnacException`

  - **Description**

    Retrieve the average connectivity of the analyzed network

  - **Returns** – the average connectivity

  - **Throws**

    * `fnac.FnacException` – if operate() has not been called or another exceptional condition was encountered

- *getConnectivityDistribution*

  `public int[]` **getConnectivityDistribution**`( ) throws fnac.FnacException`

  - **Description**

    Retrieve the connectivity distribution of the analyzed network.

  - **Returns** – an arrray where the value of each element `i` is the number of nodes with connectivity `i`

  - **Throws**

    * `fnac.FnacException` – if operate() has not been called or another exceptional condition was encountered

---

- *getConnectivityOfNode*

  `public int` **getConnectivityOfNode**`( int` **nodeIndex** `) throws fnac.FnacException`

  - **Description**

    Retrieve the connectivity of a particular node.

  - **Parameters**

    * `nodeIndex` - - The index of the desired node in the GINY GraphPerspective

  - **Returns** – the connectivity of the requested node

  - **Throws**

    * `fnac.FnacException` – if operate() has not been called, results could not be found for the particular node, or another exceptional condition was encountered

---

- *getConnectivityOfNode*

  `public int` **getConnectivityOfNode**`( java.lang.String` **nodeName** `) throws fnac.FnacException`

  - **Description**

    Retrieve the connectivity of a particular node.

  - **Parameters**

    * `nodeName` - - The name of the desired node

  - **Returns** – the connectivity of the requested node

  - **Throws**

    * `fnac.FnacException` – if operate() has not been called, results could not be found for the particular node, or another exceptional condition was encountered

---

- *getHubIndices*

  `public int[] getHubIndices( ) throws fnac.FnacException`

  - **Description**

    Retrieve the indices in the GINY GraphPerspective of all hub nodes. A hub node is defined as a node whose connectivity is more than twice the average connectivity of the network.

  - **Returns** – an array of hub node indices

  - **Throws**

    * `fnac.FnacException` – if operate() has not been called or another exceptional condition was encountered

---

- *getHubNames*

  `public java.util.Set getHubNames( ) throws fnac.FnacException`

  - **Description**

    Retrieve the names of all hub nodes. See getHubIndices() for the definition of "hub."

  - **Returns** – an array of hub node names

  - **Throws**

    * `fnac.FnacException` – if operate() has not been called or another exceptional condition was encountered.

- *operate*

  `void operate( ) throws fnac.FnacException`

  - **Description copied from fnac.Operator** (in A.5.1, page 102)

    Execute the network operation. This is almsot always required prior to retrieval of operation results.

  - **Throws**

    * `fnac.FnacException` – if Source(s) have not been specified for this Operator or another exceptional condition was encountered.

Members inherited from class `fnac.AbstractSingleSourceOperator` (in A.5.2, page 117)

- `public Source getSource( )`
- `public void setSource( Source source )`

Members inherited from class `fnac.AbstractOperator` (in A.5.2, page 114)

- `public File getDataDir( )`
- `public Properties getOptions( )`
- `public boolean isOperationComplete( )`
- `public abstract void operate( ) throws FnacException`
- `public void setDataDir( java.io.File dataDir ) throws FnacException`
- `public void setOptions( java.util.Properties options )`

Members inherited from class `fnac.FnacObject` (in A.5.2, page 124)

- `public PrintStream getErrorStream( )`
- `public PrintStream getOutputStream( )`
- `public void setErrorStream( java.io.PrintStream errorStream )`
- `public void setOutputStream( java.io.PrintStream outputStream )`

A.4 Package fnac.sources

Package Contents *Page*

Classes

A.4.1 Classes

Class **FileSource**

Implementation of the FNAC Source interface capable of reading and writing networks as "raw interaction files" where each line of the file is of the format:

```
PROTEIN_NAME1 PROTEIN_NAME2
```

Declaration

public class FileSource
extends fnac.AbstractSource (in A.5.2, page 119)
implements fnac.Source

Constructor summary

FileSource(Source, File)
FileSource(String, File)

Method summary

loadData()
saveData()
saveDataAs(File) Save a network to a file other than the one specified as its "source."

Constructors

- *FileSource*
  `public` **FileSource(** `fnac.Source` **source,** `java.io.File` **sourceFile** **)**

- *FileSource*
  `public` **FileSource(** `java.lang.String` **name,** `java.io.File` **sourceFile** **)**

- *loadData*

  `void` **loadData**`( ) throws fnac.FnacException`

  - **Description copied from fnac.Source (in A.5.1, page 107)**

    Load the network at the location specified by this Source.

  - **Throws**

    * `fnac.FnacException` – if exceptional circumstances were encountered while trying to load the network data

---

- *saveData*

  `void` **saveData**`( ) throws fnac.FnacException`

  - **Description copied from fnac.Source (in A.5.1, page 107)**

    Save the network to the location specified by this Source (optional).

  - **Throws**

    * `fnac.FnacException` – if exceptional circumstances were encountered while trying to save the network data

---

- *saveDataAs*

  `public void` **saveDataAs**`( java.io.File` **outFile** `) throws fnac.FnacException`

  - **Description**

    Save a network to a file other than the one specified as its "source."

  - **Parameters**

    * `outFile` – - The file to which to write this network

  - **Throws**

    * `fnac.FnacException` – if an exceptional condition was encountered while trying to write the file

Members inherited from class `fnac.AbstractSource` (in A.5.2, page 119)

- `public GraphPerspective` **getGraphPerspective( )**
- `public String` **getName( )**
- `public int` **getNodeIndex(** `java.lang.String` **nodeName)**
- `public int` **getNodeIndices( )**
- `public int` **getNodeIndices(** `java.lang.String[]` **nodeNames)**
- `public String` **getNodeName(** `int` **nodeIndex)**
- `public String` **getNodeNames( )**
- `public abstract void` **loadData( )** `throws FnacException`
- `public abstract void` **saveData( )** `throws FnacException`
- `public void` **setGraphPerspective(** `giny.model.GraphPerspective` **graph)**
- `public void` **setNodeMapping(** `java.lang.String` **nodeName,** `int` **nodeIndex)**

Members inherited from class `fnac.FnacObject` (in A.5.2, page 124)

- `public PrintStream` **getErrorStream( )**
- `public PrintStream` **getOutputStream( )**
- `public void` **setErrorStream(** `java.io.PrintStream` **errorStream)**
- `public void` **setOutputStream(** `java.io.PrintStream` **outputStream)**

Class **MySqlSource**

Implementation of the FNAC Source interface capable of loading (but not saving) a network from a MySQL database query. Database connection parameters and query string are specified during object construction. SQL queries should return two columns - the names of the nodes in each edge. In the event that more than two columns are returned, only the first two are examined.

Declaration

public class MySqlSource
extends fnac.AbstractSource (in A.5.2, page 119)
implements fnac.Source

Constructor summary

MySqlSource(String, String, int, String, String, String, String)
Create a new MySqlSource instance.

Method summary

loadData()
saveData()

Constructors

- *MySqlSource*
  `public` **MySqlSource**`( java.lang.String` **name**`, java.lang.String` **host**`, int` **port**`, java.lang.String` **database**`, java.lang.String` **query**`, java.lang.String` **user**`, java.lang.String` **password** `)`
  - **Description**
    Create a new MySqlSource instance.
  - **Parameters**

* `name` - - Friendly source name
* `host` - - Hostname or IP address of the server hosting the RDBMS
* `port` - - TCP port number on which to connect to the RDBMS
* `database` - - The name of the database to be queried
* `query` - - The SQL query to be executed
* `user` - - Username with which to connect to the database (optional)
* `password` - - Password with which to connect to the database (optional)

Methods

- *loadData*

  `void` **loadData**`( ) throws fnac.FnacException`

  - **Description copied from fnac.Source** (in **A.5.1, page 107**)

    Load the network at the location specified by this Source.

  - **Throws**
    * `fnac.FnacException` - if exceptional circumstances were encountered while trying to load the network data

---

- *saveData*

  `void` **saveData**`( ) throws fnac.FnacException`

  - **Description copied from fnac.Source** (in **A.5.1, page 107**)

    Save the network to the location specified by this Source (optional).

  - **Throws**
    * `fnac.FnacException` - if exceptional circumstances were encountered while trying to save the network data

Members inherited from class `fnac.AbstractSource` (in A.5.2, page 119)

- `public GraphPerspective` **getGraphPerspective**`( )`

- public String **getName**( )
- public int **getNodeIndex**(java.lang.String **nodeName**)
- public int **getNodeIndices**( )
- public int **getNodeIndices**(java.lang.String[] **nodeNames**)
- public String **getNodeName**(int **nodeIndex**)
- public String **getNodeNames**( )
- public abstract void **loadData**( ) throws FnacException
- public abstract void **saveData**( ) throws FnacException
- public void **setGraphPerspective**(giny.model.GraphPerspective **graph**)
- public void **setNodeMapping**(java.lang.String **nodeName**, int **nodeIndex**)

Members inherited from class fnac.FnacObject (in A.5.2, page 124)

- public PrintStream **getErrorStream**( )
- public PrintStream **getOutputStream**( )
- public void **setErrorStream**(java.io.PrintStream **errorStream**)
- public void **setOutputStream**(java.io.PrintStream **outputStream**)

A.5 Package fnac

Package Contents *Page*

Interfaces

Classes

A.5.1 Interfaces

Interface **MultiSourceOperator**

Interface for any utility that analyzes a multiple data networks and generates a set of statistics for them.

Declaration

```
public interface MultiSourceOperator
implements Operator
```

All known subclasses AbstractMultiSourceOperator (in A.5.2, page 112), SimpleMerger (in A.6.1, page 135), PairwiseComparator (in A.6.1, page 130)

All classes known to implement interface AbstractMultiSourceOperator (in A.5.2, page 112)

Method summary

addSource(Source) Add a network to the list of those on which to be operate.

addSources(Collection) Add multiple networks to the list of those on which to operate.

getSources() Get a reference to the data sources currently specified for this Operator

Methods

- *addSource*

  `void` **addSource(** `Source` **newSource)**

  - **Description**

    Add a network to the list of those on which to be operate.

  - **Parameters**

* `source` - - Data source to add

- *addSources*

  `void` **addSources(** `java.util.Collection` **newSources)**

  - **Description**

    Add multiple networks to the list of those on which to operate.

  - **Parameters**

    * `source` – Data sources to be added

- *getSources*

  `java.util.Collection` **getSources( )**

  - **Description**

    Get a reference to the data sources currently specified for this Operator

  - **Returns** – a copy of the set of networks on which to operate

Interface Operator

Interface for simple network operations. Only allows options to be configured and operation to be invoked. Inheriting interfaces should almost always be used instead of this one.

Declaration

public interface Operator
implements FnacObjectInterface

All known subclasses MotifAnalyzer (in A.2.1, page 80), ConnectivityAnalyzer (in A.3.1, page 87), SingleSourceOperator (in A.5.1, page 105), MultiSourceOperator (in A.5.1, page 100), AbstractSingleSourceOperator (in A.5.2, page 117), AbstractOperator (in A.5.2, page 114), AbstractMultiSourceOperator (in A.5.2, page 112), SimpleMerger (in A.6.1, page 135), PairwiseComparator (in A.6.1, page 130)

All known subinterfaces SingleSourceOperator (in A.5.1, page 105), MultiSourceOperator (in A.5.1, page 100)

All classes known to implement interface AbstractOperator (in A.5.2, page 114)

Method summary

getDataDir() Get the current scratch directory used by this Operator.

getOptions() Returns the current configuration options of this Operator.

isOperationComplete() Find out whether an invocation of `operate()` has completed and whether results are ready for retrieval.

operate() Execute the network operation.

setDataDir(File) Set the scratch directory to be used by this Operator.

setOptions(Properties) Specifies configuration options for this operator.

Methods

- *getDataDir*

  `java.io.File` **getDataDir( )**

  - **Description**

    Get the current scratch directory used by this Operator.

  - **Returns** – The scratch directory currently being used

- *getOptions*

  `java.util.Properties` **getOptions( )**

  - **Description**

    Returns the current configuration options of this Operator.

  - **Returns** – The set of options that would be used if the Operator were to be executed now.

- *isOperationComplete*

  `boolean` **isOperationComplete( )**

  - **Description**

    Find out whether an invocation of `operate()` has completed and whether results are ready for retrieval.

  - **Returns** – True if a call to operate() has completed successfully and related results are ready for examinations

---

- *operate*

  `void` **operate( )** `throws fnac.FnacException`

  - **Description**

    Execute the network operation. This is almsot always required prior to retrieval of operation results.

- **Throws**
  - `fnac.FnacException` – if Source(s) have not been specified for this Operator or another exceptional condition was encountered.

- *setDataDir*

  `void setDataDir( java.io.File dataDir ) throws fnac.FnacException`

  - **Description**

    Set the scratch directory to be used by this Operator. Many Operators require the use of temporary files. Callers should specify the directory in which the Operator can create, modify, and delete temporary files at will. No other processes or Operators should be using this directory as the Operator makes no guarantee as to its behavior within the specified directory. NOTE: The directory and parent directory will be created if they do not exist.

  - **Parameters**
    - `dataDir` – The new scratch directory
  - **Throws**
    - `OperatorException` – If the specified File does exist but is not a directory

---

- *setOptions*

  `void setOptions( java.util.Properties options )`

  - **Description**

    Specifies configuration options for this operator. Supported options vary depending on the particular Operator.

  - **Parameters**
    - `options` – The set of options used to configure this Operator

Interface **SingleSourceOperator**

Interface for any utility that analyzes a single data network and generates a set of statistics for that network.

Declaration

```
public interface SingleSourceOperator
implements Operator
```

All known subclasses MotifAnalyzer (in A.2.1, page 80), ConnectivityAnalyzer (in A.3.1, page 87), AbstractSingleSourceOperator (in A.5.2, page 117)

All classes known to implement interface ConnectivityAnalyzer (in A.3.1, page 87), AbstractSingleSourceOperator (in A.5.2, page 117)

Method summary

getSource() Get a reference to the data source that is currently specified for this Analyzer.

setSource(Source) Specify a data source to be analyzed

Methods

- *getSource*

  `Source` **`getSource( )`**

  - **Description**

    Get a reference to the data source that is currently specified for this Analyzer.

  - **Returns** – A reference to the data source currently set for analysis if one exists, null otherwise

- *setSource*

  `void setSource( Source source )`

  - **Description**

    Specify a data source to be analyzed

  - **Parameters**

    * `source` – Data source to be analyzed

Interface **Source**

Interface to all network representations. This interface allows clients to load / save data, set / retrieve GINY network representations, and set / retrieve node name-to-node index mappings.

Declaration

public interface Source
implements FnacObjectInterface

All known subclasses MySqlSource (in A.4.1, page 96), FileSource (in A.4.1, page 93), AbstractSource (in A.5.2, page 119)

All classes known to implement interface MySqlSource (in A.4.1, page 96), FileSource (in A.4.1, page 93), AbstractSource (in A.5.2, page 119)

Method summary

getGraphPerspective() Get a GINY GraphPerspective representation of this network.

getName() Get the friendly name for this network

getNodeIndex(String) Gets the index in the GINY GraphPerspective associated with a particular node name.

getNodeIndices() Get the indices of all nodes in this network.

getNodeIndices(String[]) Get the indices in the GINY GraphPerspective associated with a particular set of node names.

getNodeName(int) Gets the name of the node associated with a particular index in the GINY GraphPerspective.

getNodeNames() Get the names of all nodes in this network.

loadData() Load the network at the location specified by this Source.

saveData() Save the network to the location specified by this Source (optional).

setGraphPerspective(GraphPerspective) Reset the underlying network to a particular topology.

setNodeMapping(String, int) Maps a node name to a particular node index in the GINY GraphPerspective (and vice-versa).

Methods

- *getGraphPerspective*

  `giny.model.GraphPerspective` **getGraphPerspective( )**

  - **Description**

    Get a GINY GraphPerspective representation of this network. This representation allows efficient graph analysis for sparse networks, provided that node and edge indices are used in all communications with the GraphPerspective. See documentation of the node name-to-node index mapping functionalities also provided by the Source interface.

  - **Returns** – A GINY GraphPerspective representation of this network

---

- *getName*

  `java.lang.String` **getName( )**

  - **Description**

    Get the friendly name for this network

  - **Returns** – The name of this network

- *getNodeIndex*

  `int` **getNodeIndex(** `java.lang.String` **nodeName)**

  - **Description**

    Gets the index in the GINY GraphPerspective associated with a particular node name.

  - **Parameters**

* `nodeName` - - The name of the node for which to find an index

- **Returns** – The index of the specified node in the GINY GraphPerspective (provided such a mapping has previously been created).

- *getNodeIndices*

  `int[]` **getNodeIndices( )**

  - **Description**

    Get the indices of all nodes in this network.

  - **Returns** – The indices of all nodes in this GINY GraphPerspective for which a node name-to-node index mapping has been created.

- *getNodeIndices*

  `int[]` **getNodeIndices(** `java.lang.String[]` **nodeNames)**

  - **Description**

    Get the indices in the GINY GraphPerspective associated with a particular set of node names.

  - **Parameters**

    * `nodeNames` - - The names of the nodes for which to find indices

  - **Returns** – The indices of the specified nodes in the GINY GraphPerspective (provided such mappings have previously been created).

- *getNodeName*

  `java.lang.String` **getNodeName(** `int` **nodeIndex)**

  - **Description**

    Gets the name of the node associated with a particular index in the GINY GraphPerspective.

- **Parameters**
  - `nodeIndex` - - The node index for which to find a name
- **Returns** – The name of the node at nodeIndex in the GINY GraphPerspective (provided a name has previously been mapped to this node)

- *getNodeNames*

  `java.lang.String[]` **getNodeNames( )**

  - **Description**

    Get the names of all nodes in this network.
  - **Returns** – The names of all nodes in this network for which a node name-to-node index mapping has been created.

- *loadData*

  `void` **loadData( )** `throws fnac.FnacException`

  - **Description**

    Load the network at the location specified by this Source.
  - **Throws**
    - `fnac.FnacException` – if exceptional circumstances were encountered while trying to load the network data

---

- *saveData*

  `void` **saveData( )** `throws fnac.FnacException`

  - **Description**

    Save the network to the location specified by this Source (optional).
  - **Throws**
    - `fnac.FnacException` – if exceptional circumstances were encountered while trying to save the network data

---

- *setGraphPerspective*

  `void` **setGraphPerspective(** `giny.model.GraphPerspective` **graph)**

  - **Description**

    Reset the underlying network to a particular topology. WARNING!! Use of this method requires extreme caution as it does not adjust the node name-to-node index mappings maintained by this Source. Graphs out of sync with the node mappings can result in completely unreliable (and difficult to debug) results.

  - **Parameters**

    * **graph** - - The GINY GraphPerspective representation of the new network topology

---

- *setNodeMapping*

  `void` **setNodeMapping(** `java.lang.String` **nodeName,** `int` **nodeIndex)**

  - **Description**

    Maps a node name to a particular node index in the GINY GraphPerspective (and vice-versa).

  - **Parameters**

    * **nodeName** - - The name of the node
    * **nodeIndex** - - The index of the node in the GINY GraphPerspective

A.5.2 Classes

Class **AbstractMultiSourceOperator**

Abstract implementation of the MultiSourceOperator interface. Should be inherited by any class that operates on multiple networks.

Declaration

public abstract class AbstractMultiSourceOperator
extends fnac.AbstractOperator (in A.5.2, page 114)
implements MultiSourceOperator

All known subclasses SimpleMerger (in A.6.1, page 135), PairwiseComparator (in A.6.1, page 130)

Method summary

addSource(Source)

addSources(Collection)

getSources()

Methods

- *addSource*

  `void` **addSource(** `Source` **newSource)**

  - **Description copied from MultiSourceOperator (in A.5.1, page 100)**

    Add a network to the list of those on which to be operate.

  - **Parameters**

    * `source` - - Data source to add

- *addSources*

  `void` **addSources(** `java.util.Collection` **newSources)**

- **Description copied from MultiSourceOperator (in A.5.1, page 100)**

  Add multiple networks to the list of those on which to operate.

- **Parameters**

  * `source` – Data sources to be added

- *getSources*

  `java.util.Collection` **getSources( )**

  - **Description copied from MultiSourceOperator (in A.5.1, page 100)**

    Get a reference to the data sources currently specified for this Operator

  - **Returns** – a copy of the set of networks on which to operate

Members inherited from class `fnac.AbstractOperator` (in A.5.2, page 114)

- `public File` **getDataDir( )**
- `public Properties` **getOptions( )**
- `public boolean` **isOperationComplete( )**
- `public abstract void` **operate( ) throws FnacException**
- `public void` **setDataDir(java.io.File dataDir) throws FnacException**
- `public void` **setOptions(java.util.Properties options)**

Members inherited from class `fnac.FnacObject` (in A.5.2, page 124)

- `public PrintStream` **getErrorStream( )**
- `public PrintStream` **getOutputStream( )**
- `public void` **setErrorStream(java.io.PrintStream errorStream)**
- `public void` **setOutputStream(java.io.PrintStream outputStream)**

Class **AbstractOperator**

Abstract implementation of the Operator interface. Should only be inherited when neither SingleSourceOperator nor MultiSourceOperator are adequate.

Declaration

public abstract class AbstractOperator
extends fnac.FnacObject (in A.5.2, page 124)
implements Operator

All known subclasses MotifAnalyzer (in A.2.1, page 80), ConnectivityAnalyzer (in A.3.1, page 87), AbstractSingleSourceOperator (in A.5.2, page 117), AbstractMultiSourceOperator (in A.5.2, page 112), SimpleMerger (in A.6.1, page 135), PairwiseComparator (in A.6.1, page 130)

Method summary

getDataDir()
getOptions()
isOperationComplete()
operate()
setDataDir(File)
setOptions(Properties)

Methods

- *getDataDir*

  `java.io.File` **getDataDir( )**

  - **Description copied from Operator (in A.5.1, page 102)**
    Get the current scratch directory used by this Operator.
  - **Returns** – The scratch directory currently being used

- *getOptions*

  `java.util.Properties` **getOptions( )**

  - **Description copied from Operator (in A.5.1, page 102)**
    Returns the current configuration options of this Operator.
  - **Returns** – The set of options that would be used if the Operator were to be executed now.

- *isOperationComplete*

  `boolean` **isOperationComplete( )**

  - **Description copied from Operator (in A.5.1, page 102)**
    Find out whether an invocation of `operate()` has completed and whether results are ready for retrieval.
  - **Returns** – True if a call to operate() has completed successfully and related results are ready for examinations

---

- *operate*

  `void` **operate( )** `throws fnac.FnacException`

  - **Description copied from Operator (in A.5.1, page 102)**
    Execute the network operation. This is almsot always required prior to retrieval of operation results.
  - **Throws**
    * `fnac.FnacException` – if Source(s) have not been specified for this Operator or another exceptional condition was encountered.

- *setDataDir*

  `void` **setDataDir(** `java.io.File` **dataDir)** `throws fnac.FnacException`

  - **Description copied from Operator (in A.5.1, page 102)**

Set the scratch directory to be used by this Operator. Many Operators require the use of temporary files. Callers should specify the directory in which the Operator can create, modify, and delete temporary files at will. No other processes or Operators should be using this directory as the Operator makes no guarantee as to its behavior within the specified directory. NOTE: The directory and parent directory will be created if they do not exist.

- **Parameters**
  - `dataDir` – The new scratch directory
- **Throws**
  - `OperatorException` – If the specified File does exist but is not a directory

---

- *setOptions*

  `void` **setOptions(** `java.util.Properties` **options)**

  - **Description copied from Operator (in A.5.1, page 102)**
    Specifies configuration options for this operator. Supported options vary depending on the particular Operator.
  - **Parameters**
    - `options` – The set of options used to configure this Operator

Members inherited from class `fnac.FnacObject` (in A.5.2, page 124)

- `public PrintStream` **getErrorStream( )**
- `public PrintStream` **getOutputStream( )**
- `public void` **setErrorStream(** `java.io.PrintStream` **errorStream)**
- `public void` **setOutputStream(** `java.io.PrintStream` **outputStream)**

Class AbstractSingleSourceOperator

Abstract implementation of the SingleSourceOperator interface. Should be inherited by any class that operates on individual networks.

Declaration

public abstract class AbstractSingleSourceOperator
extends fnac.AbstractOperator (in A.5.2, page 114)
implements SingleSourceOperator

All known subclasses MotifAnalyzer (in A.2.1, page 80), ConnectivityAnalyzer (in A.3.1, page 87)

Method summary

getSource()
setSource(Source)

Methods

- *getSource*

  `Source` **getSource**`( )`

  - **Description copied from SingleSourceOperator (in A.5.1, page 105)**
    Get a reference to the data source that is currently specified for this Analyzer.
  - **Returns** – A reference to the data source currently set for analysis if one exists, null otherwise

---

- *setSource*

  `void` **setSource**`( Source source )`

- **Description copied from SingleSourceOperator (in A.5.1, page 105)**

  Specify a data source to be analyzed

- **Parameters**

  * `source` – Data source to be analyzed

Members inherited from class `fnac.AbstractOperator` (in A.5.2, page 114)

- `public File` **getDataDir( )**
- `public Properties` **getOptions( )**
- `public boolean` **isOperationComplete( )**
- `public abstract void` **operate( )** `throws FnacException`
- `public void` **setDataDir(** `java.io.File` **dataDir)** `throws FnacException`
- `public void` **setOptions(** `java.util.Properties` **options)**

Members inherited from class `fnac.FnacObject` (in A.5.2, page 124)

- `public PrintStream` **getErrorStream( )**
- `public PrintStream` **getOutputStream( )**
- `public void` **setErrorStream(** `java.io.PrintStream` **errorStream)**
- `public void` **setOutputStream(** `java.io.PrintStream` **outputStream)**

Class **AbstractSource**

Abstract implementation of the Source interface. Should be inherited by any class representing a new type of source location.

Declaration

public abstract class AbstractSource
extends fnac.FnacObject (in A.5.2, page 124)
implements Source

All known subclasses MySqlSource (in A.4.1, page 96), FileSource (in A.4.1, page 93)

Method summary

getGraphPerspective()
getName()
getNodeIndex(String)
getNodeIndices()
getNodeIndices(String[])
getNodeName(int)
getNodeNames()
loadData()
saveData()
setGraphPerspective(GraphPerspective)
setNodeMapping(String, int)

Methods

- *getGraphPerspective*

  `giny.model.GraphPerspective` **getGraphPerspective( )**

- **Description copied from Source** (in A.5.1, page 107)
  Get a GINY GraphPerspective representation of this network. This representation allows efficient graph analysis for sparse networks, provided that node and edge indices are used in all communications with the GraphPerspective. See documentation of the node name-to-node index mapping functionalities also provided by the Source interface.
- **Returns** – A GINY GraphPerspective representation of this network

---

- *getName*

  `java.lang.String` **getName( )**

  - **Description copied from Source** (in A.5.1, page 107)
    Get the friendly name for this network
  - **Returns** – The name of this network

- *getNodeIndex*

  `int` **getNodeIndex(** `java.lang.String` **nodeName)**

  - **Description copied from Source** (in A.5.1, page 107)
    Gets the index in the GINY GraphPerspective associated with a particular node name.
  - **Parameters**
    - `nodeName` – - The name of the node for which to find an index
  - **Returns** – The index of the specified node in the GINY GraphPerspective (provided such a mapping has previously been created).

- *getNodeIndices*

  `int[]` **getNodeIndices( )**

  - **Description copied from Source** (in A.5.1, page 107)
    Get the indices of all nodes in this network.

- **Returns** – The indices of all nodes in this GINY GraphPerspective for which a node name-to-node index mapping has been created.

- *getNodeIndices*

  `int[]` **getNodeIndices(** `java.lang.String[]` **nodeNames)**

  - **Description copied from Source** **(in A.5.1, page 107)**
    Get the indices in the GINY GraphPerspective associated with a particular set of node names.
  - **Parameters**
    - `nodeNames` – - The names of the nodes for which to find indices
  - **Returns** – The indices of the specified nodes in the GINY GraphPerspective (provided such mappings have previously been created).

- *getNodeName*

  `java.lang.String` **getNodeName(** `int` **nodeIndex)**

  - **Description copied from Source** **(in A.5.1, page 107)**
    Gets the name of the node associated with a particular index in the GINY GraphPerspective.
  - **Parameters**
    - `nodeIndex` – - The node index for which to find a name
  - **Returns** – The name of the node at nodeIndex in the GINY GraphPerspective (provided a name has previously been mapped to this node)

- *getNodeNames*

  `java.lang.String[]` **getNodeNames( )**

  - **Description copied from Source** **(in A.5.1, page 107)**
    Get the names of all nodes in this network.

- **Returns** – The names of all nodes in this network for which a node name-to-node index mapping has been created.

- *loadData*

  `void loadData( ) throws fnac.FnacException`

  - **Description copied from Source (in A.5.1, page 107)**
    Load the network at the location specified by this Source.
  - **Throws**
    * `fnac.FnacException` – if exceptional circumstances were encountered while trying to load the network data

---

- *saveData*

  `void saveData( ) throws fnac.FnacException`

  - **Description copied from Source (in A.5.1, page 107)**
    Save the network to the location specified by this Source (optional).
  - **Throws**
    * `fnac.FnacException` – if exceptional circumstances were encountered while trying to save the network data

---

- *setGraphPerspective*

  `void setGraphPerspective( giny.model.GraphPerspective graph )`

  - **Description copied from Source (in A.5.1, page 107)**
    Reset the underlying network to a particular topology. WARNING!! Use of this method requires extreme caution as it does not adjust the node name-to-node index mappings maintained by this Source. Graphs out of sync with the node mappings can result in completely unreliable (and difficult to debug) results.
  - **Parameters**

* `graph` - - The GINY GraphPerspective representation of the new network topology

---

- *setNodeMapping*

  `void` **setNodeMapping(** `java.lang.String` **nodeName,** `int` **nodeIndex)**

  - **Description copied from Source** **(in A.5.1, page 107)**

    Maps a node name to a particular node index in the GINY GraphPerspective (and vice-versa).

  - **Parameters**

    * `nodeName` - - The name of the node
    * `nodeIndex` - - The index of the node in the GINY GraphPerspective

Members inherited from class `fnac.FnacObject` (in A.5.2, page 124)

- `public PrintStream` **getErrorStream( )**
- `public PrintStream` **getOutputStream( )**
- `public void` **setErrorStream(** `java.io.PrintStream` **errorStream)**
- `public void` **setOutputStream(** `java.io.PrintStream` **outputStream)**

Class **FnacObject**

Declaration

public abstract class FnacObject
extends java.lang.Object
implements FnacObjectInterface

All known subclasses MotifAnalyzer (in A.2.1, page 80), ConnectivityAnalyzer (in A.3.1, page 87), MySqlSource (in A.4.1, page 96), FileSource (in A.4.1, page 93), AbstractSource (in A.5.2, page 119), AbstractSingleSourceOperator (in A.5.2, page 117), AbstractOperator (in A.5.2, page 114), AbstractMultiSourceOperator (in A.5.2, page 112), SimpleMerger (in A.6.1, page 135), PairwiseComparator (in A.6.1, page 130)

Method summary

getErrorStream()
getOutputStream()
setErrorStream(PrintStream)
setOutputStream(PrintStream)

Methods

- *getErrorStream*

  `java.io.PrintStream` **getErrorStream( )**

  - **Description copied from FnacObjectInterface**
    Get a reference to the destination where error messages will be written. The default target is System.err. Error messages are typically only generated when the Operator encounters unexpected or exceptional circumstances during its execution.
  - **Returns** – The current target for error messages or NULL if such messages are disabled

---

- *getOutputStream*

  `java.io.PrintStream` **getOutputStream( )**

  - **Description copied from FnacObjectInterface**

    Get a reference to the destination where standard output messages will be written. The default target is System.out. Such output messages typically include logs generated by the Operator during its execution.

  - **Returns** – The current target for standard output messages or NULL if such messages are disabled

---

- *setErrorStream*

  `void` **setErrorStream(** `java.io.PrintStream` **errorStream)**

  - **Description copied from FnacObjectInterface**

    Specify an alternative destination for error messages. The default target is System.err. Error messages are typically only generated when the Operator encounters unexpected or exceptional circumstances during its execution.

  - **Parameters**

    * `errorStream` – The new target for error messages or NULL to disable these messages

---

- *setOutputStream*

  `void` **setOutputStream(** `java.io.PrintStream` **outputStream)**

  - **Description copied from FnacObjectInterface**

    Specify an alternative destination for standard output messages. The default target is System.out. Such output messages typically include logs generated by the Operator during its execution.

  - **Parameters**

* **outputStream** – The new target for standard output messages or NULL to disable these messages

A.5.3 Exceptions

Class **FnacException**

Generic checked exception used throughout FNAC

Declaration

```
public class FnacException
extends java.lang.Exception
```

Constructor summary

FnacException() Creates a new instance of `FnacException` without detail message.

FnacException(String) Constructs an instance of `FnacException` with the specified detail message.

Constructors

- *FnacException*

  `public` **FnacException**`( )`

  - **Description**

    Creates a new instance of `FnacException` without detail message.

- *FnacException*

  `public` **FnacException**`( java.lang.String msg )`

  - **Description**

    Constructs an instance of `FnacException` with the specified detail message.

  - **Parameters**

    * `msg` – the detail message.

Members inherited from class `java.lang.Exception`

Members inherited from class `java.lang.Throwable`

- `public synchronized native Throwable` **fillInStackTrace( )**
- `public Throwable` **getCause( )**
- `public String` **getLocalizedMessage( )**
- `public String` **getMessage( )**
- `public StackTraceElement` **getStackTrace( )**
- `public synchronized Throwable` **initCause(Throwable)**
- `public void` **printStackTrace( )**
- `public void` **printStackTrace(java.io.PrintStream)**
- `public void` **printStackTrace(java.io.PrintWriter)**
- `public void` **setStackTrace(StackTraceElement[])**
- `public String` **toString( )**

A.6 Package fnac.pairwise

Package Contents *Page*

Classes

A.6.1 Classes

Class **PairwiseComparator**

Utility class for comparing two networks in a pairwise fashion. During a call to operate(), objects of this class find the nodes contained in both networks, find the edges contained within these common nodes, find the complete intersection of the two networks, and keeps track of orphaned edges (those not connected to any other edges).

Declaration

public class PairwiseComparator
extends fnac.AbstractMultiSourceOperator (in A.5.2, page 112)

Constructor summary

PairwiseComparator()

PairwiseComparator(Collection)

Method summary

getCommonNodeCount() Get the number of nodes present in both networks

getEdgeCountOnCommonNodes(Source) Get the count of edges of one of the networks that fall completely within the set of common nodes.

getOrphanedEdgeCount(Source) A static method for counting the number of orphan edges in a network on the fly.

getOrphanedEdgeCountOnCommonNodes(Source) Get the count of orphaned edges in the trimmed version of a particular network.

getOverlappingSource() Get a network representation of the intersection of the two networks.

getSourceWithOnlyCommonNodes(Source) Get the trimmed version of a particular network.

operate()

Constructors

- *PairwiseComparator*

  `public` **PairwiseComparator**`( )`

- *PairwiseComparator*

  `public` **PairwiseComparator**`( java.util.Collection sources )`

Methods

- *getCommonNodeCount*

  `public int` **getCommonNodeCount**`( ) throws fnac.FnacException`

  - **Description**

    Get the number of nodes present in both networks

  - **Returns** – the number of nodes present in both networks

  - **Throws**

    * `fnac.FnacException` – if operate() has not been called or another exceptional condition has been encountered

- *getEdgeCountOnCommonNodes*

  `public int` **getEdgeCountOnCommonNodes**`( fnac.Source src )`
  `throws fnac.FnacException`

  - **Description**

    Get the count of edges of one of the networks that fall completely within the set of common nodes. I generally refer to the set of such edges as a "trimmed" network.

  - **Parameters**

* `src` - - The network for which to count the number of trimmed edges

- **Returns** – the number of edges in src whose both endpoints are contained in the set of common nodes
- **Throws**
  * `fnac.FnacException` – if operate() has not been called, the specified source has not been analyzed, or another exceptional condition was encountered

---

- *getOrphanedEdgeCount*

  `public static int` **getOrphanedEdgeCount**`( fnac.Source src )`

  - **Description**

    A static method for counting the number of orphan edges in a network on the fly.
  - **Parameters**
    * `src` - - The source to examine for orphan edges
  - **Returns** - - The number of orphan edges found in the network

---

- *getOrphanedEdgeCountOnCommonNodes*

  `public int` **getOrphanedEdgeCountOnCommonNodes**`( fnac.Source src ) throws fnac.FnacException`

  - **Description**

    Get the count of orphaned edges in the trimmed version of a particular network.
  - **Parameters**
    * `src` - - The network whose trimmed version should be examined for orphan edges
  - **Returns** – the number of such orphan edges found
  - **Throws**

* `fnac.FnacException` – if operate() has not been called, the specified source has not been analyzed, or another exceptional condition was encountered

---

- *getOverlappingSource*

  `public fnac.Source` **getOverlappingSource**`( ) throws fnac.FnacException`

  - **Description**

    Get a network representation of the intersection of the two networks.

  - **Returns** – - A Source embodying the intersecting nodes and edges between the networks

  - **Throws**

    * `fnac.FnacException` – if operate() has not een called or another exceptional condition has been encountered

---

- *getSourceWithOnlyCommonNodes*

  `public fnac.Source` **getSourceWithOnlyCommonNodes**`( fnac.Source src ) throws fnac.FnacException`

  - **Description**

    Get the trimmed version of a particular network.

  - **Parameters**

    * `src` – - The network for which to obtain a trimmed version

  - **Returns** – a Source representing the trimmed source

  - **Throws**

    * `fnac.FnacException` – if operate() has not been called, the specified source has not been analyzed, or another exceptional condition was encountered

---

- *operate*

  `void` **operate**`( ) throws fnac.FnacException`

  - **Description copied from fnac.Operator (in A.5.1, page 102)**

    Execute the network operation. This is almsot always required prior to retrieval of operation results.

  - **Throws**

    * `fnac.FnacException` – if Source(s) have not been specified for this Operator or another exceptional condition was encountered.

Members inherited from class `fnac.AbstractMultiSourceOperator` (in A.5.2, page 112)

- `public void` **addSource**`( Source` **newSource** `)`
- `public void` **addSources**`( java.util.Collection` **newSources** `)`
- `public Collection` **getSources**`( )`

Members inherited from class `fnac.AbstractOperator` (in A.5.2, page 114)

- `public File` **getDataDir**`( )`
- `public Properties` **getOptions**`( )`
- `public boolean` **isOperationComplete**`( )`
- `public abstract void` **operate**`( ) throws FnacException`
- `public void` **setDataDir**`( java.io.File` **dataDir** `) throws FnacException`
- `public void` **setOptions**`( java.util.Properties` **options** `)`

Members inherited from class `fnac.FnacObject` (in A.5.2, page 124)

- `public PrintStream` **getErrorStream**`( )`
- `public PrintStream` **getOutputStream**`( )`
- `public void` **setErrorStream**`( java.io.PrintStream` **errorStream** `)`
- `public void` **setOutputStream**`( java.io.PrintStream` **outputStream** `)`

Class **SimpleMerger**

Utility class for generating the intersections between multiple networks. When operate() is called, all possible intersections are computed and can be quickly retrieved thereafter. For example, if networks A, B, and C are merged, the intersections AxB, AxC, BxC, and AxBxC are all computed. If you only want to compute a single intersection, you should only specify 2 networks.

Declaration

public class SimpleMerger
extends fnac.AbstractMultiSourceOperator (in A.5.2, page 112)

Constructor summary

SimpleMerger()
SimpleMerger(Collection)

Method summary

getAllIntersections() Get the Collection of all intersections.
getIntersection(Collection) Retrieve a specific intersection network.
operate()

Constructors

- *SimpleMerger*
  `public` **SimpleMerger**`( )`

- *SimpleMerger*
  `public` **SimpleMerger**`( java.util.Collection` **sources** `)`

- *getAllIntersections*

  `public java.util.Collection` **getAllIntersections**`( ) throws fnac.FnacException`

  - **Description**

    Get the Collection of all intersections. Each intersection is represented as a Source. Also, in addition to the "real" intersections, the returned Collection also contains the original sources and a null source (representing the intersection of none of the networks).

  - **Returns** – a Collection of Sources representing all possible network intersections

  - **Throws**

    * `fnac.FnacException` – if operate() has not been called or another exceptional condition was encountered

---

- *getIntersection*

  `public fnac.Source` **getIntersection**`( java.util.Collection sources ) throws fnac.FnacException`

  - **Description**

    Retrieve a specific intersection network.

  - **Parameters**

    * `sources` – - The Collection of sources whose intersection is being requested

  - **Returns** – the Source representing the intersection of the specified networks

  - **Throws**

* `fnac.FnacException` – if operate() has not been called, one or more of the specified sources was not analyzed during the last call to operate(), or another exceptional condition has been encountered

---

- *operate*

  `void` **operate**`( ) throws fnac.FnacException`

  - **Description copied from fnac.Operator (in A.5.1, page 102)**
    Execute the network operation. This is almsot always required prior to retrieval of operation results.
  - **Throws**
    * `fnac.FnacException` – if Source(s) have not been specified for this Operator or another exceptional condition was encountered.

Members inherited from class `fnac.AbstractMultiSourceOperator` (in A.5.2, page 112)

- `public void` **addSource**`( Source newSource )`
- `public void` **addSources**`( java.util.Collection newSources )`
- `public Collection` **getSources**`( )`

Members inherited from class `fnac.AbstractOperator` (in A.5.2, page 114)

- `public File` **getDataDir**`( )`
- `public Properties` **getOptions**`( )`
- `public boolean` **isOperationComplete**`( )`
- `public abstract void` **operate**`( ) throws FnacException`
- `public void` **setDataDir**`( java.io.File dataDir ) throws FnacException`
- `public void` **setOptions**`( java.util.Properties options )`

Members inherited from class `fnac.FnacObject` (in A.5.2, page 124)

- `public PrintStream` **getErrorStream**`( )`
- `public PrintStream` **getOutputStream**`( )`
- `public void` **setErrorStream**`( java.io.PrintStream` **errorStream** `)`
- `public void` **setOutputStream**`( java.io.PrintStream` **outputStream** `)`

Bibliography

[1]

[2] M. Ashburner, C. A. Ball, and J. A. Blake et al. Gene Ontology: tool for the unification of biology. *Nature Genetics*, 25:25–9, 2000.

[3] G. Bader, D. Betel, and C. W. V. Hogue. BIND: the Biomolecular Interaction Network Database. *Nucleic Acids Research*, 31(1):248–50, 2003.

[4] A.-L. Barabási and R. Albert. Emergence of Scaling in Random Networks. *Science*, 286:509–12, 1999.

[5] B. Berger, D. B. Wilson, E. Wolf, T. Tonchev, M. Milla, and P. S. Kim. Predicting Coiled Coils by Use of Pairwise Residue Correlations. *Proceedings of the National Academy of Sciences*, 92:8259–63, 1995.

[6] T. Chano, K. Kontani, K. Teramoto, H. Okabe, and S. Ikegawa. Truncating mutations of *RB1CC1* in human breast cancer. *Nature Genetics*, 31:255–8, 2002.

[7] C. M. Deane, L. Salwinski, I. Xenarios, and D. Eisenberg. Protein interactions: Two methods for assessment of the reliability of high-throughput observations. *Molecular and Cellular Proteomics*, 1(5):349–56, 2002.

[8] M. Delorenzi and T. Speed. An HMM model for coiled-coil domains and a comparison with pssm-based predictions. *Bioinformatics*, 18:617–25, 2002.

[9] K. L. Durst, B. Lutterbach, T. Kummalue, A. D. Friedman, and S.W. Hiebert. The inv16 Fusion Protein Associates with Corepressors via a Smooth Muscle

Myosin Heavy-Chain Domain. *Molecular and Cellular Biology*, 23(2):607–19, 2003.

[10] Y. Ho, A. Gruhler, A. Heilbut, G. D. Bader, L. Moore, S. Adams, A. Millar, P. Taylor, K. Bennett, K. Boutilier, L. Yang, and C. Wolting et al. Systematic identification of protein complexes in *Saccharomyces cerevisiae* by mass spectrometry. *Nature*, 415:180–3, 2002.

[11] T. Ito, K. Tashiro, S. Muta, R. Ozawa, T. Chiba, M. Nishizawa, K. Yamamoto, S. Kuhara, and Y. Sakaki. Toward a protein-protein interaction map of the budding yeast: A comprehensive system to examine two-hybrid interactions in all possible combinations between the yeast proteins. *Proceedings of the National Academy of Sciences*, 97:1143–7, 2000.

[12] R. Jansen, H. Yu, D. Greenbaum, Y. Kluger, N. J. Krogan, S. Chung, A. Emili, M. Snyder, J. F. Greenblatt, and M. Gerstein. A Bayesian networks approach to predicting protein-protein interactions from genomic data. *Science*, 302:449–53, 2003.

[13] H. Jeong, S. P. Mason, A.-L. Barabási, and Z. N. Oltvai. Lethality and centrality in protein networks. *Nature*, 411:41–2, 2001.

[14] H. Jeong, B. Tombor, R. Albert, Z. N. Oltvai, and A.-L. Barabási. The large-scale organization of metabolic networks. *Nature*, 407:651–4, 2000.

[15] T. Jiang and A. E. Keating. Systematic annotation of the saccharomyces cerevisiae proteome using computational integration of genomic data. *Submitted*, 2004.

[16] A. Lupas, M. Van Dyke, and J. Stock. Predicting Coiled Coils from Protein Sequences. *Science*, 252:1162–4, 1991.

[17] S. Mangan and U. Alon. Structure and function of the feed-forward loop network motif. *Proceedings of the National Academy of Sciences*, 100(21):11980–5, 2003.

[18] S. Mangan, A. Zaslaver, and U. Alon. The Coherent Feedforward Loop Serves as a Sign-sensitive Delay Element in Transcription Networks. *Journal of Molecular Biology*, 334:197–204, 2003.

[19] H. W. Mewes, D. Frishman, U. Güldener, G. Mannhaupt, K. Mayer, M. Mokrejs, B. Morganstern, M. Münsterkötter, S. Rudd, and B. Weil. MIPS: a database for genomes and protein sequences. *Nucleic Acids Research*, 30(1):31–4, 2002.

[20] R. Milo, S. S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: Simple building blocks of complex networks. *Science*, 298:824–7, 2002.

[21] J. M. Montoya and R. V. Solé. Small World Patterns in Food Webs. *Journal of Theoretical Biology*, 214:405–12, 2002.

[22] A. G. Murzin, S. E. Brenner, T. Hubbard, and C. Chothia. Scop: a structural classification of proteins database for the investigation of sequences and structures. *Journal of Molecular Biology*, 247:536–40, 1995.

[23] E. Ravasz, A. L. Somera, D. A. Mongru, Z. N. Oltvai, and A.-L. Barabási. Hierarchical Organization of Modularity in Metabolic Networks. *Science*, 297:1551–5, 2002.

[24] E. Roccato, S. Pagliardini, L. Cleris, S. Canevari, F. Formelli, M. A Pierotti, and A. Greco. Role of TFG sequences outside the coiled-coil domain in TRK-T3 oncogenic activation. *Oncogene*, 22:807–18, 2003.

[25] L. Salwinski and D. Eisenberg. Computational methods of analysis of protein-protein interactions. *Current Opinion in Structural Biology*, 13:377–82, 2003.

[26] P. Shannon, A. Markiel, O. Ozier, N. S. Baliga, J. T. Wang, D. Ramage, N. Amin, B. Schwikowski, and T. Ideker. Cytoscape: A Software Environment

for Integrated Models of Biomolecular Interaction Networks. *Genome Research*, 13:2498–504, 2003.

[27] S. S. Shen-Orr, R. Milo, S. Mangan, and U. Alon. Network motifs in the transcriptional regulation network of *Escherichia coli*. *Nature Genetics*, 31:64–8, 2002.

[28] P. Uetz, L. Giot, G. Cagney, T. A. Mansfield, R. S. Judson, J. R. Knight, D. Lockshon, V. Narayan, M. Srinivasan, P. Pochart, A. Qureshi-Emili, Y. Li, B. Godwin, D. Conover, T. Kalbfleisch, G. Vijayadamodar, M. Yang, M. Johnston, S. Fields, and J. M. Rothberg. A comprehensive analysis of protein-protein interactions in *Saccharomyces cerevisiae*. *Nature*, 403:623–7, 2000.

[29] T. Urano, T. Saito, T. Tsukui, M. Fujita, T. Hosoi, M. Muramatsu, Y. Ouchi, and S. Inoue. Efp targets 14-3-3σ for proteolysis and promotes breast tumour growth. *Nature*, 417:871–5, 2002.

[30] J. Walshaw and D. N. Woolfson. Socket: a program for identifying and analysing coiled-coil motifs within protein structures. *Journal of Molecular Biology*, 307:1427–50, 2001.

[31] E. Wolf, P. S. Kim, and B. Berger. Multicoil: a program for predicting two- and three-stranded coiled coils. *Protein Science*, 6:1179–89, 1997.

[32] I. Xenarios, L. Salwinski, X. J. Duan, P. Higney, S. Kim, and D. Eisenberg. DIP, the Database of Interacting Proteins: a research tool for studying cellular networks of protein interactions. *Nucleic Acids Research*, 30(1):303–5, 2002.

[33] A. Zanzoni, L. Montecchi-Palazzi, M. Quondam, G. Ausiello, M. Helmer-Citterich, and G. Cesareni. MINT: a Molecular INTeraction database. *FEBS Letters*, 513(1):135–40, 2002.

[34] H. Zhu, M. Bilgin, R. Bangham, D. Hall, A. Casamayor, P. Bertone, N. Lan, R. Jansen, S. Bidlingmaier, T. Houfek, T. Mitchell, P. Miller, R. A. Dean,

M. Gerstein, and M. Snyder. Global Analysis of Protein Activities Using Proteome Chips. *Science*, 293:2101–5, 2001.