

Contents

List of Figures	vi
List of Tables	viii
Preface	ix
Abstract	xi
1 Introduction	1
1.1 Background and Motivation	1
1.2 Objectives	2
1.3 Related Work	2
2 Method	5
2.1 Methodological Foundation	5
2.2 Statistical Methods	6
2.3 Validity	6
2.4 Reliability and Replicability	6
2.5 Data Collection	7
2.6 The Research Process	7
3 Analysis	9
3.1 Modelling the Problem Area	9
3.1.1 Comparing Neighborhoods	10
3.2 Data Sources	11

3.2.1	Protein–protein interactions: BIND	11
3.2.2	Protein Clustering: Clusters of Orthologous Groups (COG)	13
3.2.3	Protein Annotation: Gene Ontology (GO)	13
3.2.4	Data Abstraction	16
3.3	Requirements	16
4	Algorithms	19
4.1	The Three Perspectives	19
4.2	Networks with Unique Protein Labelling	21
4.2.1	Measuring Neighborhood Similarity	21
4.2.2	Discussion	24
4.3	Networks with Categorical Protein Labelling	24
4.3.1	Measuring Neighborhood Similarity	25
4.3.2	Distance Metrics	26
4.3.3	Indexing	28
4.4	Networks with Structured Categorical Protein Labelling	28
4.4.1	Measuring Neighborhood Similarity	28
4.4.2	Comparing Neighborhood Members: GO Distance Functions	29
4.4.3	Selecting the Most Interesting GO Category	33
4.4.4	Scoring the Neighborhoods	38
5	Implementation	41
5.1	Overview	41
5.2	BioLogue - The RPC server	42
5.2.1	Directed Acyclic Graph Library for GO	43
5.2.2	Weighted Bipartite Matching	46
5.3	BioGraph - The Interaction Network Explorer	47
5.4	Licensing	49
6	Results	53
6.1	Information Content	53

6.2	Robustness	57
6.2.1	The Test Case	59
6.3	Performance	60
6.4	Optimizations	60
6.4.1	Overestimating Weights	61
6.4.2	Memoization	64
7	Discussion	65
7.1	Discussion of Results	65
7.2	Future Work	67
A	Notation and Terminology	69
B	Protein–protein interaction databases	71
C	Basic graph concepts	73
C.1	Properties of Graphs	73
C.2	Classes of Graphs	74
D	User Manual for BioGraph and BioLogue	75
D.1	Software and hardware requirements	75
D.1.1	Installing Java	75
D.1.2	Installing Python	75
D.1.3	Installing NetworkX	76
D.1.4	Installing our Software	76
D.1.5	Installing Data Sources	76
D.1.6	Configuring the Server	77
D.1.7	Configuring the Client	77
D.1.8	Starting the RPC Server	77
D.1.9	Starting the Client	77
D.2	Using BioGraph	78
D.2.1	The Main Window	78

D.2.2	The PPI explorer	78
D.2.3	Running Plug-ins	80

List of Figures

3.1	Illustration of gene products	11
3.2	Histogram of BIND neighborhood degree	13
3.3	Orthologous genes in a phylogenetic tree.	14
3.4	A subsection of the Molecular Function Gene Ontology tree.	15
4.1	Protein neighborhood similarity	20
4.2	Algorithm 1	22
4.3	Overview of neighborhood similarity calculation	30
4.4	Scoring situations for GO similarity	31
4.5	Protein similarity by Gene Ontology analysis with single category	32
4.6	GO classification count histogram	33
4.7	Protein similarity by Gene Ontology analysis with multiple categories	34
4.8	GO similarity measures	37
4.9	Protein neighborhoods converted to a bipartite graph	39
5.1	Client-server model	42
5.2	Implementations of weighted bipartite matching	47
5.3	Plug-in Java interface	48
5.4	BioGraph	50
5.5	BioGraph system design	51
6.1	Neighborhood score vs. direct score	56
6.2	Genome mutations	58
6.3	Mutation resistance	59

6.4	Optimization by overestimation	62
6.5	Expected theoretical speedup factors	63
6.6	Histogram: Speedup/real score ratios	64
7.1	Different size neighborhoods	66
7.2	Neighborhood / annotation count plot	66
C.1	An example graph.	73
C.2	Examples of graphs	74
D.1	One expanded protein connected to three collapsed proteins	78
D.2	Tooltip showing information about a protein	79
D.3	Three selected proteins	79

List of Tables

3.1	Dimensions of data sources	12
3.2	Subsection of BIND data source	12
3.3	Subsection of BIND taxonomy information	12
3.4	Subsection of protein GI-to-GO category file	16
3.5	Subsection of GO data source	16
3.6	Abstraction	17
4.1	Correlation coefficients for level similarity	37
5.1	The most time-consuming methods in the first run.	43
5.2	The 4 most time-consuming methods in the second run plus the time of getAcestors	44
5.3	Run-times for comparing two nodes in the CC DAG	45
5.4	Run-times for comparing two nodes in the BP DAG	45
6.1	Direct scores between selected proteins (statistical scoring)	54
6.2	Direct scores between selected proteins (level scoring)	54
6.3	Statistical scoring, one level	55
6.4	Level scoring, one level	55
6.5	Statistical scoring, two levels	55
6.6	Statistical scoring, two levels	57
6.7	Correlation coefficients	57
6.8	Speed of neighborhood comparison	60
A.1	Notation	69

A.2 Terminology	70
---------------------------	----

Preface

The important thing is not to stop questioning. Curiosity has its own reason for existing. One cannot help but be in awe when he contemplates the mysteries of eternity, of life, of the marvellous structure of reality. It is enough if one tries merely to comprehend a little of this mystery every day. Never lose a holy curiosity.

– Albert Einstein

This thesis is the result of a collaboration between students Petter Braute and Jorg Rødsjø. It was written during the fall 2004 and spring 2005 at *Institutt for Datateknikk og Informasjonsvitenskap* (IDI), NTNU, Norway, with Magnus Lie Hetland as main teaching supervisor. We have also recieved invaluable help from Geir Kjetil Sandve from IDI and Finn Drabløs from the Institute of cancer research and molecular medicine. The thesis is our finishing work for a M.Sc. in complex computer systems.

Our work is based on problems in the field of molecular biology, and the reader should have some basic knowledge on this area. A basic understanding of graphs is also needed to get the most out of our report. A short brush-up on graphs can be found in appendix C.

We would like to thank each and every one supporting us throughout the process of writing this thesis. It could not be done without your help, and we owe you our deepest gratitude!

Abstract

We discuss various ways to approach the *guilty by association* concept, identifying three increasingly complex ways to represent protein–protein interaction neighborhoods. We present ways to measure similarities between the neighborhoods, concentrating on requirements, algorithms, speed and where relevant, implementation and testing.

We develop a method that, based on the annotations of neighboring proteins, predicts the functions of unannotated proteins. Under good conditions, the method yields a Spearman correlation coefficient of 0.7 with direct function similarity measurements between proteins. This indicates that it may be a reliable method for protein function prediction. Results from tests show that wide comparisons are slightly time expensive, so future work on optimization would be interesting.

We implement a tool for viewing, exploring and comparing protein–protein interaction neighborhoods. The tool dynamically downloads new data from internationally acclaimed databases, and the tool utilizes our new method for predicting protein function.

New words that we feel need further explanation are written in **bold**. This means that a definition of the word can be found in appendix A.

Chapter 1

Introduction

Education is the best provision for the journey to old age.

– Aristotle

The medical industry relies heavily on developing new drugs for the future – drugs that help us fight the seemingly more and more aggressive diseases we are so vulnerable to. In this struggle for finding new, effective drugs, *target discovery* has become an important method. Simply put, target discovery is about finding proteins that are linked to a given disease. This is not an easy task, but several hypotheses claim that predicting the function of targets may be a step on the way to discovering new drugs. [Knowles and Gromo 2003]

Predicting the function of unknown proteins has been a goal for scientists for a long time. One method for predicting how unclassified proteins behave is based on studying the proteins they interact with instead of the actual "mystery protein" itself.

1.1 Background and Motivation

In the past decades, scientists have completed the sequencing of several key genomes. In recent years, understanding the functions of the genes has turned out to be a very important area of research [Deng et al. 2002], and various approaches to inferring gene and protein function have been produced (see section 1.3 for an overview on related work).

Proteins play an important role in all species, and they often express themselves in groups, cooperating to complete a common objective. These interactions between proteins can occur between proteins with both similar and different functions. Several biotechnical networks have been discovered based on research on interacting proteins, containing both **annotated** and unannotated proteins and the interactions between them. These large networks form the base of our research. Several publicly available databases with protein–protein interactions exist today, and we will take advantage of one such database, and try to predict the function of the many unannotated proteins whose functions are still unknown.

For computer scientists like us, the problem addressed presents an opportunity for research on many interesting methods and techniques applicable on graphs. This thesis is basically a collaborative study of graphs – their structure, complexity and range of uses.

1.2 Objectives

Our goal is to apply tools and methods from IT to analyze networks of protein–protein interactions. We seek to offer new insight for biologists concerning functional connections between proteins in various species, enabling them to do research on genes in, say yeast, and apply the new knowledge on humans. We seek to do this by offering a tool for predicting the function of unannotated proteins.

1.3 Related Work

Apart from various **in vivo** approaches to determining gene and protein function, several **in silico** methods have been introduced in literature and computer systems. In vivo techniques are based on, for example, disabling the expression of specific genes in mice (as the mouse genome is more than 95% homologous to that of humans), followed by an analysis of the outcome. This is done in order to determine the function of these genes [Winston 2005]. In silico approaches are much more related to our kind of research, and range from textmining to detailed analyses of gene products:

- Chromosomal proximity analysis

As mentioned before, sequencing of genes and proteins has been conducted over the past decades, resulting in many genes, and even entire genomes, with fully described gene sequences. Analysing these data gives biologists more information about genes, based on the assumption that genes with similar sequences, and genes encoded close to each other in the DNA behave similarly.

- Phylogenetic proximity analysis

A phylogenetic tree represents the evolution of genes and proteins in various genomes. The mutations are often amino acid-replacements, -insertions and -deletions. Techniques used to estimate the phylogenetic profile of a gene based on its sequence string are used to predict its function, as genes closely related in a phylogenetic tree can be assumed to have similar functionality.

- Gene Ontology proximity

A technique that will prove very similar and helpful to our work is Gene Ontology proximity analysis. As we will describe later, proteins may be annotated in various namespaces classifying their function, the biological processes they participate in, and the cellular components in which they operate. Defining distances between such categories offers a way to measure the similarity between the gene products classified by them. This type of research was being conducted by, amongst others, Ph.D. student Einar Ryeng at NTNU as we wrote our thesis, and this was of great help during our work.

Our work is based on predicting protein function based on protein–protein interaction neighborhoods, but none of the above uses this kind of data to achieve this. The concept of inferring protein function based on its interacting neighbors is often called *guilty by association*. This concept has also been subject to previous research: Deng et al. [2002, 2004] developed a model that uses Markov random fields to find the probability that a given unannotated protein has a given function. Schwikowski et al. [2000] attempted to do the same by analyzing the frequencies of its neighbors having a certain function. Another technique is one described by Hishigaki et al. [2001], applying chi-square statistics. Many of the mentioned methods apply statistical approaches to model the connection between the function of a protein, and that of its neighbors. This thesis will show that our methods are somewhat different, presenting novel methods to achieve our goal.

Chapter 2

Method

There is no method but to be very intelligent.

– T. S. Eliot

We need to be conscious of what choices we make, and how we make them as the work on our thesis progresses. This chapter will describe the methods we used during our work, and the reasons for using them. We also feel the need for a short discussion on the definition of science, how our method relates to it and fits into the context of IT research. Questions regarding validity and replicability are also answered.

2.1 Methodological Foundation

IT research is a diverse subject – encompassing everything from a classical hypothetical-deductive approach, to research more inspired by social sciences. One of the reasons behind this is that within the field, artifacts are both made and studied, and researchers can contribute to both activities. This is opposed to what can be defined as "Natural Science" [March and Smith 1995], e.g. traditional research in physical, biological, social and behavioral domains where researchers only try to understand reality. In information technology, it is possible to both make and study research objects. We are not saying that all research within these fields is natural science, nor are we saying that no IT-research is natural science – only that within IT there is a lot of room for doing something else than understanding reality.

Our method is closer to what has been dubbed *design science*. This is research which *"attempts to create things that serve human purposes"* [March and Smith 1995]. Our "human purposes", (i.e., the motivation behind our thesis) can be found in sections 3.3 (requirements) and 1.1 (background and motivation). Of course, not all creation of artifacts that serve human purposes is research. What makes it *design science*, is that it produces and applies knowledge of tasks or situations in order to create effective artifacts. If we accept science as the production of credentialed, i.e. documented knowledge, then design science must be an important part of it. Our work has, for a large part, been

concerned with building a tool that uses the results of natural science to extract new (and hopefully credentialed) knowledge from protein-interaction research.

Design-research can be divided in two phases – building and evaluating. Building is the process where the artifact is constructed for a given purpose. Evaluation is the process of finding out how well it performs. The practical part of our build-process is described in chapter 5 (implementation). The ideas behind the implementation and the construction of the implemented algorithms are described in chapter 4 (algorithms). We have to a certain degree performed our own evaluation (see chapter 6), but since we were not the originators of the problem to be solved, it is impossible for us to give a fair evaluation. This is because evaluation is related to intended use, and we are not the intended users. It would take an expert biologist to give the program a fair evaluation. Our advising biologist has been closely involved in the design-process and, to a certain extent, the evaluation process. It is fair to assume that our tests are not completely irrelevant, but still, they may not be good enough to make any real judgement of how well our solution solves the stated problem.

2.2 Statistical Methods

We were drawn to statistical resampling methods by their elegant simplicity and our access to reasonably fast computers. These methods involve either sampling or scrambling the original data a large number of times. What we have used are randomization tests. They are based on scrambling the order of the original data. To give a quick example, we can imagine that we have two sets of numbers, and that comparing them gives a certain correlation score. What randomization involves is to randomly scramble one of the sets, check for correlation, and see how it compares to the original correlation coefficient. This is repeated many times. This will yield a measurement of what the chances of obtaining a correlation coefficient which is equal or better to the original one by pure chance. The higher the number of scrambling and re-calculation of correlation coefficients, the more sure you can be of the result. For a more complete discussion on such methods, see J.Manly [1997].

2.3 Validity

A common questing when engaging in research is one of validity. That is - what do we think we are measuring, and is it what we are looking for? The first question is easy to answer: We think we are measuring protein similarity. Are we actually measuring this? As noted later in this chapter, this is hard for us to judge. The test we have done in chapter 6 give some evidence that we are on the right track, but as discussed in section 7.1, we are aware of the possibility that the measurements may in some cases be less relevant.

2.4 Reliability and Replicability

Will someone else, doing what we have done in this thesis, get the same results? Our algorithms should be explained in enough detail for anyone to reproduce if someone should want to doublecheck the output from our implementation. Another solution would be to compare the algorithms in the

report to the actual implementation. We have included the source code to all of our programs on the attached CD, to make it possible to compare the code to the description of the algorithms. However, replicating our exact results will be impossible as our input data is updated on a regular basis and we have not kept archives of the old versions. Still, the results should be very similar.

2.5 Data Collection

All the data in our thesis come from external databases – mainly BIND and GO (see sections 3.2.3 and 3.2.1). This means we have no control over the input data. For us as non-biologists, this makes life a lot easier when designing our software. We are aware that the data may not be perfectly accurate, but this is beyond our control, and we can not do anything beyond stating the uncertainty, and thus our results. This being said, we consider our data sources to be fairly good and under constant improvement. Our algorithms will therefore give increasingly better results as the interaction data becomes "better", i.e., more accurate.

2.6 The Research Process

At the beginning of the thesis we knew very little about the problem area. We did know a bit about graphs, but that was about it. Our first step was to brush up on our knowledge of biology. This was mostly done with the help of *The Cartoon Guide to Genetics* [Gonick and Wheelis 1991]. The next step involved getting to know what the problem area was, and what our advisors saw as our assignment. This was a long and iterative process. We had semi-regular meetings with our advisors for over a year where we discussed various aspects of the problem and its solutions. Many ideas were proposed, and many were rejected. At first we worked with fairly simple ideas. Then we moved on to more interesting and complex ones as we became more proficient within the problem domain.

Abstracting the data was a fairly simple task, as representing protein-interactions as a graph is a very common and obvious strategy. Abstracting the problem into the general idea of comparing neighborhoods of proteins was a part of our advisors' idea from the start, so we did not ponder this for too long either. The problem we were left with was how to compare the neighborhoods of different proteins. What we have come up with as a solution to this problem is presented in this thesis. Regarding the question of how we have come up with it is more difficult. Algorithm construction and software design have been described as more of an art than an engineering task [Gomes et al. 2002; Warr and O'Neill 2005], and thus it is hard to describe in detail. This art is some times associated with, or referred to, as creativity [Streitz et al. 1999]. One aspect of this thesis which we feel has been helpful in this regard, is having worked in a fairly heterogenous team in which participation was encouraged. As Warr and O'Neill [2005] says: *"the larger the number of ideas produced, the greater the probability achieving an effective solution."*

Chapter 3

Analysis

Nearly all men can stand adversity, but if you want to test a man's character, give him power.

– Abraham Lincoln

Our first step to understanding the problem at hand was to analyse what kind of data we had at hand, the structure of the problem itself, and take it from there. This chapter introduces the data sources we used in our project, introduces various ways to look at the problem, and finally debates what sort of answers we are really looking for.

For example, how can neighborhoods be compared to each other? How do we even compare the neighbors themselves? And what does it really matter if two proteins have similar neighborhoods?

3.1 Modelling the Problem Area

In biotechnology, the guilty by association principle is based on the way proteins work in living organisms: They often express themselves in groups, as proteins only perform bits and pieces of the overall biological process they are participating in. It is therefore plausible that these interacting proteins have functions that are related (but not necessarily similar) to each other. This assumption is the basis of our entire thesis, and we can only embrace this as the truth at the current moment, and use our results to see whether it was in fact a plausible idea. As described in chapter 2, this thesis is a learning process, and we don't have all the facts available from the beginning.

Instead of thinking of protein-protein interaction networks, let us for a while consider a human-human friendship network. What we want to do is to find out what kind of person A is based on knowledge about his friends. A naïve approach would be to assume that however they behave, so does A. In our case, though, this is too much of a simplification. We cannot assume that if a protein interacts with another protein, they have the same functionality. Similarly, in our friendship analogy, we cannot assume that A is a white sheep just because his friends are.

It would seem we need a model to simulate the neighborhoods and the functions of proteins in it.

This could for example be achieved by neural networks learning what protein–protein interaction neighborhoods normally look like. We have, however, used another approach where we do not need to model these connections. The basic idea is to find a protein with a similar neighborhood, and assume similarity between this one and our unknown protein. We will describe this in mathematical notation:

We let \mathcal{F}_α and \mathcal{F}_β denote the functions of the two proteins α and β . Further, we let $N(\cdot)$ return the neighborhood of a protein, and finally, $f(\cdot)$ models the relationship between the function of a protein and its neighborhood. So, for the two proteins α and β , we have...

$$\begin{aligned}\mathcal{F}_\alpha &= f(N(\alpha)) \\ \mathcal{F}_\beta &= f(N(\beta))\end{aligned}$$

Now, what if the only thing we know about α is its neighborhood? Theoretically, we can, to a certain extent, reveal the functions of α by looking at its neighborhood and a similar (\simeq) neighborhood (that of β):

$$N(\alpha) \simeq N(\beta) \Rightarrow \mathcal{F}_\alpha \simeq \mathcal{F}_\beta$$

So, the conclusion so far is that since the neighborhood of a protein α is believed to reveal information (but not necessarily similarity) about the functions of α , the similarity between the neighborhood of α and the neighborhood of β should tell us something about the similarity between α and β themselves. Therefore, this thesis will be an attempt to measure similarities between pairs of neighbors, those of α and β .

3.1.1 Comparing Neighborhoods

We have now stated that we can reveal functions of a protein α by finding a protein β with a similar neighborhood. This, in turn, means that we need to find a way to compare neighborhoods. We define a neighborhood by the set of vertices within a maximum distance of n steps away from a given vertex, which we will call the **base** of the neighborhood. For example, we call the neighborhood of first order $N^1(\alpha)$, meaning the set of vertices that are directly connected to the base α . We also define $N_i^n(\alpha)$ as the i 'th neighbor in the n degree neighborhood of α . Expanding the neighborhood, we can define the neighborhood of second order as described in equation 3.1, simply meaning the set of vertices either directly connected to α , or connected to it by 2 steps.

$$N^2(\alpha) = N^1(\alpha) + \bigcup_i [N^1(N_i^1(\alpha))] \quad (3.1)$$

Various ways of comparing neighborhoods will be an important issue to be addressed throughout the rest of our thesis. We will look at various approaches to compare neighborhoods, as a naïve approach is likely to result in little or no biological relevance. One example of over-simplifying is considering neighborhood similarity as "binary", that is either equal or not. Due to data sources with varying levels of detail and accuracy, this way of comparing neighborhoods will probably yield no interesting results. We will still describe this way of comparing neighborhoods in section 4.2.1

as an introduction to the more advanced ways of modelling neighborhood comparisons. Partial similarity is a more interesting, if not to say the only interesting approach to the problem. However, this adds to the complexity of the problem, with new questions arising, like how to measure the degree of partial similarity, how to penalize differences etc.

3.2 Data Sources

To have something to test out methods on, we need data, and preferably lots of it. For our computations we are first of all dependent on a network containing protein–protein interactions. For some of the calculations we describe later, we also need annotation data for the proteins in the interaction network. Finally, we will also need metadata for the annotations, describing these in detail, revealing connections between the annotations as well. Here we present the data we have been dealing with throughout our project. See figure 3.1 for a simple overview of the data we are working with.

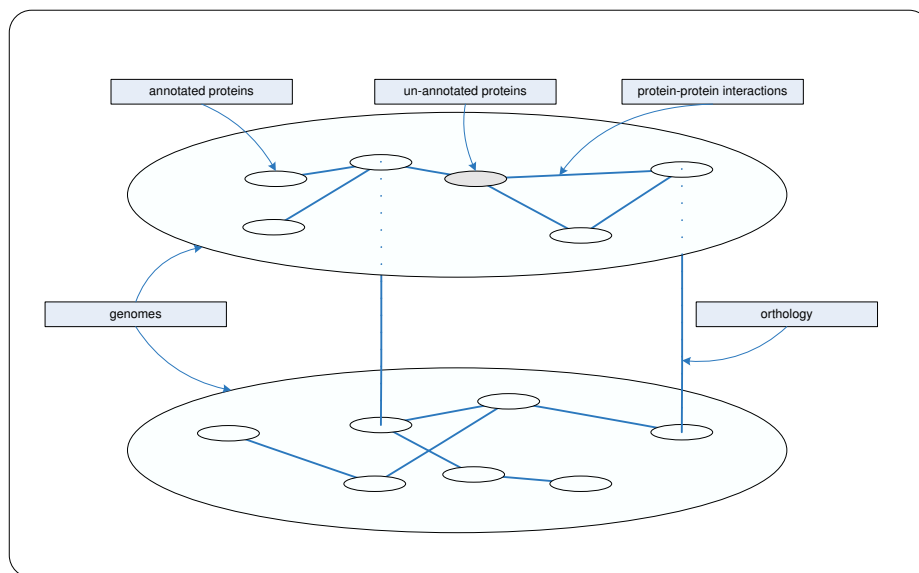


Figure 3.1: Illustration of gene products – genomes, proteins and protein–protein interactions. Some proteins are un-annotated, some are annotated, and some are orthologous.

3.2.1 Protein–protein interactions: BIND

Several publicly available protein–protein interaction databases exist. Appendix B contains a complete list of the databases we have explored during our work. A summary of the dimensions of some of the databases we have explored can be found in table 3.1.

Because of the large amount of data available, we have decided to concentrate on only *one* of these. The Biomolecular Interaction Network Database (BIND) [Bader et al. 2003] has become our database of choice, because (a) we were made aware of the data early in the project, (b) the size of the data

Data source (<i>See Appendix B</i>)	Registered interactions
BIND	64,053
HPRD	25,758
HPID	15,540,018
UniProt/Swiss-Prot	54,964
Fly GRID	28,407
Yeast GRID	25,915
Worm GRID	4,453

Table 3.1: Dimensions of data sources

is not dauntingly large, nor is it so small it becomes uninteresting, (c) the data is stored in an easily readable format and is available through anonymous ftp, and (d) the data is updated regularly, resulting in relevant data for our queries. In fact, BIND contains more than 120,000 interactions, but only 64,053 are protein–protein interactions. Other interactions are interactions between genes, DNA, RNA, small molecules and complexes. We concentrate on the interactions between proteins, and will therefore exclude other interactions from the input data. Table 3.2 shows an example of the data available from BIND. One line represents one interaction between two gene products. Each gene product is uniquely identified by a Gene Identifier (GI). For information about which species the gene products belong to, an additional file is available, connecting the numbers in the row *a-tax* and *b-tax* to the names of species (See table 3.3).

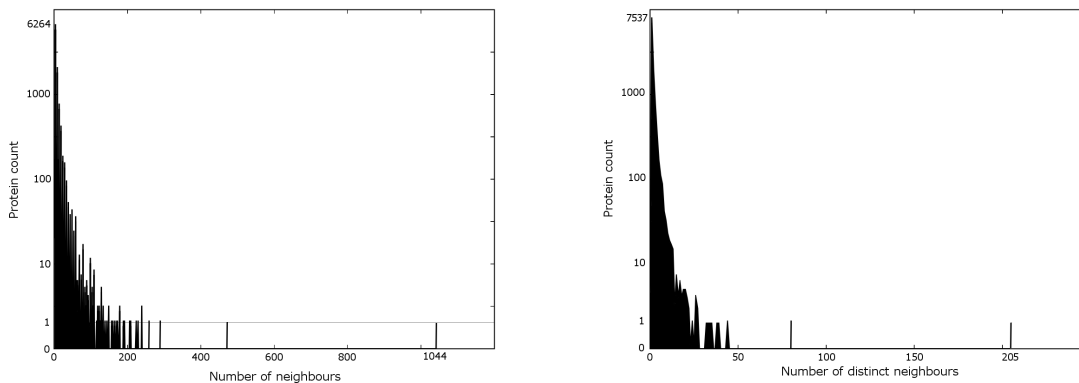
rg-id	b-id	a-type	a-db	a-acc	a-id	a-tax	b-type	b-db	b-acc	b-id	b-tax	ab
510	74259	protein	GenBank	AAA48688	211526	9031	protein	GenBank	AAA48688	211526	9031	yes
510	74258	protein	GenBank	AAA48688	211526	9031	protein	GenBank	AAA48688	211526	9031	yes
510	74257	protein	GenBank	AAA48688	211526	9031	protein	GenBank	AAA48688	211526	9031	yes
511	171207	gene	GenBank	NP_010934	6320855	4932	gene	GenBank	NP_011656	6321579	4932	yes
512	38505	protein	GenBank	NP_572669	24641148	7227	protein	GenBank	NP_650025	21356363	7227	yes
513	221661	small-molecule	BIND	NA	172	0	protein	GenBank	NP_000445	4507149	9606	yes
513	247047	small-molecule	BIND	NA	172	0	protein	GenBank	NP_000445	4507149	9606	yes
514	233626	small-molecule	BIND	NA	17523	0	protein	GenBank	NP_851380	30794362	9913	yes

Table 3.2: Subsection of BIND data source. Columns "a-id" and "b-id" contain gene product identifiers.

tax-code	Species
11065	Dengue virus type 2 (NGC-prototype)
11066	Dengue virus type 2 (strain PR159/S1)
11070	Dengue virus type 4
11072	Japanese encephalitis virus
11073	Japanese encephalitis virus strain SA-14
11077	Kunjin virus
11084	Tick-borne encephalitis virus
11088	Tick-borne encephalitis virus (WESTERN SUBTYPE)

Table 3.3: Subsection of BIND taxonomy information

Figure 3.2 shows the neighborhood degree distribution of the protein–protein interaction subgraph of BIND. The histogram is represented with a logarithmic y-axis for clarity.



(a) 6264 proteins have 5 interacting proteins. 1 protein has 1044 interacting proteins.

(b) Several interactions may be stored in BIND between two unique proteins. 7,537 proteins have only 1 *unique* interacting neighbor protein. The highly connected protein from (a) has 205 unique neighbors.

Figure 3.2: Histogram of BIND neighborhood degree. The highest degree protein is *Tat* from the *Human immunodeficiency virus 1* genome, with a whopping 1044 interacting proteins, or 205 unique ones.

3.2.2 Protein Clustering: Clusters of Orthologous Groups (COG)

As species evolve, proteins change through mutations. Two species, once identical, contain different versions of proteins and genes that were originally the same. Genes and proteins contained in different species, but evolved from the same starting point are called **orthologous** (See figure 3.3). These proteins contain precious information, because we can assume functional similarity between most orthologous genes. *Clusters of Orthologous Groups* (COG) [Tatusov et al. 1997] is a "framework for functional and evolutionary genome analysis", originally consisting of 720 clusters, each cluster containing orthologous individual proteins or sets of **paralogs**.

We haven't studied the structure of COG much during our work, it will merely be used as a theoretical foundation for some scenarios for protein-protein prediction presented in section 4.2.

3.2.3 Protein Annotation: Gene Ontology (GO)

Proteins do not necessarily have to be orthologous to perform the same functions. Classification systems can categorize proteins into groups based on several factors. BIND is updated with data from a multitude of sources including journal submissions, public submissions, reviewing scientific literature and pre-publication datasets. Most of the proteins have one or more classifications in the Gene Ontology classification-system (GO)[Ashburner et al. 2000]. GO makes it possible to compare proteins in one species with proteins from a completely different species, regardless of **homology**. Each protein can be assigned to one or more GO categories. These categories are arranged in a

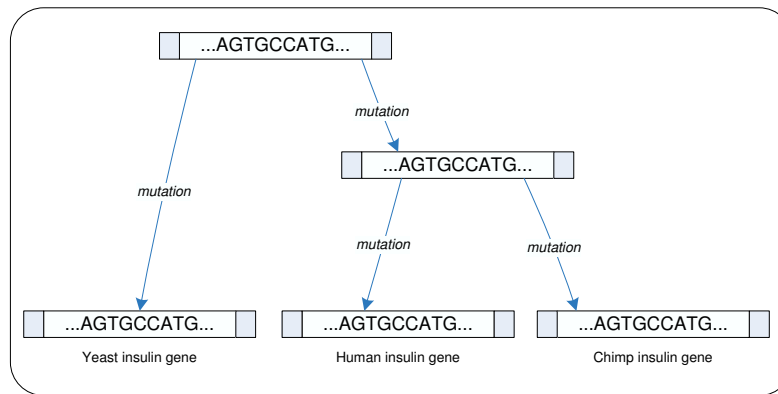


Figure 3.3: Orthologous genes in a phylogenetic tree.

DAG structure, and maintained by the Gene Ontology Consortium. Their goal is:

(...) to produce a controlled vocabulary¹ that can be applied to all organisms even as knowledge of gene and protein roles in cells is accumulating and changing.

This tells us that GO is exactly what we are looking for, namely a cross-species language for classifying genes and proteins.

GO is structured as a graph, with the vertices representing categories, and the edges describing the relationship between them. GO is divided into three separate namespaces, each modelled by its own DAG. The three DAGs are *biological process (BP)*, *cellular component (CC)* and *molecular function (MF)*. The levels in the three Gene Ontology trees describe in increasing detail the function, biological process and cellular component of the categorized protein. As figure 3.4 shows, this means that a protein categorized as *GO:0051288* (meaning its molecular function is NADH binding), automatically implies that the protein in addition, though more general, is a *GO:0050662 coenzyme binding* protein, a *GO:0048037 cofactor binding* protein, and so on.

Namespaces and relationships

The vocabulary of the cellular component ontology consists of terms such as chromosome and nucleus. Cellular components are physical and measurable. A molecular function is defined as *the action characteristic of a gene product*.² In layman's terms it can be described as entities which do not endure but rather occur, for example "decoy death receptor activity" and "lactase activity". Biological process and molecular function are closely related. The Gene Ontology consortium defines a biological process as *"a phenomenon marked by changes that lead to a particular result, mediated by one or more gene products"*, and clarifies by saying that *"a biological process is accomplished via one or more ordered assemblies of molecular functions"*.

¹Whether the GO should be described as an ontology or as a structured vocabulary is open for debate [Smith et al. 2003].

²GO uses "gene products" to refer to any protein or RNA encoded by a gene.

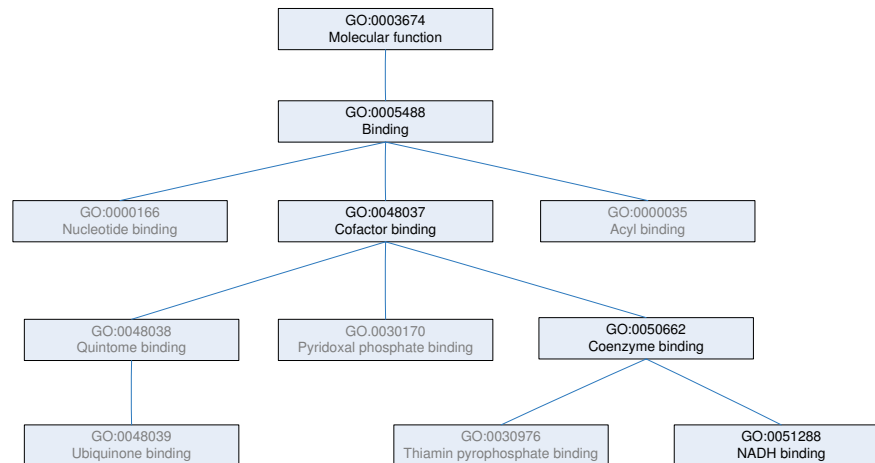


Figure 3.4: A subsection of the Molecular Function Gene Ontology tree.

The relationship between two nodes in a GO DAG can be either *part-of* or *is-a*. Is-a relations are used to describe the relationship between a child term and a parent term, where the child term is an instance of, or a type of the parent term. For instance, *terminal glycosylation* is a subclass of *protein glycosylation* just like a blaster-rifle is a subclass of weapon. Part-of relations are more complicated. GO uses the term in two different ways. The first is *necessarily is-part*. This means that wherever the child exists, it is as part of the parent. Whenever the child exists, so does the parent. But the parent can exist without the child. The second meaning is *necessarily is-part and necessarily has-part*. This means that neither the child nor the parent can exist without the other. The part-of relationships in GO are usually of the type necessarily is-part.

When populating the GO DAGs, we were faced with the problem of what to do with the different links between the nodes. Should we ignore the "part-of" relationship, should we treat it as an "is-a" relationship or should we assign it some special meaning? After consulting our biologist, we decided to ignore the difference between the two types of links, and treat them as equal. Another alternative would be to ignore the "part-of"-relationship altogether. Changing the code to do this would be trivial.

Using GO

Even though GO seems to fit our needs perfectly, it is not without its flaws. One issue is the growth of GO. The bigger GO gets, the more useful it will potentially be to researchers. At the same time, the larger it gets, the harder it becomes to maintain consistency and semantic integrity through manual inspection and curation. Every time someone adds a node in one of the DAGs, the curator needs to understand both the node in question, and its relationship to the entire GO to avoid redundancy and to make sure that the right links are established.

Arguments have also been made against the clear division between the DAGs and the fact that GO uses only two types of links between nodes and that this could lead to loss of possibly useful

information. [Smith et al. 2003]

Since the proteins in the BIND-interactions are classified in the GO-hierarchy, the choice of GO was an obvious one. Flat files linking proteins directly to one or many GO categories are readily available for download, and table 3.4 shows an example of the layout.

GI (Gene Identifier)	GO
14250896	0015036
494929	0005179
494929	0005576
494929	0006006
494929	0007582
17509323	0006730
17509323	0004089
17509323	0008270
17509323	0016829
22095023	0006468

Table 3.4: Subsection of protein GI-to-GO category file

The description of the relations between various GO categories is maintained by the Gene Ontology Consortium, and the definitions can be downloaded in the format shown in table 3.5.

```

id: GO:0000051
name: urea cycle intermediate metabolism
namespace: biological_process
def: "The chemical reactions involving any of the intermediate compounds involved in the urea cycle, a cyclic metabolic pathway that converts waste nitrogen in the form of ammonium to urea." [ISBN:0198506732]
subset: gosubset_prok
is_a: GO:0008152 ! metabolism
relationship: part_of GO:0006807 ! nitrogen compound metabolism

id: GO:0000052
name: citrulline metabolism
namespace: biological_process
def: "The chemical reactions involving citrulline, N5-carbamoyl-L-ornithine, an alpha amino acid not found in proteins." [ISBN:0198506732]
subset: gosubset_prok
is_a: GO:0000051 ! urea cycle intermediate metabolism
is_a: GO:0019794 ! nonprotein amino acid metabolism

id: GO:0000053
name: argininosuccinate metabolism
namespace: biological_process
def: "The chemical reactions involving argininosuccinate, 2-(Nw-arginino)succinate, an intermediate in the ornithine-urea cycle, where it is synthesized from citrulline and aspartate." [ISBN:0198506732]
is_a: GO:0000051 ! urea cycle intermediate metabolism

```

Table 3.5: Subsection of GO data source

3.2.4 Data Abstraction

This far, we have been talking about proteins, protein-protein interactions, genes and genomes. To be able to work on the huge amounts of data present in the databases described in section 3.2, we will look at the data from a more abstract point of view; as vertices and edges in a network. The data is abstracted as described in table 3.6.

3.3 Requirements

The requirements for our project were a set of quite fuzzy specifications. The project was very research-oriented, as opposed to being a strict development-project. Therefore, we are unable to

Biological meaning	Abstraction
Proteins	One protein is identified by its GI identifier, and is represented by one vertex in the graph.
Genomes	All proteins, regardless of genome membership, are stored in one graph. The protein taxonomy is an attribute labelling each protein indicating to which genome it belongs.
Protein–protein interactions	Protein–protein interactions are represented as unweighted, undirected edges in the graphs.
Gene Ontology categories	The GO categories to which a protein belongs are attributes of the vertex in the interaction network. The internal structure of GO categories is also stored in a separate DAG
Gene Ontology category relations	Edges in the GO DAG represent <i>is-a</i> and <i>part-of</i> relations between GO categories.

Table 3.6: Abstraction

state a set of precise requirements, but rather discuss what goals we *thought* we had to achieve. The requirements are the results of discussions with biologist Finn Drabløs.

We have identified three relevant types of searches that the methods we are about to describe should support. The input in all of them is *one* un-annotated protein α , in addition to either (a) the same genome as α is in, (b) a user specified genome, or (c) all genomes available. Some protein–protein interactions are inter-genome interactions (interactions between proteins in different species), and the user should be able to select whether these interactions are included in the search.

We are not interested in exact similarities between neighborhoods, we are rather interested in queries returning neighborhoods with the highest score or smallest distance to that of α . Further, k -nearest neighborhoods searches (finding the k best candidates) are probably the most interesting type of searches.

Finding an optimal way of scoring neighborhoods is not a simple task, there are many considerations to be done. First of all, we have the question of how wide we define the neighborhood to be. Theoretically, all vertices in a connected graph are the neighbors of each other (besides from the vertex itself). It is not hard to see that a query with a neighborhood size this large won't be an interesting query. However, we may assume that neighbors far away from the base of the neighborhood are less interesting than neighbors close to it. This implies that some sort of weighing is most likely the way to go, one that weighs similar neighbors many steps away from the base less than neighbors close to the base.

Level dissimilarities should also be considered. For instance, if both proteins α and β are connected to γ , but by another number of steps, some penalty to the similarity of the neighborhoods should be given. The best match would be if both α and β were the same number of steps away from γ , but as minor flaws in the input data could be enough to corrupt interaction steps (e.g. by faulty experiments), our algorithms should include some way to preserve *some* score even if the distances are unlike. Some weight dependent on the neighbors' differences in distances from the two bases is

likely to be a good solution.

As seen in figures 3.2 (p. 13) and 7.2 (p. 66), the neighborhood degree of the proteins in BIND vary all the way from only one to more than a thousand direct neighbors. This means that we may encounter three different comparisons: Comparing two dense neighborhoods, two sparse neighborhoods, and one dense to one sparse neighborhood. Consider two pairs of proteins about to be compared to each other. The first pair has two densely populated neighborhoods, while the proteins in the other pair both have very sparse neighborhoods. If the resulting similarity-score or -distance is normalized, say by dividing the result by the number of neighbors on which it is based, the two comparisons could easily return similar values. But would this be correct? It could result in comparisons based on very slim neighborhoods (say, with only one connected neighbor) could return unjustifiably high similarities. Still, not normalizing results is likely to blindly favour large neighborhoods in an additive scoring-scheme (as more neighbors will add to the score). This is a dilemma that will be considered later.

Chapter 4

Algorithms

I have yet to see any problem, however complicated, which when you looked at in the right way, did not become still more complicated.

– Poul Anderson

After the analysis of the problem and available data in chapter 3, we will now present our methods for neighborhood similarity searches. This chapter consists of three main parts in addition to some introductory material. These three main sections describe our results based on three different scenarios with increasing levels of complexity. The first perspective describes neighborhood similarity searches with uniquely labelled proteins and the use of COG. The two next perspectives are more interesting, and take advantage of GO data to achieve the same. Only results from the last perspective will be used in implementations and testing, as the two former are meant as introductory material to the third.

This chapter contains formulas and a style of writing that requires the reader to understand the notation described in Appendix A. We recommend referring to this while reading chapter 4.

4.1 The Three Perspectives

We will outline three perspectives for neighborhood similarity searches, and will in the following sections describe how we propose our neighborhood similarity-problem be solved in the individual scenarios. The three perspectives demand increasingly complex solutions, but they are also increasingly interesting in nature. The two first perspectives are mainly meant as introductory material to the third, which will be described in higher detail. This is a result of our iterative approach described in chapter 2. Still, reading sections 4.2 and 4.3 is important for understanding section 4.4.

The structure of the interaction network is the same in all three perspectives, but more information about the proteins is added in (2) and (3). The main idea is the same in all perspectives: As described in section 3.1, we will assume functional similarity between proteins α and β if their neighborhoods are similar. The difference between the three scenarios is how to calculate similarity between the neighborhoods.

1. Networks with Unique Protein Labelling (section 4.2)

When looking at the protein–protein interaction network with only unique labelling of proteins, we will use COG to compare the neighborhoods. For each neighbor to a protein φ , we will try to find ortholog proteins in another genome, and assume functional similarity for proteins connected to these orthologs. This scenario will probably not yield much interest, as we believe too few matches will occur to assume functional similarity of the bases. Still, this chapter is meant as an introduction to our concepts of neighborhood similarity searches.

2. Networks with Categorical Protein Labelling (section 4.3)

Categorical labelling of nodes opens for more interesting searches. Proteins in neighborhoods will in this scenario be treated as similar if they belong to the same GO category. Many neighbors in the same GO classes will give high degrees of similarity between the two neighborhoods.

3. Networks with Structured Categorical Protein Labelling (section 4.4)

In the third, and most interesting perspective, we will add structural information about the GO categories, meaning we can determine degrees of protein similarity based on the relative nearness of the various GO categories.

Figure 4.1 explains what we wish to achieve, and what can be achieved by the three perspectives. Figure 4.1a illustrates the problem common for our entire thesis. We have a protein (here labelled "?") of which we know little - only which proteins it interacts with. We cannot assume any function based on the functions of the neighboring proteins alone. Figure 4.1b illustrates what we can achieve by perspectives 1 and 2 - finding neighborhoods with *equal* function. Finally, figure 4.1c describes our ultimate goal: Assigning protein function based on functionally *similar* neighbors.

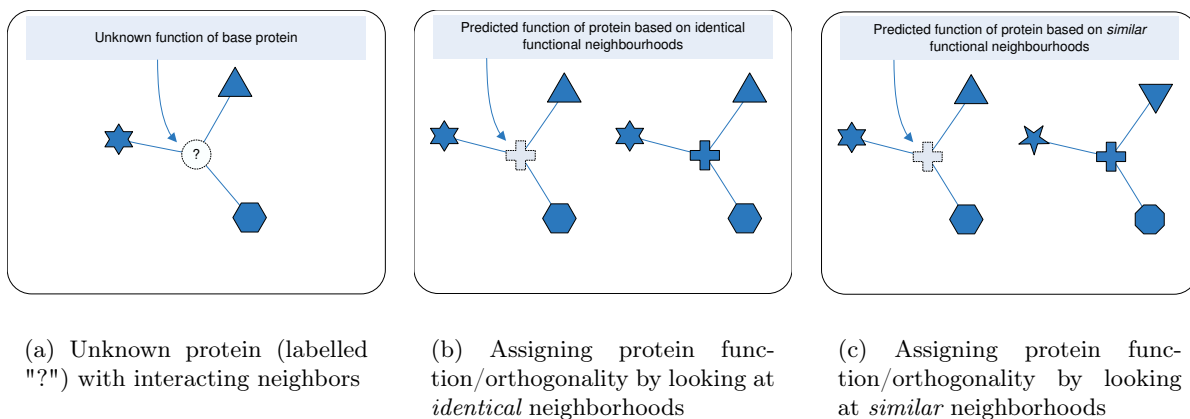


Figure 4.1: Protein neighborhood similarity. The various shapes of the proteins illustrates different functions.

4.2 Networks with Unique Protein Labelling

This first scenario is a fairly simple one: We wish to consider proteins with many orthologous neighbors as more similar than those with few orthologous neighbors. This chapter will be a rather theoretical approach to a solution. As a simplification of the COG database, we will assume that we have available a function $C_{\mathcal{Q} \rightarrow \mathcal{S}}(\alpha)$ that, given the input (a protein α that belongs to the genome \mathcal{Q}), returns the protein from the genome \mathcal{S} that belongs to the same COG as α . We assume that this can be achieved in constant time, for instance by a lookup in a hash table.

4.2.1 Measuring Neighborhood Similarity

Consider a "mystery" protein φ of which we know very little. We wish to reveal information about its function in the genome \mathcal{Q} . We do not have any known orthologs, we only know which proteins it interacts with.

The first problem we wish to solve is to find proteins with identical N^1 neighborhoods to that of φ , in a given genome \mathcal{S} . As this requires the potential targets to have neighbors in the same COG as those in the neighborhood of φ , the vast majority of the nodes in the genome we are searching will not be of interest. A naïve approach would sequentially compare all neighborhoods in \mathcal{S} , but taking advantage of the constraints we have, we can perform this search a lot faster: We can simply start from all elements in $N^1(\varphi)$, distributing points to all of their neighboring nodes in the target genome:

Algorithm 1

```

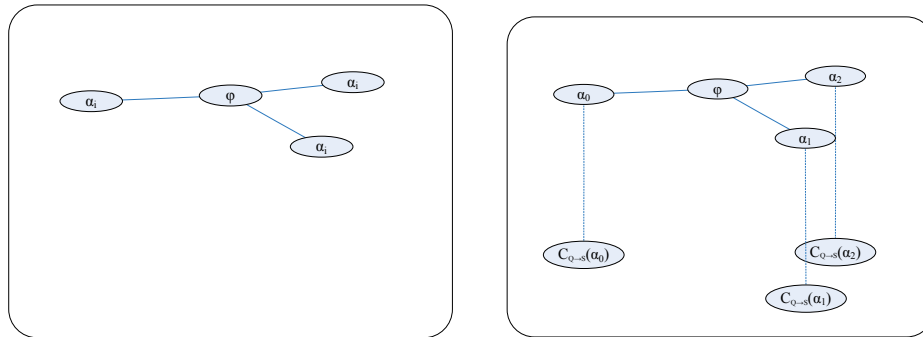
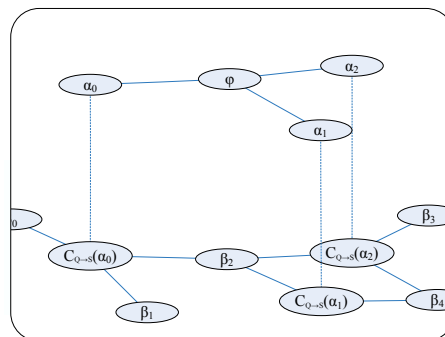
potentials = array()
result = array()
for each  $\alpha$  in  $N^1(\varphi)$ 
    for each  $\beta$  in  $N^1(C_{\mathcal{Q} \rightarrow \mathcal{S}}(\alpha))$ 
         $P(\beta) += 1$ 
        potentials.put( $\beta, P(\beta)$ )
for each  $\beta$  in potentials
    if  $P(\beta) = |N^1(\varphi)|$ 
        result.put( $\beta, P(\beta)$ )

```

Algorithm 1 returns the array **result**, in which all members would be a protein with a neighborhood completely orthologous¹ to that of φ . In the example in figure 4.2, β_2 has received points from all three $C_{\mathcal{Q} \rightarrow \mathcal{S}}(\alpha_i)$, leaving it with three points, meaning all its neighbors are orthologous to one of φ 's neighbors. What we have done here is to substantiate that φ and β_2 have functional similarity, based on the fact that their neighbors are orthologous (and thus functionally similar).

Nevertheless, finding proteins in a genome with a *completely* identical orthologous neighborhood to a query node is not very interesting. This is because the protein-protein interaction networks

¹By an orthologous neighborhood, we mean that all the proteins in the neighborhoods are pairwise orthologous.

(a) Find $N^1(\varphi)$ (b) Find $C_{Q \rightarrow S}(\alpha)$ (c) Find $N^1(C_{Q \rightarrow S}(\alpha))$, add 1 point to each neighborFigure 4.2: Algorithm 1: Finding proteins in \mathcal{S} with orthologous neighborhoods to φ

often are based on experiments with far from perfect results. Interactions are often either missing, superfluous or inaccurate. Attempting to make a more interesting model of the problem, we will now extend this algorithm to find neighborhoods that are *partially* similar to that of the query node. This is shown in algorithm 2a.

Algorithm 2a

```

result = array()
for each  $\alpha$  in  $N^1(\varphi)$ 
  for each  $\beta$  in  $N^1(C_{Q \rightarrow S}(\alpha))$ 
     $P(\beta) += 1$ 
    result.put( $\beta, P(\beta)$ )

```

By using algorithm 2a, **result** will contain all β 's with a score higher than 0. This time, all β 's in figure 4.2 would be in **result**, and we could take note of β_4 as a relatively interesting node as well as β_2 .

However, if a potential hit β_i has more neighbors than φ – neighbors that have nothing to with the neighborhood of φ , its score should be smaller than that of β_j with the exact same size of the neighborhood as φ . We therefore introduce a penalty for superfluous nodes if these are not orthologs to any of φ 's neighbors. This is implemented by penalizing potential β 's after the positive scoring has been distributed. This is explained in algorithm 2b.

Algorithm 2b

```

result = array()
for each  $\alpha$  in  $N^1(\varphi)$ 
  for each  $\beta$  in  $N^1(C_{Q \rightarrow S}(\alpha))$ 
     $P(\beta) += 1$ 
    result.put( $\beta, P(\beta)$ )
for each  $\beta$  in result
   $P(\beta) = 2 * P(\beta) - |N(\beta)|$ 

```

Algorithm 2b is a simple, but powerful improvement from 2a: It subtracts one point from the score for each uninteresting² neighbor the potential hits β have. This is done by subtracting $(|N^1(\beta)| - P(\beta))$ from the score (Uninteresting neighbors = Total number of neighbors - interesting neighbors). So, the total score should be $P(\beta) - (|N^1(\beta)| - P(\beta)) = (2 * P(\beta) - |N(\beta)|)$. Still, we can improve search results by allowing for deeper searches for similar neighbors. We keep the penalty for superfluous neighbors only in the N^1 -neighborhood of the potential orthologs. Our recommendation for this implementation is described in Algorithm 3.

As we can see from algorithm 3, proteins are scored based on how far away from the base in question

²Not orthologous to any of φ 's neighbors

Algorithm 3

```

result = array()
define  $D \in \mathbb{N} \geq 1$  //depth of search
for each  $\alpha$  in  $N^D(\varphi)$ 
    for each  $\beta$  in  $N^d(C_{Q \rightarrow S}(\alpha))$ 
         $d_\alpha = \text{distance from } \alpha \text{ to } \varphi$ 
         $d_\beta = \text{distance from } \beta \text{ to } C_{Q \rightarrow S}(\alpha)$ 
         $P(\beta) += \frac{1}{2}^{(d_\alpha + d_\beta - 2)}$ 
        result.put( $\beta, P(\beta)$ )
for each  $\beta$  in result
     $P(\beta) = 2 * P(\beta) - |N(\beta)|$ 

```

they are, halving the distributed score for each step. The falloff-factor of $\frac{1}{2}$ is not a definite solution, and may be changed in order to achieve better results. The factor is used to simulate the falloff of interest as we traverse the neighborhood levels: Ortholog *direct* neighbors are important, ortholog neighbors farther away give less information, and distribute less points. In algorithm 3, two direct neighbors will yield a score of 1. As in algorithm 2b, **result** will be an array of proteins and their scores.

4.2.2 Discussion

The algorithms just presented is a simple example of how we can find proteins with similar neighborhoods. It has one obvious advantage over the techniques we are about to present - that of speed. The technique used in all algorithms in this section ensure that we find the proteins with similar neighborhoods in $O(n^2)$ time, where n is the size of the neighborhoods. Although $O(n^2)$ may not be a good result in itself, values for n are normally low, and all the algorithms do is to increment a counter for each step, so there is no heavy calculations involved. We therefore predict that these algorithms should run very fast if implemented wisely (for instance, by ensuring that $C_{Q \rightarrow S}()$ functions are constant time lookups, and graph traversal is fast.).

The most important drawback is simply the information content of the results. We suspect that little new information can be found by using these algorithms, because COG information is mainly available on well-known, annotated genomes. In the following chapters, we will therefore leave COG, and base our similarity measures on GO instead.

4.3 Networks with Categorical Protein Labelling

Say we are still interested in finding proteins with similar neighborhoods to φ , but the only thing describing a protein is its categories as described in GO. This immediately poses a problem: Unlike COG, which connects proteins from genomes, GO is a vocabulary used to annotate proteins. This means that we cannot take the neighborhood of φ as a starting point, because there are no direct

connections between it and the proteins we are searching for. Since the potential targets may be anywhere in the search genome, we have to use sequential searches to find them, as long as we don't perform any indexing, of course. Since this perspective is only an introduction to neighborhood similarities, we will not discuss indexing techniques at this point.

In this chapter, we will describe how we can measure the similarity between protein neighborhoods based on their GO categories, with one important simplification: We will not take advantage of the structure of GO. As an effect, we will not be able to assume any similarity between a protein categorized as category \mathcal{A}_i and one classified as \mathcal{A}_j , even if these categories are closely linked in one of the GO DAGs. This is, of course, a restrictive constraint, but we are doing it in order to explore more aspects of protein neighborhood similarity measuring.

4.3.1 Measuring Neighborhood Similarity

As mentioned above, we can't use the GO structure in this perspective. So, the only thing indicating direct protein similarity would be an *identical* GO classification between the two. At present time, there are more than 18,000 GO categories defined. We therefore assume that trying to align proteins with each other based on GO categories alone won't be sufficient, as the quality between various classifications will vary, resulting in proteins that in reality are alike, will receive no score for similarity. This is, however, a problem that will be solved in the next perspective.

We will in this chapter introduce a *distance* based measure of neighborhoods instead of a *scoring* based approach as described in the first perspective.

Because we need to search all proteins for neighborhood similarity, we assume the need for some sort of indexing if our method is to be sufficiently time-efficient (although we will not discuss it in great detail). We will therefore strive to produce *metric* distances, as this makes the use of more indexing methods possible. Distance functions can be defined as metric if they satisfy all following three requirements:

- **Symmetry**

$$d(a, b) = d(b, a)$$

- **Positivity**

$$d(a, b) > 0 \quad a \neq b$$

$$d(a, a) \equiv 0$$

- **Triangularity**

$$d(a, c) \leq d(a, b) + d(b, c)$$

In simple terms, this implies that a function measuring the distance between two points must (a) return the same distance between two points independently of the direction in which we are "travelling", (b) always return distances greater than 0, unless source and destination are the same, and (c) not allow for routes between points that are shorter than the direct route from source to destination. In our case, proteins are the points, and distances between them represent the similarity of their neighborhoods.

The first idea is to generate a binary string based on the first-order neighborhood of a protein α . For each category \mathcal{A}_i , we find out whether α has a neighbor classified as \mathcal{A}_i . If it has, the bit in string position i is a 1. If not, it produces a 0. As mentioned in section 3.2.3, proteins are classified by none, one or many GO categories. So the neighborhood string of protein α would generate n binary 1s for each neighbor, $n \geq 0$. The distance between two proteins (or, more precisely, between their neighborhoods), could be expressed as the hamming distance between their strings. However, the method is in its current state only able to compare direct neighborhoods. We will in the following section describe similar, but slightly more advanced methods.

4.3.2 Distance Metrics

We wish to define a distance function $d(\alpha, \beta)$ returning the distance between two proteins α and β , based on their neighborhood classifications. In stead of generating binary strings, we will now find the distance (in number of interactions) from α to the closest protein classified as a given \mathcal{A}_i to define its position in dimension i . That means, if α and β have the exact similar distances to the closest protein of categories $\mathcal{A}_0, \mathcal{A}_1, \dots, \mathcal{A}_m, m = |GO|$, then $d(\alpha, \beta) = 0$. Increasing functional differences between the proteins will theoretically result in an increasing $d(\alpha, \beta)$.

First, let $\mathcal{A}_0, \mathcal{A}_1, \dots, \mathcal{A}_m$ be the universe of categories in GO. Secondly, let $J(\alpha, \mathcal{A}_i)$ be the smallest number of interaction steps from α to a protein classified as \mathcal{A}_i .

L₁ Distance

If we opt to use the L_1 distance, also known as Manhattan distance, we would end up with the following distance function. All L_p distances are metric.

$$d_{L1}(\varphi, \alpha) = \sum_{i=0}^m (|J(\varphi, \mathcal{A}_i) - J(\alpha, \mathcal{A}_i)|) \quad (4.1)$$

L₂ Distance

Another approach is to use the L_2 distance, the Euclidean distance. We would get...

$$d_{L2}(\varphi, \alpha) = \sqrt{\sum_{i=0}^m [J(\varphi, \mathcal{A}_i) - J(\alpha, \mathcal{A}_i)]^2} \quad (4.2)$$

Selected Sample L₂ Distance

It may not be likely that a complete comparison of all m categories is an interesting approach for biology research. Instead, we assume we want to compare only a selected range of $J(\dots)$ functions, in contrast to comparing all of them. The most interesting ones to compare is likely to be the ones with the lowest values for φ . The number of interesting comparisons is assumed to be dependent on each query case, and should therefore be a parameter $n \in [1, m]$ selected by the user.

First, we define the set S by the following induction:

$$S \subseteq \mathcal{A} \mid (x \in S \wedge y \notin S \Rightarrow J(\varphi, x) \leq J(\varphi, y)) \quad (4.3)$$

The set S now contains the n categories with lowest values for $J(\varphi, \dots)$. We can then define the distance function $d_{selL2}(n, \varphi, \alpha)$:

$$d_{selL2}(n, \varphi, \alpha) = \sqrt{\sum_{i=0}^n ((J(\varphi, S_i) - J(\alpha, S_i))^2, |S| = n)} \quad (4.4)$$

Canberra Distance

Still, our selected distance function has a rather serious flaw: It returns the same distance between two points independent on their relative $J()$ -values. For example, $J(\varphi, S_i) = 2$ and $J(\alpha, S_i) = 4$ gives the same distance as if $J(\varphi, S_j) = 102$ and $J(\alpha, S_j) = 104$. Because of the nature of the protein-protein interaction networks, similar neighbors far away from the protein in question should weigh less than similar neighbors in the immediate vicinity. The *Canberra distance* can be considered a relative Manhattan distance, since it is given by the absolute difference between the values divided by their absolute sum. It is defined in equation 4.5 and seems to fit our needs quite well.

$$D_{canberra}(\varphi, \alpha) = \sum_{i=0}^m \left(\frac{|J(\varphi, \mathcal{A}_i) - J(\alpha, \mathcal{A}_i)|}{|J(\varphi, \mathcal{A}_i) + J(\alpha, \mathcal{A}_i)|} \right) \quad (4.5)$$

Selected Sample Canberra Distance

Like the selected sample L_2 distance, we can derive a selected sample Canberra distance as well, for the same reasons as for selected sample L_2 distance. We still have the set S as defined in equation 4.4. Selected sample Canberra distance will then be defined as...

$$d_{selcanberra}(n, \varphi, \alpha) = \sum_{i=0}^n \left(\frac{|J(\varphi, S_i) - J(\alpha, S_i)|}{|J(\varphi, S_i) + J(\alpha, S_i)|} \right) \quad (4.6)$$

Other Metrics

Other distance functions may also be considered, such as the Squared Chord distance and the Squared Chi-squared distance. We have, however, not looked into these options in any specific detail.

$$D_{squaredchord}(\varphi, \alpha) = \sum_{i=0}^m \left(\sqrt{J(\varphi, \mathcal{A}_i)} - \sqrt{J(\alpha, \mathcal{A}_i)} \right)^2 \quad (4.7)$$

$$D_{\text{squaredchisquared}}(\varphi, \alpha) = \sum_{i=0}^m \left(\frac{(J(\varphi, \mathcal{A}_i) - J(\alpha, \mathcal{A}_i))^2}{|J(\varphi, \mathcal{A}_i) + J(\alpha, \mathcal{A}_i)|} \right) \quad (4.8)$$

4.3.3 Indexing

As this second perspective of our three also is an introduction to the third perspective, we haven't performed any hands-on testing of these distance functions, neither for speed nor biological relevance. Still, it doesn't require a lot of testing to see that a sequential search throughout an entire genome will be time-consuming to say the least. We therefore include a short discussion on potential indexing methods one could use to boost the performance.

BoostMap [Athitsos et al. 2004] is a method that is originally meant to be used in image and video data retrieval. This application does however have a fundamental similarity to our problem: Distance measures are used to range similarities between vertices. The difference is that vertices in our case is proteins, and image-data in their case. The method is used to rapidly find approximate nearest neighbors to a vertex based on preserving some, but not all information about the distances. Without having done any more research on the use of BoostMap, we think that it may very well be of good use if one should choose the distance-metrics-approach to comparing protein neighborhoods.

As with the first perspective, this approach was meant as an introduction to the field of neighborhood comparisons, and we have thus not prioritized improving the methods. The research done on this technique has mainly been a good tool for learning the ins and outs of neighborhood similarity. Finally, in the next section, we arrive at the method we have spent most time on developing. Leaving distances as a measure between neighborhoods, and yet again introducing scoring schemes, we will continue to use GO, but also use GO metadata to perform the calculations.

4.4 Networks with Structured Categorical Protein Labelling

As mentioned, the method in section 4.3 has one serious simplification that needs to be taken into consideration: Because the annotations of various proteins have been proved through experiments with varying precision, the similarity distances will be wrong in many cases: Proteins classified as *Thiamin pyrophosphate binding* will not be treated as similar to proteins classified as *Cofactor binding*, even though these categories are very similar, and indeed closely related in the GO *Molecular Function* DAG.

4.4.1 Measuring Neighborhood Similarity

We now take advantage of all the data available from both BIND and GO. The technique we are about to explain should be robust to differing levels of input quality, and enable the user to choose how deep and accurate searches to run.

Consider two proteins α and β with the neighborhoods $N^1(\alpha) = \{\mathcal{A}_a, \mathcal{A}_b\}$ and $N^1(\beta) = \{\mathcal{A}_c, \mathcal{A}_d\}$, respectively. Our goal is to score the similarity of these neighborhoods to determine whether the two proteins in question can be assumed to have a functional similarity. With the methods described in

section 4.3, none of the categories would be considered equal, and the neighborhoods would yield high distances (based on how far away the identical categories are from the two bases). To improve our results, we will now attempt to make use of the hierarchical arrangement of the GO categories to better describe the similarities between two proteins. With this information, we attempt to score the similarity between the various GO categories, and use these scores to align the neighbors for an optimal neighborhood similarity score. The category \mathcal{A}_a may not be equal to category \mathcal{A}_d , but they may have a certain degree of similarity.

Overview

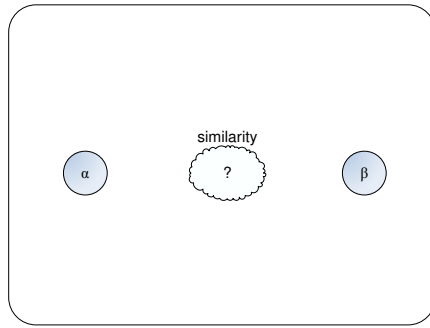
We first give a short version of the technique, since the process is quite intricate. Figure 4.3 shows an overview of the entire process. This should make the detailed description in the following chapters easier to understand. The main idea is to first pair each neighbor of α and β with each other, scoring their individual similarities. Obviously, this is a task that will be performed many times ($O(n^2)$ for comparing two neighborhoods size n), and needs to be computed quickly. Sections 4.4.2 and 4.4.3 explains how this neighbor-neighbor comparison is done. After this, we want to match pairs of neighbors from each neighborhood such as to produce the highest score possible for the two neighborhoods. We do this by building a weighted bipartite graph between the two sets (being the two neighborhoods). This part is described in section 4.4.4. Finally, a function is applied to the score of the matched pairs, returning the final score that will represent the similarity between the neighborhoods. Section 4.4.4 also describes this final step of the process.

4.4.2 Comparing Neighborhood Members: GO Distance Functions

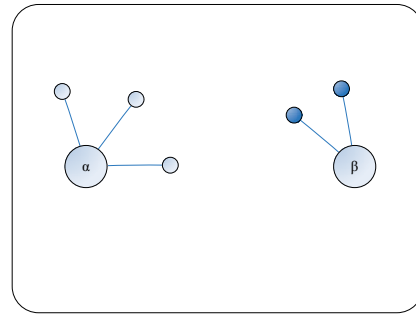
Basically, this first step means building a complete, weighted bipartite graph with α 's neighbors on one side and β 's neighbors on the other. Each edge will be given a weight representing the similarity between the two neighbors it connects. This similarity is based on Gene Ontology, and we will now discuss ways to define it.

We will begin with introducing similarity measures between two *singly classified* proteins, by which we mean that each protein is classified by only *one* GO category. Initially we will begin by describing what output is desired given different conditions: The highest similarity would be the result of the two categories being highly detailed (i.e., far away from the root node), and close to each other. On the other hand, if they were not as detailed (i.e., closer to the root), though still close to each other, a lower score would be expected, as this tells us much less regarding their similarity. Also, even if both categories are highly detailed, but far from each other, a low similarity should be expected as they are not similar at all.

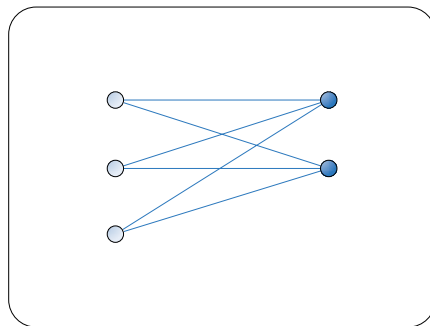
A reasonable approach for obtaining these outputs is to discover what the two GO categories have in common. If the categories belong in the same GO DAG (BB, CC, MF), we can easily find their common ancestors, or more accurately, their common *superordinates*. This is a simple matter of an intersection between the two sets of superordinates. Now, the most *interesting* common superordinate will represent the level of similarity between the two categories, and consequently, between the two proteins. See figure 4.5 for illustration. The most interesting node is found either by its level of detail or statistical significance. Finding the most interesting category is discussed in



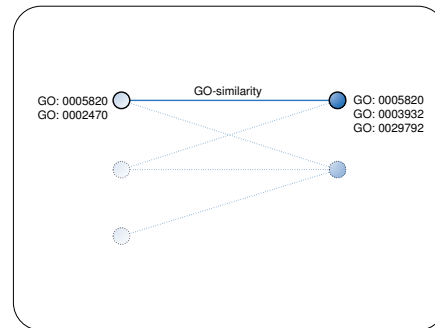
(a) We wish to find the similarity between two proteins α and β .



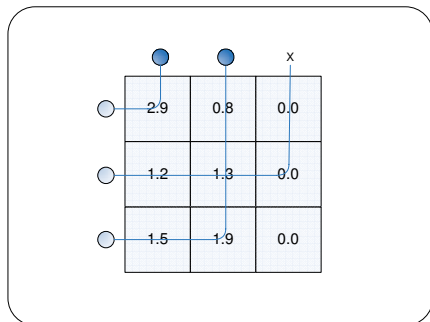
(b) We find their neighbors, in this example only directly connected neighbors.



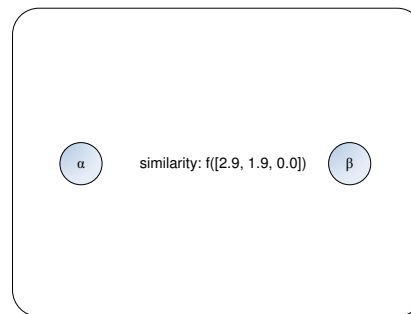
(c) We build a bipartite graph based on the neighborhoods.



(d) Weighing the edges in the graph is done by comparing GO categories in the various neighbors.



(e) Scores are represented as costs in a maximize-cost-version of the **assignment-problem**. Lacking neighbors in one neighborhood yield zeros in the matrix (rightmost column in the example)



(f) The selected costs are fed to a function returning the similarity of the two original neighborhoods. A naïve function could be $f(a_0, a_1, \dots, a_m) = \sum_{i=0}^m a_i$

Figure 4.3: Overview of neighborhood similarity calculation

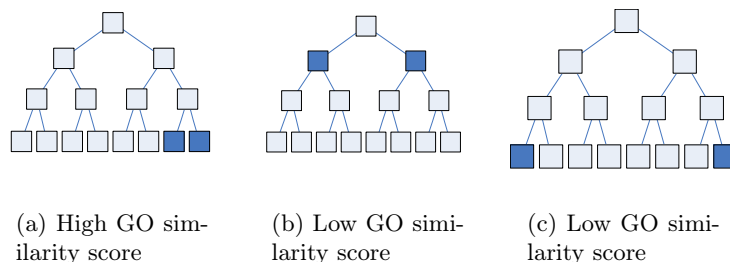


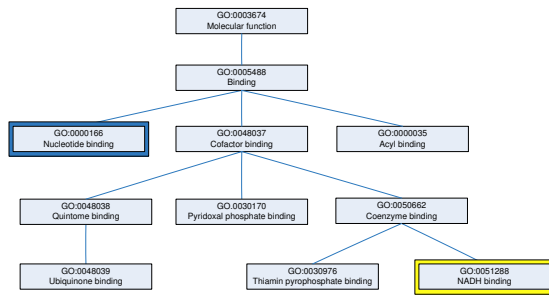
Figure 4.4: Scoring situations for GO similarity. The boxes represent nodes in one of the GO DAGs. The two dark boxes represent the categories of the two proteins.

section 4.4.3, but first we will have to expand our model to include proteins classified by more than one GO category, as most proteins are.

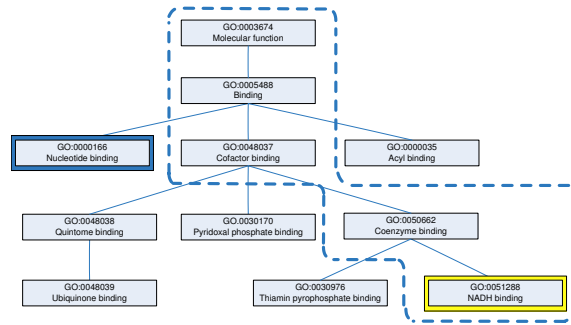
We recall that every protein is classified by none, one or many Gene Ontology categories. With multiply classified neighboring proteins, the question of scoring the proteins gets a bit more complex. How can we now define the similarity between the proteins? What we will do is to select the two *most* similar GO categories and use the common ancestors of these as the measure of protein similarity. The naïve way to find the two most similar categories would be to make an $m \cdot n$ matrix, m = the number of GO categories classifying protein $N_i(\alpha)$, and n = the equivalent for protein $N_j(\beta)$. Now, filling this matrix would involve the GO comparison explained above (and described in figure 4.5) for each matrix cell. Finally, one would choose the best score. Analysing figure 4.6, we quickly find that this might turn out to be a major bottleneck, as the mean number of GO categories per protein in BIND is 6.84. The mean matrix size would be $6.84^2 \approx 47$. Performing the computations described earlier 47 times for each protein pair in the neighborhood we want to score will be hairy at best, especially if we include more than one neighborhood level in the comparisons. We need to find a better approach.

Our refined method for scoring proteins classified as *more than one* category is almost identical to the method for comparing two proteins classified as *one* category. The difference is that instead of finding the superordinates of each protein's category, we find the *union* of all its categories' superordinates. This is a matter of adding all categories and their superordinates into a set. Elements already contained will consequently not be duplicated. The next step is equal to that of the first method: find the intersection of the two sets. Finally, we return the most interesting category from the produced intersection. This method will return the same value as comparing all categories iteratively and returning the best candidate from the matrix.

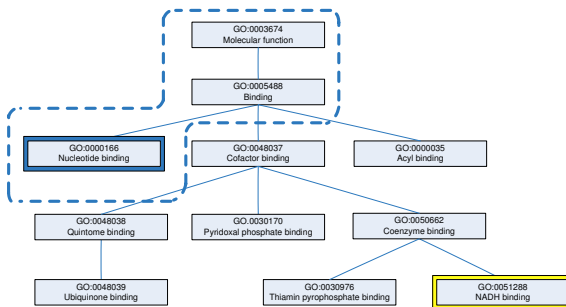
Figure 4.7 describes how two proteins classified as more than one GO-category are scored: We assume the topmost category in the figure is the root in the Molecular Function Gene Ontology tree. (a) shows four marked categories. The yellows are the categories describing protein $N_i(\alpha)$, the blues describe protein $N_j(\beta)$. In (b) and (c), we mark the union of the superordinates of the respective classes, before we in (d) find the intersection of the two sets. This intersection defines the similarity between the two proteins. We can now return the most interesting category from this set, and extract a score value from it.



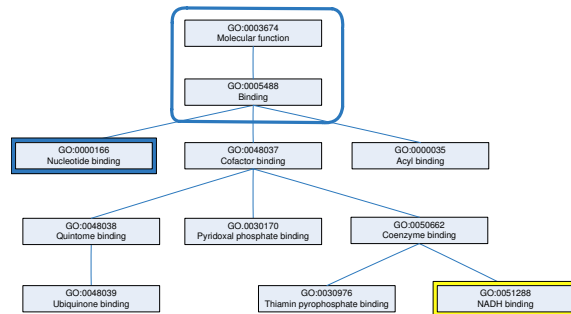
(a) The yellow and blue markers define the categories of proteins $N_i(\alpha)$ and $N_j(\beta)$. Protein $N_i(\alpha)$ is classified as a NADH binding, and protein $N_j(\beta)$ is a Nucleotide binding.



(b) The superordinates of $N_i(\alpha)$'s category.



(c) The superordinates of $N_j(\beta)$'s category.



(d) The intersection of sets produced in (b) and (c), producing the set of common superordinates.

Figure 4.5: Protein similarity by Gene Ontology analysis with single category. The figure shows the calculation of the common superordinates of the two categories. This is done in 3 steps: In (b), we find the superordinates of the first category, and in (c), we do the same for the other category. Finally, in (d), we find the intersection of the results from (b) and (c).

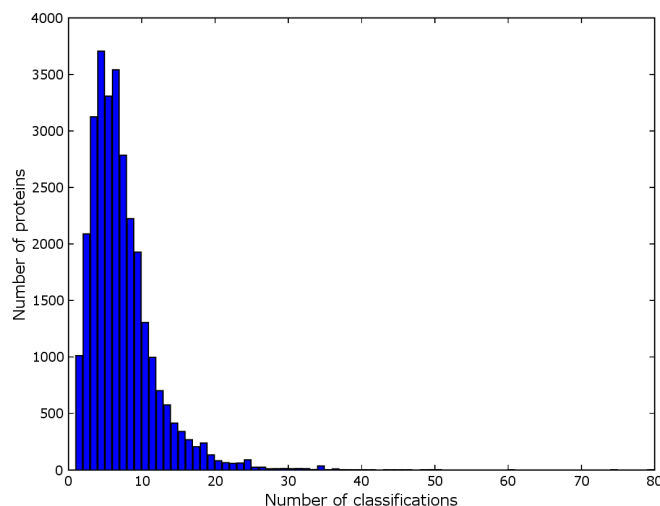


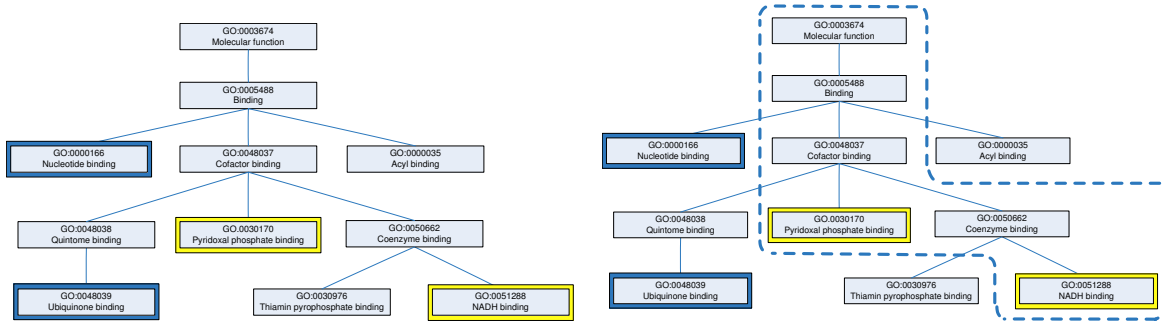
Figure 4.6: GO classification count histogram

4.4.3 Selecting the Most Interesting GO Category

The final step in the scoring of two neighboring proteins is selecting the most interesting node from the set of common superordinates. We have devised two techniques for this: (a) level selection and (b) statistical selection. Level selection is based on the fact that nodes far away from the root node in the GO DAG (i.e., with a high level) bear more information than nodes close to the root. Statistical selection, on the other hand, selects the least probable GO category based on protein annotation counts. We wish for both methods to return a 0 score for the three GO categories *Biological Process*, *Cellular Component* and *Molecular Function*, as no similarity can be deduced from annotations with only these in common. In contrast, proteins with common annotations with a lot of information content will be scored higher.

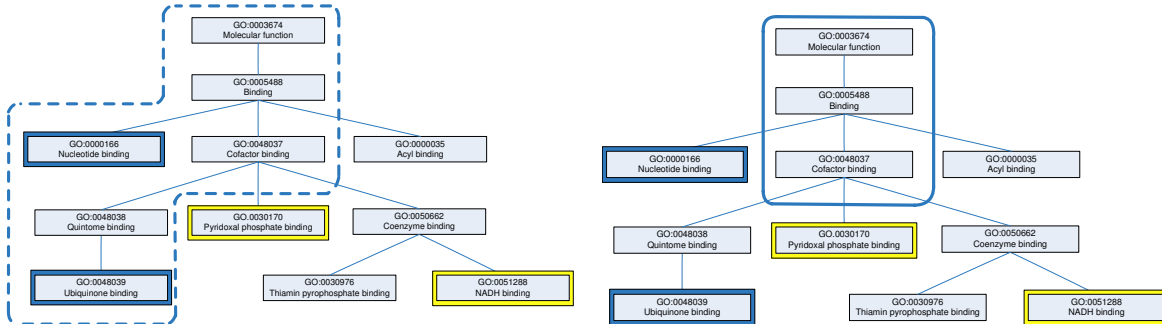
GO Level Selection

Scoring the proteins by highest detail level selection is quite simple, we simply return the level in which the most specific element in A is located (Where A is the set of common superordinates of both proteins). This ensures several things. First, proteins with poorly determined categories (i.e., at a low level in the GO tree) will score low, as will proteins whose categories became separate at an early level in the GO tree (As their most specific common superordinate is at a low level). On the other hand, highly detailed classifications will result in a high similarity if the categories have highly detailed common superordinates. As an improvement to only returning the highest level, we suggest returning the *log* value of the returned value, as the differences become less and less interesting the more detailed they get. A log value would give the user a better feeling of the importance of the scores. In the case depicted in figure 4.7, the similarity would be scored 2 - the level of the most specific category in figure 4.7d. A general definition of this scoring scheme would be as given in equation 4.9.



(a) The yellow and blue markers define the categories of proteins $N_i(\alpha)$ and $N_j(\beta)$. Protein $N_i(\alpha)$ is both a NADH binding and a Pyridoxal phosphate binding. Protein $N_j(\beta)$ is an Ubiquinone binding and a Nucleotide binding.

(b) The union of the superordinates of $N_i(\alpha)$'s categories.



(c) The union of the superordinates of $N_j(\beta)$'s categories.

(d) The intersection of sets produced in (b) and (c).

Figure 4.7: Protein similarity by Gene Ontology analysis with multiple categories. The figure shows the calculation of the common superordinates of the categories. We have implemented both this method and the naïve way described above (comparing all $m \cdot n$ categories iteratively). Our method appeared to be faster than the naïve approach by orders of magnitude. We have measured empirically speedup factors of > 20 by using the improved method.

$$S_{level}(\mathcal{A}_i) = \log_2(\mathcal{L}(\mathcal{A}_i)) \quad (4.9)$$

$\mathcal{L}(\mathcal{A}_i)$ is the number of steps from category \mathcal{A}_i to the root node in the corresponding GO DAG, and \mathcal{A}_i is the GO category we want to score.

Statistical Selection

An alternative approach to selecting the most interesting GO category from the set of common superordinates is to select the *least probable* category instead of selecting the *most specific* one. This is done by initially counting the number of known proteins classified as each category, or subcategories of it. Now, we can assume the probability for a random protein α belonging to a given category \mathcal{A}_i is as described in equation 4.10.

$$P(\mathcal{A}_i) = \frac{\text{number of proteins classified as } \mathcal{A}_i}{\text{number of proteins in DAG}} \quad (4.10)$$

Therefore, we can choose the category with the smallest amount of "participants" as the most interesting category. To achieve a score of zero if two categories have only a root node in common, and higher scores the more interesting the common superordinates are, we define

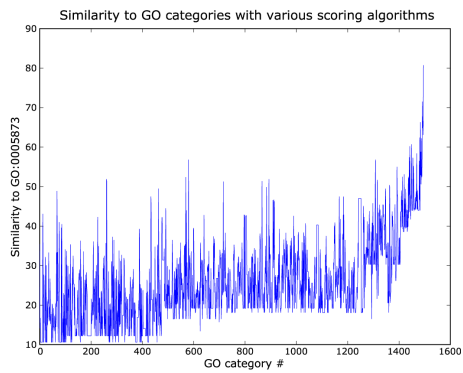
$$S_{statistics}(\mathcal{A}_i) = -\log[P(\mathcal{A}_i)] \quad (4.11)$$

We implemented this by pre-processing the proteins described and annotated in BIND. When a gene or protein is classified as \mathcal{A}_i , the GO category \mathcal{A}_i increases its count by 1, as do all the superordinates of \mathcal{A}_i . Now, the most relevant GO category in the intersection from figure 4.7d will be the one which minimizes $P(\mathcal{A}_i)$, and thus maximizes $S_{statistics}(\mathcal{A}_i)$.

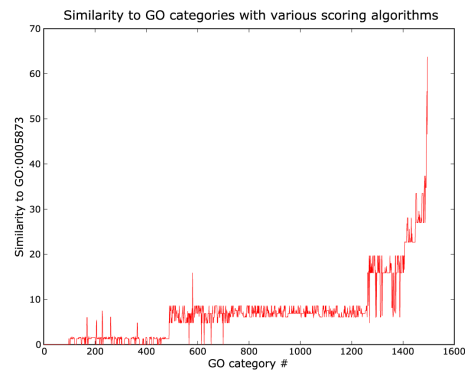
Our approach to defining the similarity between proteins by their GO annotations highly resembles methods developed by others. Resnik [1995], Lin [1998] and Jiang and Conrath [1997] describe quite similar techniques. The method described by Resnik is made for evaluating semantic similarity in natural language using WordNet [Miller 1995]. WordNet can be compared quite easily to the Gene Ontology DAGs. In WordNet, english nouns, verbs, adjectives, and adverbs are linked in an *is-a* hierarchy. For example, the words *nickel* and *dime* are both subsumed by *coin* [Resnik 1995].

For comparing the results of our two developed similarity measures, we selected a random GO node and compared it to all other GO nodes in its DAG. Using FuSSiMeg [Couto et al. 2003], we compared the same proteins with Resnik [1995], Lin [1998] and Jiang and Conrath [1997]. We plot the scores to see whether they correlate. See figure 4.8 for the plot.

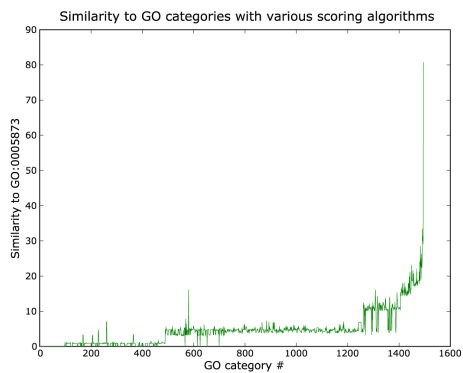
We have also calculated Pearson correlation coefficients between our two methods and the three others. The results are illustrated in table 4.1. The correlation coefficient between our two techniques is 0.733. As the correlation coefficients in the table show, our methods are correlated with the three, mostly to Resnik. We suspect the differences may arise from both the fact that we use different calculations, and a small size of annotation data on which we base our statistics. We will continue using our own scoring models, mainly because we haven't found implementations of the three, apart from the web-interface provided by FuSSiMeg.



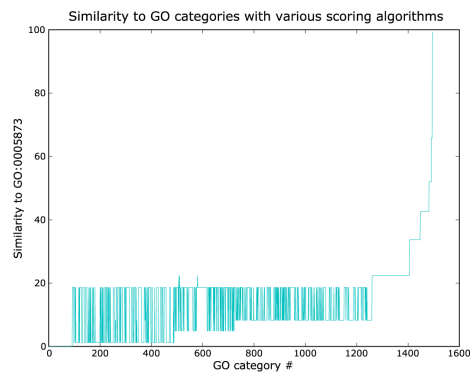
(a) Jiang Conrath scoring



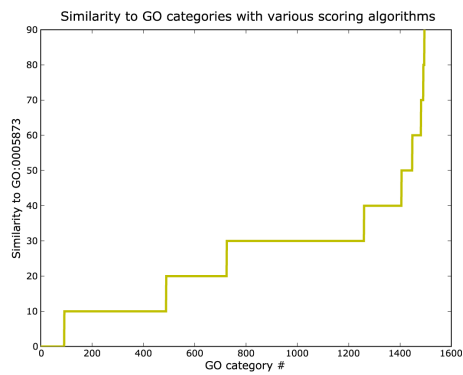
(b) Resnik scoring



(c) Lin scoring



(d) Our statistical scoring



(e) Our level scoring

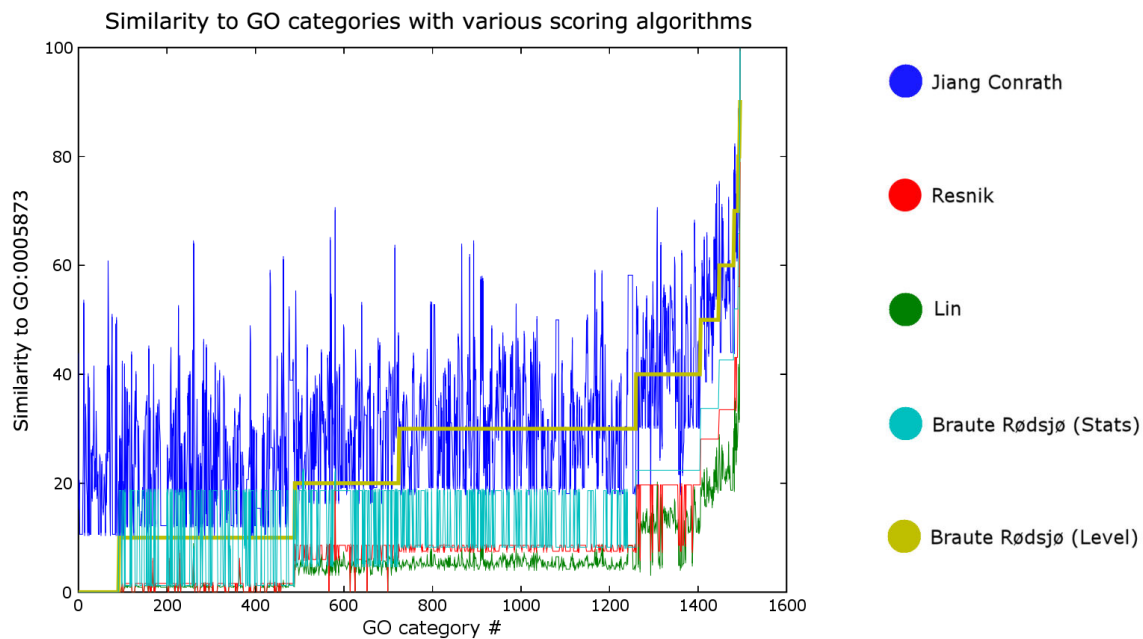


Figure 4.8: The figure shows a plot of the similarity between all nodes in the Cellular Component Gene Ontology tree and GO:0005873 (Plus-end kinesin complex). The plot compares our methods to the more known algorithms by Resnik, Lin and Jiang and Conrath. The output is sorted by our level-method, which explains the strictly increasing property of the corresponding plot, and the rather varying property of the others.

In the smaller figures above, the plots are displayed in individual diagrams for clarity.

	Braute Rødsjø: Level	Braute Rødsjø: Stats
Resnik	0.907	0.778
Lin	0.856	0.749
Jiang	0.589	0.451

Table 4.1: Correlation coefficients for level similarity

Incomparable proteins

One issue that may render us incapable of comparing neighboring proteins is the fact that the Gene Ontology hierarchy is not described in a single DAG, but as three separate DAGs (Biological Process, Cellular Component and Molecular Function). This is because these biological domains are (as the GO consortium states) "considered independent of each other." This requires us to work in parallel with three different Gene Ontology DAGs - BP, CC and MF.

When comparing two proteins, we can end up in several different situations:

1. None of the proteins are categorized in GO.
2. One protein is not categorized in GO.
3. The proteins are both categorized in GO, but they are not categorized in the same DAG (BP, CC, MF).
4. The proteins are both categorized in GO, and only share one DAG.
5. The proteins are both categorized in GO, and share more than one DAG.

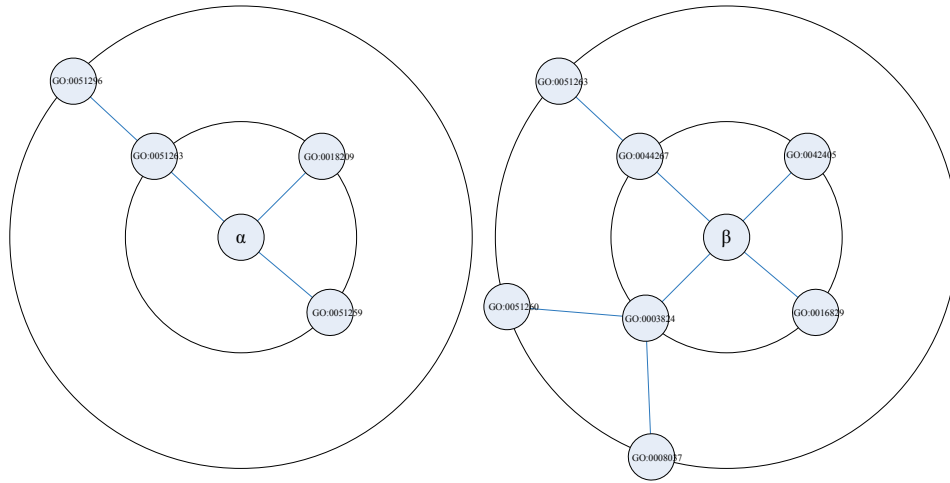
Cases 1 through 3 are not very interesting, as they leave us unable to say anything about similarity, and we must simply return a 0 score. Case 4 does not result in any special considerations, as it leaves us with only one DAG to work with. Case 5, on the other hand, gives us a problem. Which one should we use? Our solution is to run a search in all the DAGs the proteins have in common, and return the result from the DAG which yields the highest score.

4.4.4 Scoring the Neighborhoods

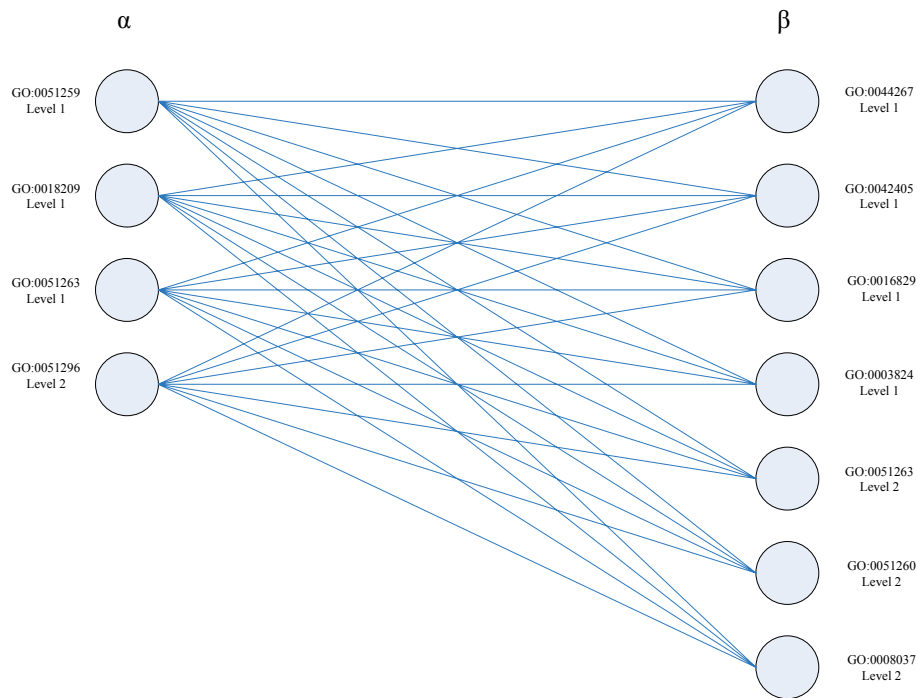
We have now built and weighted a complete, bipartite graph connecting the two neighborhoods. Our next challenge is to find the best alignment of the neighbors in the two neighborhoods. Inspired by Kelley et al. [2003], we considered using sequence alignment methods. Kelley et al. used global sequence alignment methods to match metabolic pathways in protein-protein interaction networks. However, there is a significant difference between pathways and neighborhoods - the relevance of order. While the sequence in which the proteins appear in a pathway is highly relevant, it is not in neighborhoods. We therefore found that weighted bipartite matching would be better suited for our purpose. In many ways similar to sequence alignment, bipartite matching will not care about the order in which the data appears.

Figure 4.9 explains our progress so far. All the edges in the bipartite graph are now weighted, and we only need to connect pairs of proteins from each side of the bipartite graph. The result we want to achieve is that all the proteins from the *smallest* neighborhood are connected to one, and only one, protein from the largest neighborhood. This means that some proteins from the largest neighborhood will not be a part of the matching, and will thus not contribute to the final scoring of the neighborhoods.

As we learned from the first perspective we discussed in section 4.2, information content of neighbors decreases as the distance from the bases increases. If we use the same technique as in section 4.2, we



(a) The original neighborhood of two proteins α and β .



(b) The neighborhoods connected and represented as a complete bipartite graph

Figure 4.9: Protein neighborhoods converted to a bipartite graph. This example is somewhat simplified, as many proteins are classified as more than one GO category. The principle is still the same.

would have to multiply the weights in the bipartite graph by a *distance factor* before matching the graph. This distance factor should decrease as the difference between the level of the two vertices increases. Also, it should decrease when their absolute level increases. We choose the smallest of the two levels for this second penalty. In effect, this means that two proteins with levels p and q , respectively, should be penalized by some factor of $|p - q|$ and $\min(p, q)$. We suggest the following multiplication factor for weights:

$$\text{penalty}(p, q) = \left(\frac{1}{2}\right)^{|p-q|+\min(p,q)-1} \quad (4.12)$$

For instance, the edge between two proteins both at level 1, would be multiplied by $\left(\frac{1}{2}\right)^{|1-1|+1-1} = \left(\frac{1}{2}\right)^0 = 1$. Logical, as two neighbors close to their respective bases are likely to contain a lot of information. Another example would be two proteins at levels 1 and 2. Their score would be multiplied by $\frac{1}{2}$, effectively penalizing the edge for connecting two proteins at different levels. The third example is two proteins at levels 2 and 3. These will be both penalized for being at different levels, and for being generally farther away from the bases. Their multiplication factor will be 0.25. The factor $\frac{1}{2}$ is chosen with no biological foundation, and may be adjusted to increase or decrease the amount of falloff.

We now have a bipartite graph, connected with edges weighted by an algorithm that is based on GO categories, and multiplied with factors dependent on the levels of the proteins. We define the data we now have as an **assignment problem**, and calculate the maximum possible score obtainable by solving the assignment problem for maximum cost. This score finally represents the score of the similarity between the neighborhoods.

Interpreting the score

How to interpret the data coming from the matching of the bipartite graph is an important issue we need to address. The scores returned by the algorithm will have no direct semantic meaning. So, there is no way of telling whether a given neighborhood score is a good one indicating high probability of functional similarity between two proteins. A solution could be to provide a list of proteins with well-known functions, and display the similarities of their neighborhoods. We have not been able to get this kind of data, though we still think this may be a good approach (See "Future work" on page 67 for more information). Still, our method is not useless. As mentioned in section 3.3, we anticipate wide searches throughout entire genomes to be performed. This kind of searching will in any case return a large number of similarity scores, and the user may refer to the scores as relative to each other, indicating how interesting the various hits are compared to each other. This can for instance be done in a search for the k -nearest neighborhoods, where the scores themselves are not very interesting, but the order of the output is. This way, one does not have to be too much concerned about what a score of 18.5 really means biologically.

This concludes the presentation of the algorithms. We have put a lot of effort into implementing this technique, and the following chapters will concentrate on this, and on results available from this implementation. A further discussion on the quality of this technique will be presented in chapters 6 (results) and 7 (discussion).

Chapter 5

Implementation

We have implemented a fully working system for protein function prediction based on protein–protein interaction networks. The application implements our methods from section 4.4.

Our program is based on a client-server model. We were looking for a way to easily split the development into two separate parts, as we are two developers collaborating. This was meant to speed up the development process in order to make a prototype available early. Developing the parts separately would hopefully lead to a higher productivity and less administration overhead. It would also give us the power to use the techniques and language skills each of us was most comfortable with. It would be an effective division of the project into two smaller parts which could for the most part be developed independently of each other. Based on these thoughts and a run-through of personal skills, we decided that the user-interface would be implemented in Java and that the server would be implemented in Python.

For later use, the client-server model can be dropped in favor of a more monolithic application. This would reduce all network overhead, which at some times has been a problem for us.

5.1 Overview

What we ended up with was a program where the division between logic and user interface was almost where we wanted it to be. In an ideal world, the client would not know anything about BIND or GO - and would only be a general graph-browser. Unfortunately, time did not permit this. The client has knowledge of the interactions in BIND and the GO categories each protein is classified as, but it does not use them for anything apart from specifying the search. For a long time, the client did some of the work needed to compare the neighborhoods of proteins, something we knew was not very elegant. What we wanted to do, was to move as much of the logic from the client to the server as possible. Considering the amount of time we had left at this point, we had to make a choice – either keeping some of the logic in the client, or moving it to the server. If we chose to move it to the server, we needed a working graph library. As noted in section 7.2, this is not very hard, but it would nevertheless take some time. We therefore made a decision to try to move the logic, and use one of the packages we had earlier looked when considering how to represent the

DAGs of GO. A quick look at the two candidates (NetworkX and pygraph), left NetworkX as the winner, as it seemed to have the best documentation. NetworkX performed well, and we were able to move almost all of the logic from the client to the server in a short amount of time. We consider this to be a good trade-off, even though it introduced the dependency to NetworkX.

When developing the application in two parts like this, we needed something to glue them together. We both had positive experiences using **XML-RPC** [Winer 1999]. We had some minor problems with different data-types in Python and Java, but all in all the experience has been a good one.

An schematic overview of how the client, server and input data work together and how data flows can be seen in figure 5.1.

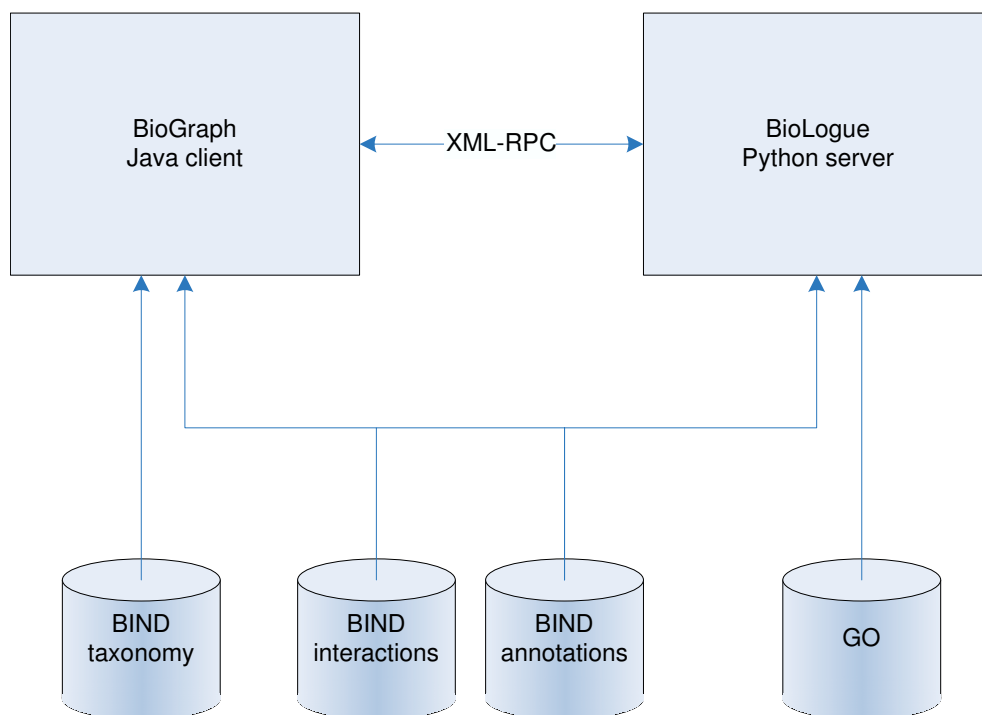


Figure 5.1: Client-server model: The schematic shows the connections between the data sources and the two main modules: BioGraph, the Java client GUI, and BioLogue, the Python RPC server.

5.2 BioLogue - The RPC server

The most important modules to implement in our project were without a doubt the algorithms we present in chapter 4. The two first scenarios, described in sections 4.2 and 4.3 are meant as an introduction to the problem area, and these are not implemented. We have focused on implementing the more interesting methods described in section 4.4. We will not go into detail regarding these, since we feel that they are described thoroughly in the earlier chapters. What we will describe are the supporting cast of our algorithms. These include the tool for browsing the protein-protein interactions and specifying searches, the tool for parsing the GO data (DAG functionality) and the matching algorithms used in solving the assignment problem. The implementation of these will be

presented here.

5.2.1 Directed Acyclic Graph Library for GO

When starting the project, we soon realized that we needed some kind of graph library to represent the GO-tree. Initially we were under the impression that GO was represented as a tree, but after looking more closely at it we found it to be a DAG. The quest then became to find a package for Python with support for DAGs. We looked at both Pygraph [Istvan] and NetworkX [Bold et al.]. Both packages had support for directed graphs, and could probably have been made to work in our system with some modifications and additions. Still we chose to implement our own library. The reasons for this were as follows:

1. Learning how a graph-library could be implemented
2. Adding functionality into own code is easier than in existing code
3. Offering the possibility to optimize code for our problem domain
4. Having a DAG library one understands the ins and outs of could be very useful for later

The graph-library was inspired by Guido van Rossum's essay "Implementing Graphs" [van Rossum]. The essay only talks about undirected graphs, but the example is easily changed to work for a DAG, being directed. The DAG is implemented as a hash table containing other hash tables. At first our implementation read the structure of the GO-tree from a flat text-file downloaded from the GO-website [Ashburner et al. 2000]. Basic DAG functionality such as building, traversing and searching was added, and the three DAGs making up GO were populated using data from file. Next, we implemented an algorithm comparing GO nodes based on the height of the lowest common ancestor in the DAG (see figure 4.7). The first step in optimization was to store the DAGs on disk, so we did not need to populate them for each run of the program. This made a significant difference in run-times, but still there was probably a lot to finetune. The solution was to run a comparison through Python's profile module [Roskind] to see what it could tell us. See table 5.1 for the initial results, showing the most time consuming methods. Note that since the profiler does not run alongside **Psyco** [Rigo 2004], the run-times are used for comparison with later tests, not absolute measuring. For absolute run-times, see later in this chapter.

number of calls	total time	time per call	cumulative time	function
1	0.000	0.000	0.023	compareNodes
1	0.000	0.000	0.019	getCommonAncestors
488/2	0.008	0.000	0.019	getAncestors
372	0.006	0.000	0.009	getParents

Table 5.1: The most time-consuming methods in the first run.

While scoring two random nodes in the BP DAG, the profiler gave us a hint that the method "getAncestors" was being called a lot, and taking up a lot of time. The slash in the first column in

row three of the table denotes that the method is recursive, and that two initial calls have resulted in 488 recursive calls. At this point, "getAncestors" traversed the tree upwards from a node, and returned everything it found. Since the GO DAGs do not need to be modified while the program is running, this information could be pre-calculated, and stored in a dictionary for future use. A dictionary-lookup should be a lot faster than traversing every node from the search-nodes and up. So the DAG-library was changed to store all ancestors for all nodes in a dictionary. The program was then run again with the same nodes, resulting in the profile we see in table 5.2.

number of calls	total time	time per call	cumulative time	function
1	0.000	0.000	0.004	compareNodes
1	0.000	0.000	0.003	makeSearchDag
26	0.001	0.000	0.002	getChildren
263	0.001	0.000	0.001	append
.
.
2	0.000	0.000	0.000	getAncestors

Table 5.2: The 4 most time-consuming methods in the second run plus the time of getAcestors

As we can see, we have now reduced the number of calls to getAncestors to only 2, and thus the run-time has been reduced. At this point we did not do any exact measuring of run-time apart from noting that the 488 recursive calls had been reduced to two lookups in a hash-table. The next step was to examine the reason for the 263 list-appends (from table 5.2), which turned out to be completely pointless and could be removed.

A relatively easy and effective way to optimize Python programs is to use Psyco. As Psyco is not the subject of this thesis we will just say that it is a Just-In-Time compiler, and agree with David Mertz when he says: *"Explaining Psyco is relatively difficult, but using Psyco is far easier"* [Mertz 2002]. We wanted to see what Psyco could do for our algorithm. At the same time, we wanted to see how fast the new set-class in Python 2.4 (implemented in C) would be compared to the old set-class (implemented in Python). To get a feel for how big an improvement we got from pre-calculating all ancestors in the trees, we also measured run-time with and without the ancestors pre-calculated and stored in a hash table.

The test was run on two randomly chosen nodes in the CC DAG and two in the BP DAG. The reason for using two different DAGs, was that we wanted to see if the different tests would affect DAGs of different sizes in different ways. CC has around 1500 nodes and BP has around 10000 nodes.

To examine the two different implementations of Set, we can look at line 1 vs. line 5 in both tables 5.3 and 5.4. The run-times seem almost identical, so for our case, the C-implementation did not speed things up considerably. For a while we attributed this to the fact that Psyco was speeding up the Python-implemented Set, but comparing lines 2 and 6 in the same tables showed pretty much the same thing. At least for this case, we can conclude that it makes little difference which set-implementation we choose. Since the difference is so small, we will only use the runs with the C-implementation for the rest of this section.

AncestorsInDict	Psyco	set/Set	Time
Yes	No	set	0.000437
Yes	Yes	set	0.000222
No	No	set	0.000623
No	Yes	set	0.000282
Yes	No	Set	0.000430
Yes	Yes	Set	0.000223
No	No	Set	0.000700
No	Yes	Set	0.000298

Table 5.3: Run-times for comparing two nodes in the CC DAG. *AncestorsInDict* determines whether are precalculated for each node. *Psyco* tells whether Psyco is enabled. *Set* means we have used the Python implementation of the datatype set, while *set* is the C-implementation. *Time* is measured in seconds based on an average of 1,000 runs on an Intel P4 3Ghz with 512MB RAM running WinXP and Python 2.4

AncestorsInDict	Psyco	set/Set	Time
Yes	No	set	0.000476
Yes	Yes	set	0.000243
No	No	set	0.002730
No	Yes	set	0.001157
Yes	No	Set	0.000463
Yes	Yes	Set	0.000256
No	No	Set	0.003292
No	Yes	Set	0.001211

Table 5.4: Run-times for comparing two nodes in the BP DAG

When looking at the difference between having the ancestors of each node pre-calculated and traversing the DAG for each method call we expected to see a big difference. Comparing lines 1 and 3 (or 2 and 4) in table 5.3 we notice a slight decrease in run-time when using the pre-calculated ancestors. Looking at the same lines in table 5.4 shows us that traversing the DAG for every call to `getAncestors` scales horribly, whereas the pre-calculated approach hardly notices the fact that BP is around six and half times bigger than CC. Pre-calculated wins by a mile.

Examining lines 1 and 2 in both tables show us that with two lines of code, `Psyco` cut the run-time for both DAGs in half. A rather nice speed-up considering it only involves two lines of code.

A further reduction of the number of unnecessary method-calls and some nit-picking reduced the time of the fastest call (line 2 in table 5.4) to 0.000100. Considering that fine-tuning this algorithm is not the main point of our thesis, we were satisfied with this run-time.

All the tests were done with the level-based comparison of GO-categories (see section 4.4.3). The reason for this was that the statistical selection (see section 4.4.3) was not implemented at this time. Since the statistical-selection also uses pre-calculated values to measure similarity, the run-times should not be very different.

5.2.2 Weighted Bipartite Matching

Our approach to comparing two neighborhoods in section 4.4.4 was to view it as the assignment problem. When searching for a solution to the problem, we first came upon the Hungarian method [Kuhn 1955; Fukuda and Matsui 1992]. We implemented the method in Java, which worked perfectly, although it was much too slow for our needs. Our next attempt was to try the Branch and Bound approach to solving the problem. This was also implemented in Java, and turned out to be even slower than the Hungarian method. We were sure that there had to be a faster way of solving such a classical problem. After some searching, we found a promising article describing *An efficient cost scaling algorithm for the assignment problem* [Goldberg and Kennedy 1995]. Reading the article, we soon figured out that implementing it ourselves would probably take more time than we had. After some more searching, we found a working implementation in C [Goldberg 2002]. This implementation is called CSA (Cost Scaling Algorithm). It took a little tinkering for it to run on our system, but when it did, it outperformed the two previous methods by orders of magnitude.

CSA came to our attention rather late, after we had implemented both the Hungarian method and the branch and bound-approach. Luckily, CSA turned out to be a major lifesaver for our project. As we have described, the Hungarian Method and the Branch and Bound-approach are hopelessly slow. Several seconds for comparing two neighborhoods with less than 20 neighbors would simply not be sufficient for our needs.

Figure 5.2 shows running times for our implementation of the Hungarian method and CSA given several random complete bipartite graphs with varying sizes. As we can see, CSA outperforms our implementation of the Hungarian method by orders of magnitude.

When incorporating CSA into our system, we made some minor changes to its **make-file** in order to make it compile on Windows. The C code was compiled using **GCC** on both Windows and Linux, using `Cygwin` [Faylor and Vinschen] to make `GCC` accessible on Windows. The downside of using CSA is that it is implemented in C, and the rest of our project uses Python and Java. To integrate

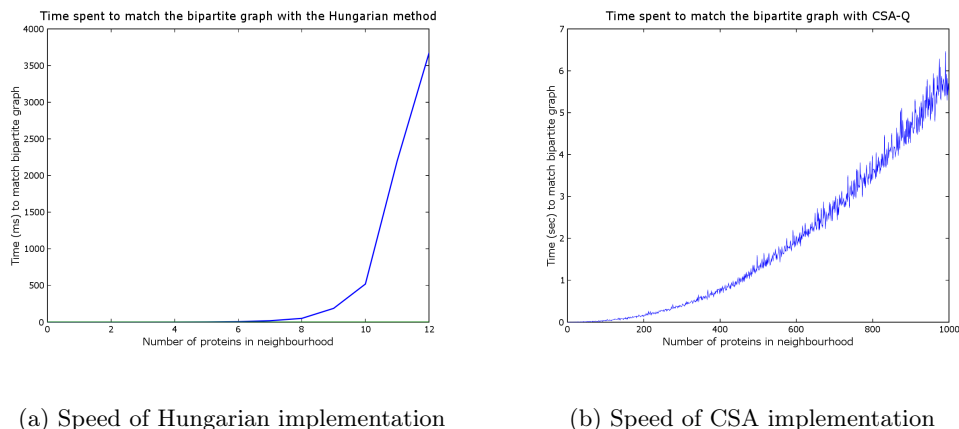


Figure 5.2: Implementations of weighted bipartite matching: The graph shows the time consumed to find the optimal matching for different values of n (Number of neighbors included in the search) with the Hungarian method and CSA

CSA, we considered wrapping it into a Python module using tools such as SWIG [Beazley] or Pyrex [Ewing]. We did not end up doing this, mostly due to lack of time and lack of experience using C. As far as we can tell, wrapping CSA would take a considerable amount of time. This was mainly because its coding style made it unsuitable for conversion into a library. To use CSA, we came up with another solution. CSA only accepts well-formed problems in the DIMACS-format [DIMACS]. What we did first was to convert our bipartite graph into the DIMACS-format and store it as a textfile. We then opened a pipe to the command line, and ran CSA through it, redirecting the textfile to standard input and capturing the output through the same pipe. The output is then parsed to extract the result of the CSA run. This is all done using libraries built into Python (`os.popen`).

5.3 BioGraph - The Interaction Network Explorer

To explore the large amounts of data available from the BIND database, we implemented a graphical user interface we call *BioGraph*. This reads data downloadable from the BIND website [Bader et al. 2003]. BioGraph is a Java application and is mainly a front-end to the various modules we have implemented. We will therefore not go into the same levels of detail when describing its implementation. What is important about it, is that it is a very flexible application meant to be expanded by others. This is done by defining the key functionality as plug-ins. The application currently has 4 plug-ins. They perform simple tasks such as looking up a specific GI from the data source to more complex tasks of comparing neighborhoods with help from the Python server. Making other plug-ins, or changing the existing ones, is an easy task. It simply involves making a class that extends the `plugins.Plugin` Java class, and place the compiled version in `plugins/install`. The class is described in figure 5.3.

As figure 5.3 shows, there are 5 methods in the abstract class *Plugin*, all are quite self-explanatory: The method `run()` is invoked by the system when the user presses the button for the plug-in, and

```

/**
 * Created on Nov 28, 2004
 *
 */
package plugins;

import java.awt.event.WindowListener;

/**
 * @author petterbr
 *
 * TODO To change the template for this generated type comment go to
 * Window - Preferences - Java - Code Style - Code Templates
 */
public abstract class Plugin extends javax.swing.JDialog implements WindowListener {
    public Plugin(String name) {
        super();
        setResizable(false);
        setTitle(name);
        addWindowListener(this);
        setDefaultCloseOperation(Plugin.HIDE_ON_CLOSE);
        setLocationRelativeTo(null);
    }
    /**
     *
     * @return String giving the user a clue about what the plug-in does.
     */
    public abstract String getToolTipText();
    /**
     *
     * @param graph - Send by the main BioGraph application, containing all
     * nodes currently available in memory.
     *
     * For accessing proteins in persistent storage, use
     BioGraph.getGraphFactory().getInstance()
     * (by Singleton pattern)
     */
    public abstract void run(Graph graph);
    /**
     *
     * @return name of the plug-in, to be displayed on plug-in button in BioGraph
     */
    public abstract String getPluginName();
    /**
     *
     * @return icon image, to be displayed on plug-in button in BioGraph
     */
    public abstract String getImage();
}

```

Figure 5.3: Plug-in Java interface

this typically opens a new window with settings for the plug-in.

Persistence is also handled by the plug-in approach. We have made two plug-ins for getting input from BIND and GO – one that reads from the files directly available from the sources on the Internet, and one that uses an SQL database as a buffer. The first plug-in loads all available data into memory as startup, and is therefore quite memory-intensive, but gives a better experience for the user once the application has started. New data source providers can also be created, by implementing the `persistence.DataSourceProvider` java interface and placing the compiled class in the `persistence` folder.

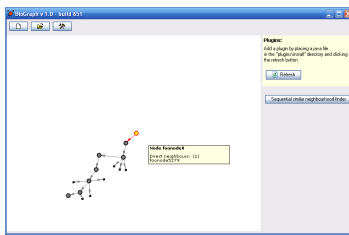
BioGraph is equipped with a configuration file called *biograph.properties*, in which the user can change settings such as which data source provider to use, how to connect to the Python RPC server, where to obtain datafiles from, etc.

Figure 5.5 shows the internal structure of the most important parts of BioGraph.

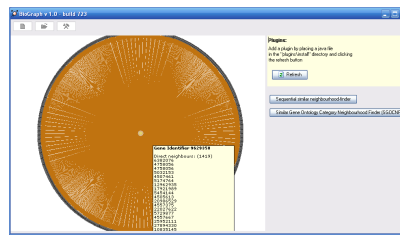
5.4 Licensing

At the moment, our application (the Python server and the Java client) is licensed under GPL [Free Software Foundation 1991a]. This is because we have compiled the CSA-algorithm to be used on Windows using Cygwin [Faylor and Vinschen]. If a need arises to do something that the GPL does not allow, the CSA-algorithm could be either re-implemented or just compiled using another set of tools, e.g., MinGW [Peters et al.] (which is public domain). Our choice to use Cygwin was one of comfort. CSA compiled with relative ease under Cygwin, and our first attempts to use MinGW were not met with success. Even so, we think compiling without the links to `cygwin1.dll` should be relatively easy if the need should arise. The sources for CSA are included in our program. They are almost identical to the version we downloaded, except for a slight change in the make-file to make it compile using `gcc` in Cygwin. Should one choose to remove the dependencies to Cygwin, the application would no longer be licenced under GPL, and could be considered public domain, except for NetworkX which is LGPL (see below). All we ask is props to us if you find the program useful.

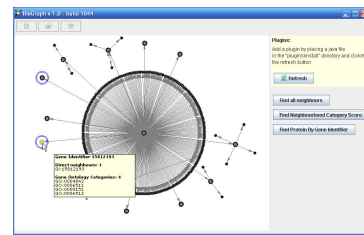
NetworkX is licenced under LGPL [Free Software Foundation 1991b]. In short terms, this means that it in it self has the same license as GPL [Free Software Foundation 1991a], but it does not infect the rest of the application with the GPL. Should you wish to redsitribute our application, you only need to include the sources to NetworkX. As noted in 7.2, this dependancy can be removed with out too much work should it become necceary.



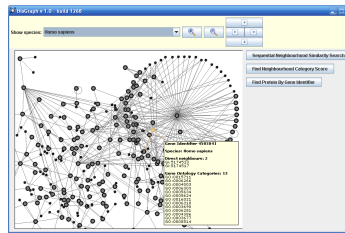
(a) Build 651



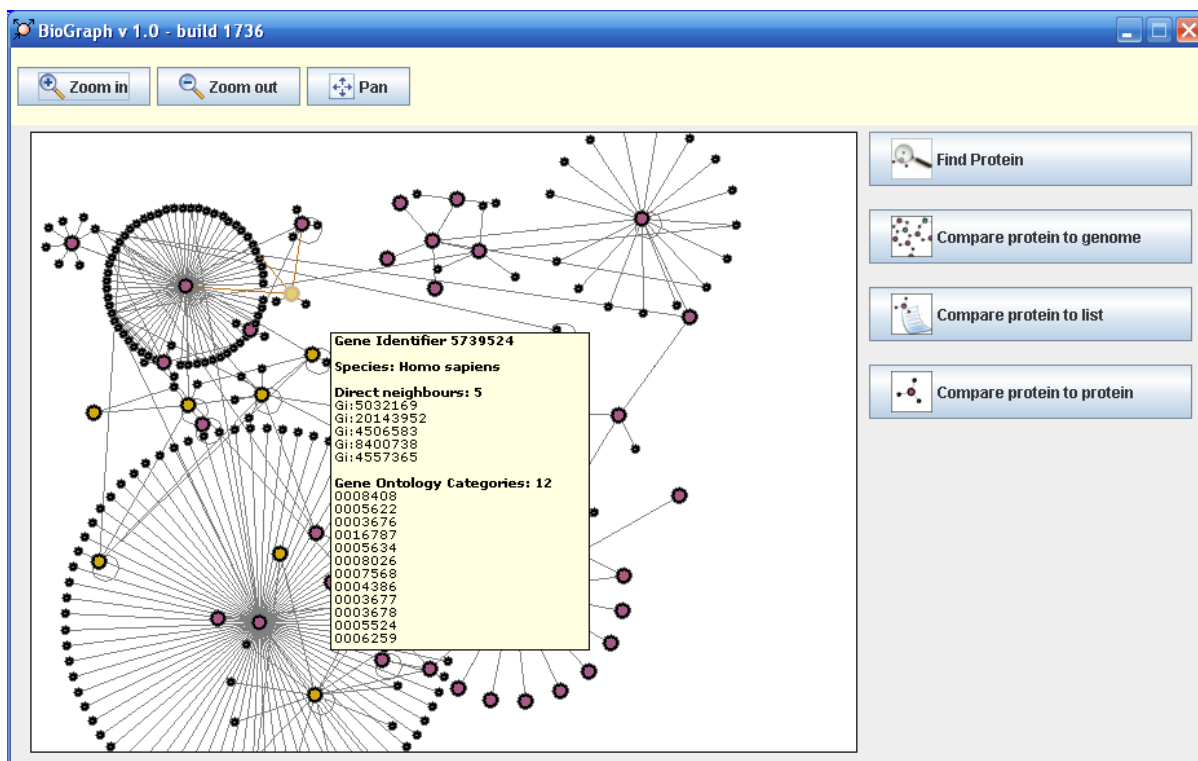
(b) Build 723



(c) Build 1044



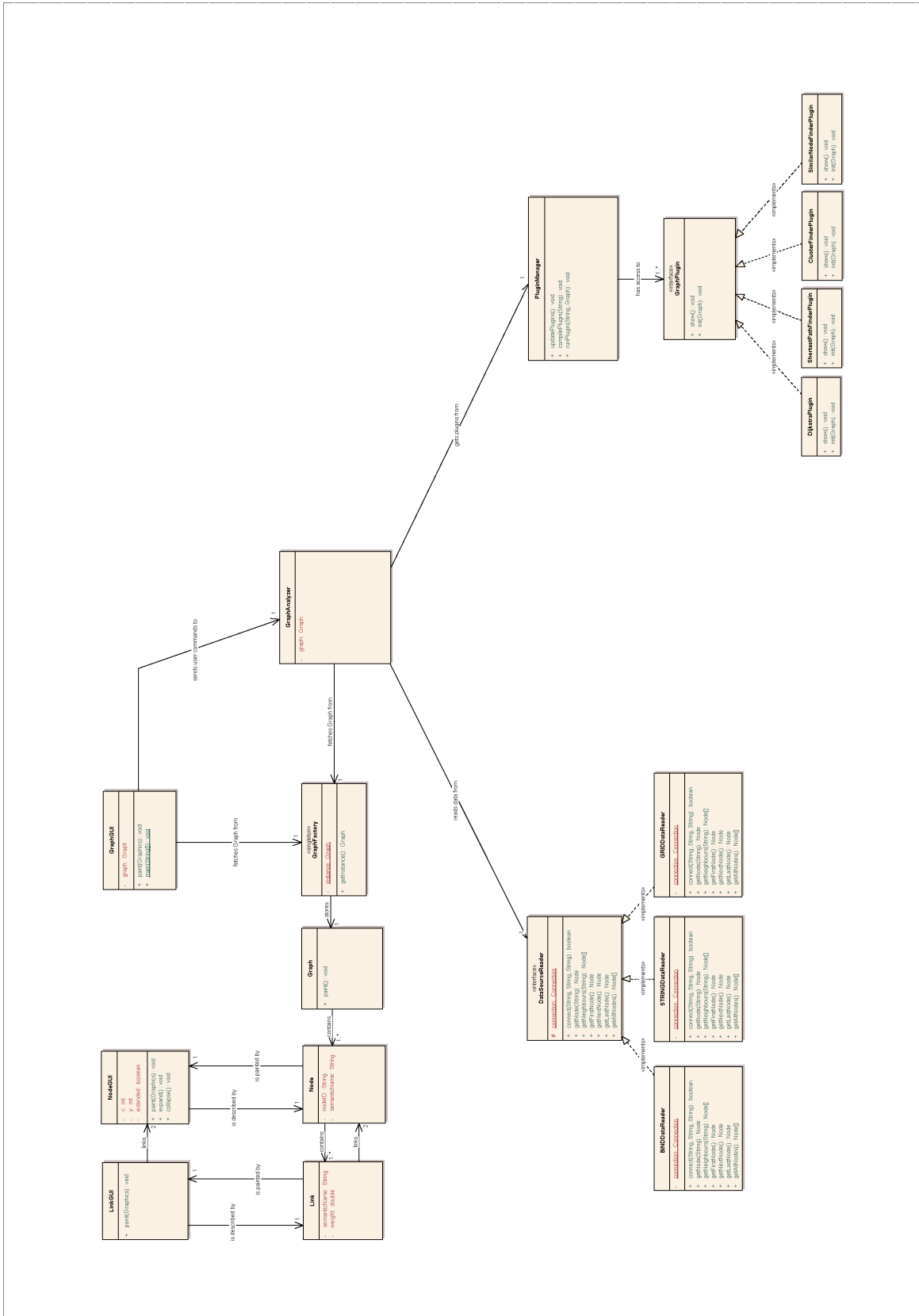
(d) Build 1268



(e) Build 1736

Figure 5.4: Various builds of BioGraph. Protein interaction data is dynamically read from the BIND database, and proteins are annotated by Gene Ontology.

Figure 5.5: BioGraph system design



Chapter 6

Results

Don't tell people how to do things. Tell them what to do and let them surprise you with their results.

– George S. Patton

This chapter presents results coming from our algorithms in chapter 4, and our implementations from chapter 5. This chapter includes testing of output data from the algorithms and programs. The tests are performed for two reasons: (a) To find out whether our methods return relevant, interesting results, and (b) to find out whether or not they do this at reasonable speeds.

6.1 Information Content

Chapter 4 on algorithms ended with us not knowing how to give semantic meaning to the output from the algorithms: There is no theoretical way to describe the quality of the output. Therefore, we try an empirical approach. We have selected a number of interesting proteins with a sufficient number of interacting neighbors in BIND. These proteins were compared and scored, resulting in the matrix shown in the following tables. We used both level- and statistic-based scoring schemes as described in sections 4.4.3 and 4.4.3, and increasing number of neighborhood levels to include in the calculation.

In an ideal world, we would compare these results to opinions from a biologist to see whether there is a correlation. We haven't been able to do this, but instead, we used the direct protein-protein scoring algorithm (originally used as part of the neighborhood similarity measure) with the same set of proteins. Thus, we produce two sets of results: (a) Scores based on neighborhood information only, and (b) scores based directly on the proteins themselves. This will be used as a test case indicating the quality of our neighborhood-based scoring algorithms, and we assume that the correlation between these results will indicate how well our algorithms actually perform.

We will also use this experiment to determine what neighborhood-size we recommend using for comparing neighborhoods. We will do this by running the same tests with increasing numbers of

neighborhood levels. We will stop this when the correlation between the neighborhood-results and the direct results decrease.

Tables 6.1 and 6.2 are the results of the *direct* scores between the proteins. This has not used any neighborhood information, only GO similarities.

GI / GI	4557757	5453830	7705592	4504191	4557761	4506017	4758952	4757930	14327896
4557757	3,02								
5453830	2,25	3,40							
7705592	0,99	1,67	3,09						
4504191	3,02	2,25	0,74	3,02					
4557761	3,02	2,25	0,99	3,02	3,61				
4506017	2,25	2,25	1,20	2,25	2,25	3,83			
4758952	0,96	0,96	0,96	0,45	0,96	3,13	4,01		
4757930	1,77	0,99	0,99	1,80	1,77	1,98	0,96	2,93	
14327896	1,77	0,99	0,99	1,80	1,77	1,98	0,96	2,18	2,82

Table 6.1: Direct scores between selected proteins (statistical scoring scheme).

GI / GI	4557757	5453830	7705592	4504191	4557761	4506017	4758952	4757930	14327896
4557757	3.32								
5453830	3.00	3.32							
7705592	2.58	3.00	3.17						
4504191	3.32	3.00	2.58	3.32					
4557761	3.32	3.00	2.58	3.32	3.32				
4506017	3.00	3.00	2.81	3.00	3.00	3.81			
4758952	2.00	2.00	2.00	2.00	2.00	3.00	3.17		
4757930	2.58	2.58	2.58	2.58	2.58	2.81	1.58	2.81	
14327896	2.58	2.58	2.58	2.58	2.58	2.81	1.58	2.81	2.81

Table 6.2: Direct scores between selected proteins (level scoring scheme).

The tables 6.3 and 6.4 show neighborhood scores with *one* neighborhood level.

To see whether we get better scores with a higher number of neighborhood levels, we performed the same tests for $N^2(\cdot)$ searches. The results are shown in tables 6.5 and 6.6.

We have calculated both Spearman and Pearson correlation coefficients between the results from the neighborhood-scores and the direct scores. Table 6.7 shows the results. As we can see from the table, the most interesting connection is between direct scoring and statistical neighborhood scoring with *one* level. We therefore calculated p -values using randomization (see section 2.2) for these coefficients, and both values have a p -value of $\ll 0.0001$, found by the randomization technique run with 10 million iterations. In these, no random permutations resulted in higher coefficients for Spearman correlations, and 15 for Pearson.

Figure 6.1 shows a visual representation of the results from direct proteins similarity and neighborhood similarity with one level as input.

GI / GI	4557757	5453830	7705592	4504191	4557761	4506017	4758952	4757930	14327896
4557757	26.71								
5453830	17.25	30.4							
7705592	4.39	4.56	3.5						
4504191	10.97	10.34	4.57	14.87					
4557761	17.36	13.11	4.57	12.5	19.7				
4506017	9.1	9.96	4.88	9.91	9.08	14.11			
4758952	4.65	4.62	2.16	4.25	4.53	6.76	11.25		
4757930	9.57	13.17	4.56	9.39	8.64	9.34	4.2	14.75	
14327896	14.23	16.21	5.33	10.65	12.16	9.55	4.16	11.68	25.82

Table 6.3: neighborhood scores between selected proteins using statistical scoring, and *one* level of neighborhood as input.

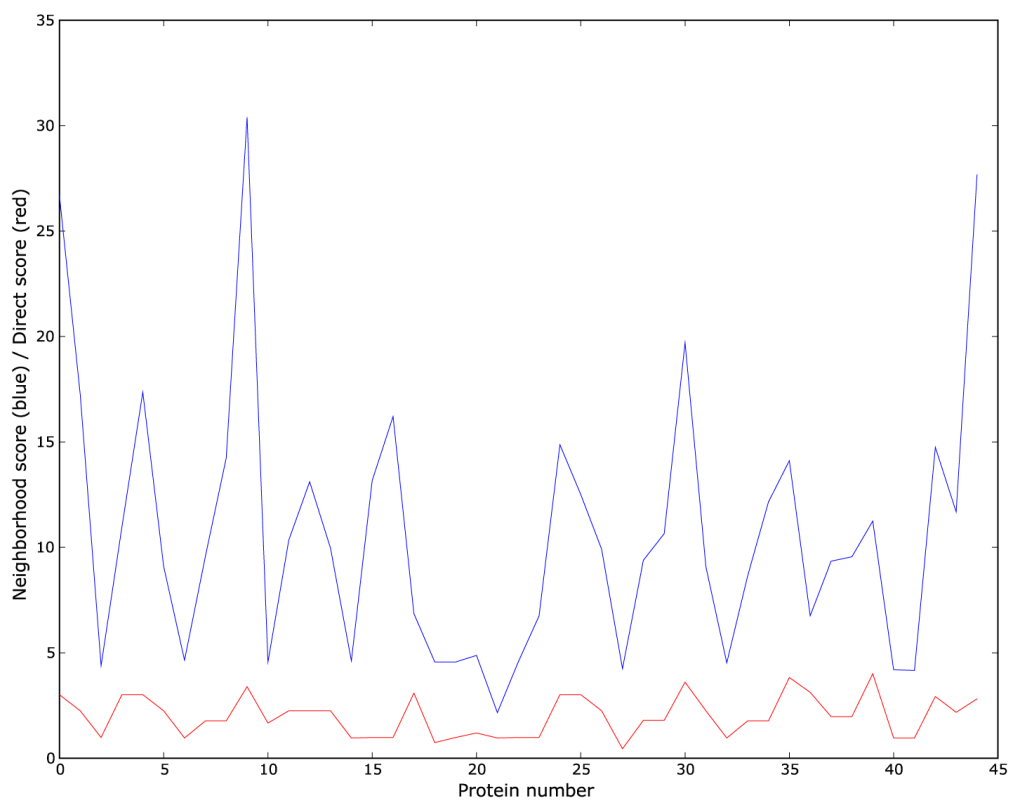
GI / GI	4557757	5453830	7705592	4504191	4557761	4506017	4758952	4757930	14327896
4557757	26.24								
5453830	29.3	28.82							
7705592	5.39	5.17	6.75						
4504191	12.81	12.17	5.61	13.55					
4557761	19.46	18	5.61	13.23	19.87				
4506017	12	12.34	5.98	12.17	12	14.07			
4758952	8.58	8.8	4.91	8.17	8.58	8.81	9		
4757930	11.49	12.49	5.75	11.56	11.39	12.49	8.17	12.81	
14327896	19.12	19.1	6.39	11.78	16.95	11.61	8.17	11.81	21.37

Table 6.4: neighborhood scores between selected proteins using level scoring, and *one* level of neighborhood as input.

GI / GI	4557757	5453830	7705592	4504191	4557761	4506017	4758952	4757930	14327896
4557757	160.89238								
5453830	97.29502	338.43719							
7705592	88.52146	90.79898	195.21995						
4504191	143.48575	169.54535	131.18252	248.8271					
4557761	142.93446	143.03194	115.78714	196.22822	200.97136				
4506017	140.75588	277.33805	180.15234	223.57689	182.45111	2771.2644			
4758952	135.48969	272.51724	179.75498	220.1326	178.55544	2639.23283	2652.16875		
4757930	120.82371	258.18381	158.07057	202.77767	166.95484	587.3588	584.63588	604.96952	
14327896	136.14799	241.45283	172.3047	220.6579	186.40512	661.89195	658.39705	582.11834	604.96952

Table 6.5: neighborhood scores between selected proteins using statistical scoring, and *two* levels of neighborhood as input.

Figure 6.1: Neighborhood score vs. direct score



GI / GI	4557757	5453830	7705592	4504191	4557761	4506017	4758952	4757930
4557757	109.31468							
5453830	95.48908	220.67284						
7705592	88.47906	104.08009	131.64507					
4504191	105.80722	153.72821	117.16254	167.69322				
4557761	106.47997	127.00276	110.63882	138.33692	139.2734			
4506017	101.8479	213.93163	130.16983	161.94288	132.84277	1747.77545		
4758952	101.64723	212.62606	130.16983	161.63685	132.6421	1660.50865	1662.13939	
4757930	99.6652	208.15175	126.15596	158.50949	130.02748	364.69452	363.98302	365.98101
14327896	102.14336	205.2935	128.00925	162.20994	134.05688	423.3535	423.15283	363.14851

Table 6.6: neighborhood scores between selected proteins using level scoring, and *two* levels of neighborhood as input.

	Pearson	Spearman
Level-scoring, 1 level	0.54	0.62
Stats-scoring, 1 level	0.64	0.71
Level-scoring, 2 levels	0.47	0.25
Stats-scoring, 2 levels	0.48	0.30

Table 6.7: Correlation coefficients

The correlation coefficients indicate that our method does indeed yield interesting results. It proves that even with no annotation data on a given protein, it can, with the right amount of protein–protein interaction data, find similarities with other proteins with a high level of confidence. It also tells us that we should not recommend using neighborhood-levels higher than 1 with the current version of the algorithms, as the correlation coefficients already are decreasing when using 2 neighborhood levels.

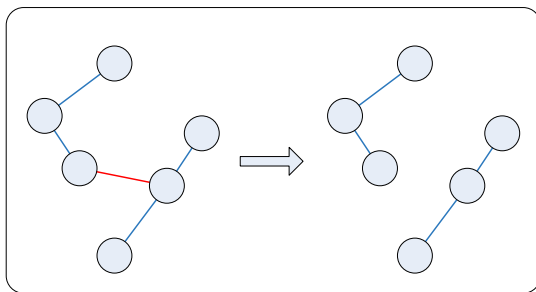
Still, these results may be slightly misleading. All proteins here have reasonably similarly sized neighborhoods, and in the real world, when comparing entire genomes, the output would probably be less correlated. A discussion on this problem is included in chapter 7.

6.2 Robustness

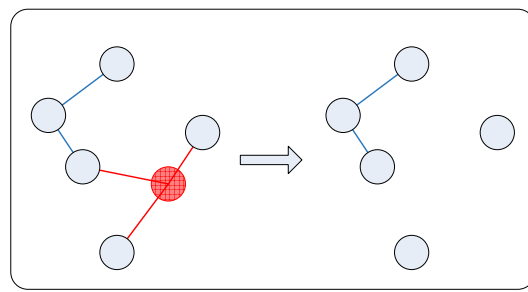
The input data for our function prediction methods – the protein–protein interaction networks and GO classification of the proteins – are prone to errors: Interactions between proteins may be false positives and/or false negatives, Gene Ontology classifications may be inaccurate, either simplified or directly misleading. To find out how stable our methods are, we generated a "mutated" genome derived from a sufficiently described genome. We iteratively applied noise to the new genome, logging the similarity between a protein and all proteins in the mutated genome for each mutation step. Mutation steps simulate errors in the data, and include the following (See figure 6.2 for examples):

1. Deletions of protein–protein interactions
2. Deletions of proteins (and incident edges)
3. Additions of protein–protein interactions
4. Simplification of protein classifications (A random GO classifier is replaced by one of the superordinates of the original classifier)

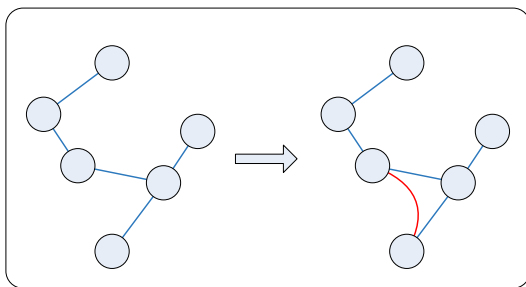
Since the mutations listed simulate errors in prior research, we do not know to which extent these errors occur. The chances of errors are dependent on the quality of the experiments previously performed on the proteins. Thus, we will simply select a likelihood of 10% for each of the listed mistakes to occur. Eventually, the mutated genome will turn into complete rubbish, but what we are interested in finding, is the limit where we can still rely on our methods for predicting protein function. We can then discuss whether this limit indicates a robust method.



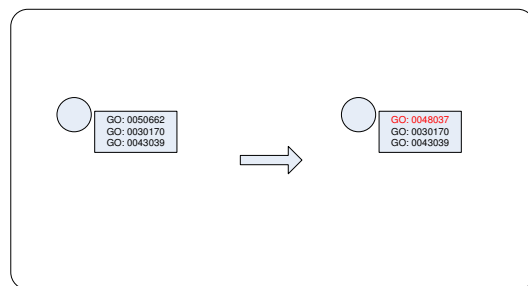
(a) Deletions of protein–protein interactions



(b) Deletions of proteins (And incident edges)



(c) Additions of protein–protein interactions



(d) Simplification of protein classifications (A random GO classifier is replaced by one of the superordinates of the original classifier)

Figure 6.2: Examples of genome mutations.

6.2.1 The Test Case

Arabidopsis thaliana is a plant, described by approximately 800 protein–protein interactions in BIND. We chose this genome because of its suiting size (not too small, nor too large), and started running the mutations. For each mutation, we compared the protein *Nuclear receptor coactivator 3* (GI 23396777 - randomly chosen from BIND) to all proteins in *Arabidopsis thaliana*. We then calculated correlation coefficients between the first output (before any mutations) for each iteration. We assume the coefficients describe well the degree of similarity between the two sets. We kept mutating the genome until both pearson and spearman correlation coefficients dropped below 0.5.

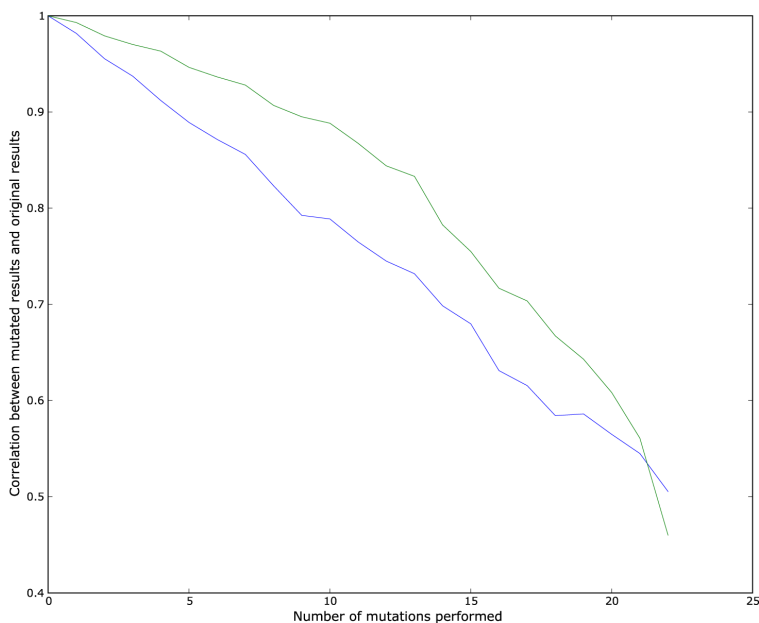


Figure 6.3: Mutation resistance

The results are shown in figure 6.3, where the blue line is spearman correlation, and the green line is pearson. We see from the plot that after 17 mutations, the correlation coefficients drop below 0.6, a correlation level we consider so low that the results are uninteresting. At this point, 40% of the interactions and proteins in the mutated genome have been removed, changed or otherwise mutated (as described in the previous section).

Considering the destructive mutations our genome has been exposed to, and the relatively slow decay of correlation, we conclude by regarding our methods to be relatively robust to erroneous changes in the genomes, an advantage if one wants to use them on less accurate data.

6.3 Performance

We have tested our algorithm to see how fast it will perform, and we have tested neighborhood-neighborhood similarity calculations with a selection of proteins with varying sized neighborhoods. The results from these measurements are shown in table 6.8. Looking back at figure 3.2 on page 13 (and the numbers it is based on), we find a mean neighborhood size (N^1) of 10 proteins. Also, genome sizes vary from hundreds of proteins to a few thousands, plus three genomes larger than 10,000 proteins. The average size of a genome is 319 proteins.

	1	5	10	20	50	100	678
1	0.05	0.05	0.06	0.06	0.08	0.13	3.68
5		0.06	0.06	0.07	0.09	0.16	3.83
10		0.06	0.07	0.10	0.18	4.12	
20				0.09	0.14	0.25	4.87
50					0.23	0.41	6.28
100						0.60	8.38
678							45.98

Table 6.8: Speed of neighborhood comparison. The table shows time (sec) needed to score the similarity between two neighborhoods with varying sizes.

We assume the user uses N^1 neighborhoods in the comparisons, as a result of our recommendations from section 6.1. Experiments have also shown that the mean size of a N^1 neighborhood is 10 proteins. We see from table 6.8 that comparing two neighborhoods where $|N^1| = 10$ will take 0,06 seconds. This means that an average total time for finding k -nearest neighborhoods in a random genome will be $0.06sec/protein \cdot 300proteins \approx 18sec$. However, if searching through the largest genome, that of *Drosophila Melanogaster*, with 35,744 proteins, the calculations will take $0.06sec/protein \cdot 35,744proteins \approx 35minutes$. Drastically higher, but still times we should be able to live with. It seems our tool may be a valuable asset to finding similar neighborhoods.

Also, the times above are calculated on the server-side, meaning traffic between the server and client is not included. Still, empirical tests have shown that network overhead is very small, and will not increase the time much.

6.4 Optimizations

As the performance results from section 6.3 shows, our application performs well under normal circumstances. However, if one should choose to include more levels of neighborhoods when searching, run-times would increase drastically. We will therefore discuss some approaches to optimizing the algorithms, and discuss whether these can speed up the performance of our application.

6.4.1 Overestimating Weights

The operation we thought would be most time-consuming – the matching of the bipartite graph – is done very fast with CSA. Profiling our application showed that the bottleneck was the weighting of the bipartite graph. So, what we tried to do was find a way to efficiently overestimate the score of a neighborhood without weighing every edge in the bipartite graph. The idea was to quickly skip neighborhoods that yield too low or otherwise uninteresting estimates, when a large k -nearest neighborhoods search is being run.

Let α and β be the two proteins of which we want to compare the neighborhoods. $|N^x(\alpha)| = m$, $|N^x(\beta)| = n$. Our goal is to find the best score any of the $m \cdot n$ edges in the bipartite graph can theoretically obtain. We let this optimal weight be called \hat{w} . The number of edges that will be kept in the matching of the graph is equal to the smallest value of m and n , so the best score the neighborhood now can achieve, will be:

$$\hat{S}(\alpha, \beta) = \hat{w} \cdot \min(m, n) \quad (6.1)$$

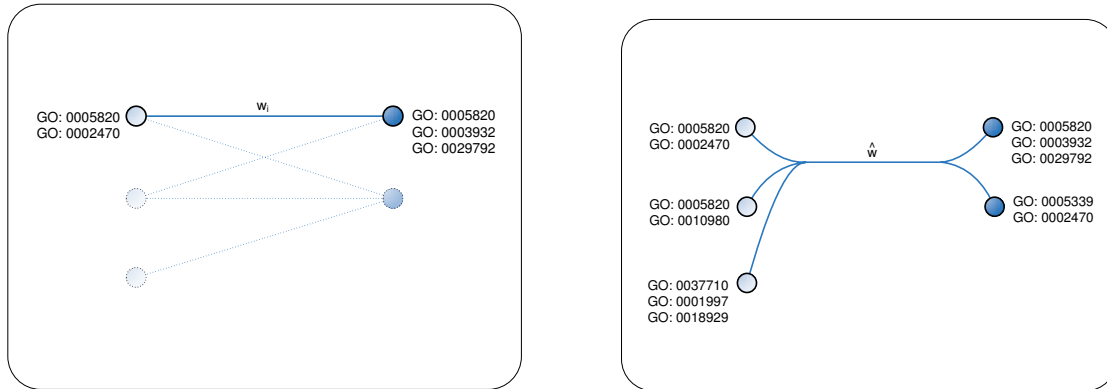
So, how can we find \hat{w} ? Normally, when calculating *one* weight w , we find the union of all of $N_i(\alpha)$'s neighbors and all of $N_i(\beta)$'s neighbors, and intersect these two sets. This is in itself equivalent to selecting the two best GO matches from the two neighbors and using these as a measure. This method is very fast, it is repeating it $m \cdot n$ times for each neighborhood that is slow. Now, the overestimate is calculated by treating all the proteins in $N(\alpha)$ as one, and all in $N(\beta)$ as one. The same weighing-algorithm is run on these two "virtual proteins", and the result: The equivalent to selecting the two best GO matches from the two *neighborhoods* (instead of two neighbors). This is indeed the best score any pair of neighbors could obtain, and we have calculated \hat{w} .

To explain the difference between calculating weights w the normal way and calculating \hat{w} , see figure 6.4. Figure 6.4a shows the normal way of calculating one of the $m \cdot n$ weights. In figure 6.4, we see that all proteins on each side are compared to each other in one step, finding \hat{w} . The figure also illustrates the calculation of $\hat{S}(\alpha, \beta) = \hat{w} \cdot \min(m, n)$. In this case, $\min(m, n) = 2$.

Now, if the user is running a k -nearest neighborhoods search over a genome \mathcal{A} , the overestimate can be calculated for each protein neighborhood being scored. If \hat{S} is smaller than k values already calculated, we can discard the neighborhood and move on to the next, without calculating its real score.

Speedup factor

The speedup factor achieved by the method just described will depend on several factors. We will estimate it based on a k -nearest neighborhood search, meaning we wish to find the k most similar neighborhoods to that of α . We let G define the size of the genome we are searching through. Further, we let \bar{n} be the mean neighborhood size. *goSim* denotes the amount of time needed to compute one weight in the bipartite graph, and *goSimEstimate*(n) the time to calculate \hat{S} . Finally, we let *csa*(n) represent the time needed to match the bipartite graph (used when calculating the real score, not the overestimate).



(a) Normal calculation of weights w_i , $0 \leq i < (m \cdot n)$

(b) Optimized evaluation of \hat{w}

Figure 6.4: Optimization by overestimation

The time needed to perform the k -nearest neighborhoods without the overestimate optimization would be as defined in equation 6.2. The explanation is pretty straight forward: To compare α 's neighborhood to all neighborhoods in the genome, we have to sequentially perform \bar{n}^2 weighings plus matching with $csa(n)$, and do this a total of G times.

$$t_{normal} = G \cdot (\bar{n}^2 \cdot goSim + csa(n)) \quad (6.2)$$

With the optimization, however, we must in addition run the overestimate calculation ($goSimEstimate(n)$) for each neighborhood comparison. This adds to the total time needed, but only in $\frac{K}{G}$ cases do we have to calculate the real score (the cases where the overestimate makes it to the $top-K$ scores). The needed time with the overestimation method is shown in equation 6.3

$$t_{optim} = G \cdot goSimEstimate(n) + \frac{K}{G} \cdot t_{normal} \quad (6.3)$$

This gives us a speedup factor of...

$$\begin{aligned} speedup &= \frac{t_{optim}}{t_{normal}} = \frac{G \cdot (\bar{n}^2 \cdot goSim + csa(n))}{G \cdot goSimEstimate(n) + \frac{K}{G} \cdot G \cdot (\bar{n}^2 \cdot goSim + csa(n))} \\ &= \frac{\bar{n}^2 \cdot goSim + csa(n)}{goSimEstimate(n) + \frac{K}{G} \cdot (\bar{n}^2 \cdot goSim + csa(n))} \end{aligned}$$

By profiling our implementations, we have found the values for $goSim$, $goSimEstimate(n)$ and $csa(n)$ under different conditions (varying values for n). The plots in figure 6.5 shows expected

speedup factors for various values of k , G and \bar{n} . As the figures show, we can expect quite good speedup factors if the algorithm works as intended.

However, we have made one simplification we must consider first: We haven't taken false positives into consideration. That is, $\frac{K}{G}$ should be higher, depending on how often the overestimate determines that a score needs to be calculated "for real", when in fact it shouldn't have been. If the overestimates are too far off, we cannot use the optimization technique at all. This is because the overestimates would trick the algorithm into thinking that a pair of neighborhoods will get a high score, making it calculate the similarity with the exact algorithm, only to find out it didn't yield that high a score anyway.

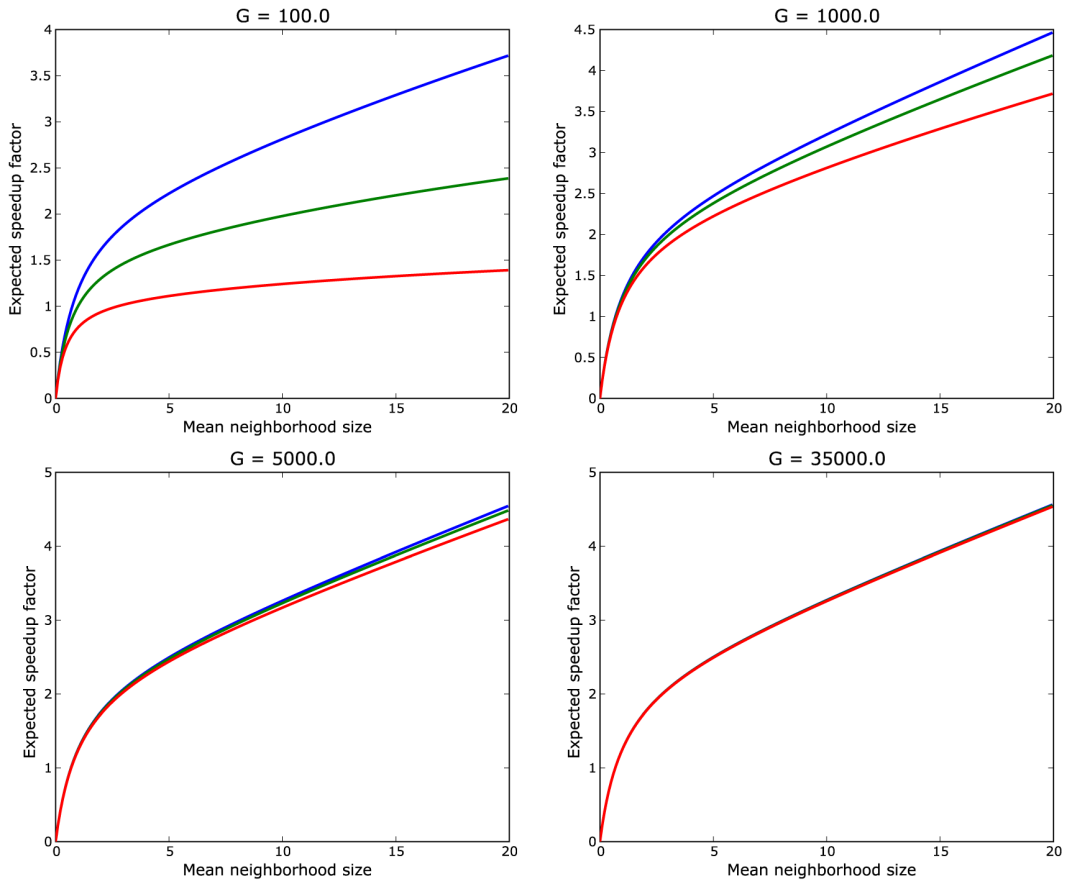


Figure 6.5: Expected theoretical speedup factors. Blue lines: $k = 5$. Green lines: $k = 20$. Red lines: $k = 50$.

We have implemented the overestimate method to find out whether the estimated scores are way off. The results, unfortunately, indicated that they are. In fact, so much that most of the neighborhoods would be estimated to be amongst the top k , while most of them in reality are not. We calculated overestimates and actual scores for 200 random neighborhoods. Then we calculated the

overestimate/real score ratios, and plotted these in a histogram. The histogram is shown in figure 6.6. As we can see, all overestimates are higher than the real scores (so it is in fact an overestimate), but unfortunately, most of them are far too high. Many overestimates are 4 times the real score, and far too many scores would be assumed to be interesting and re-calculated with the exact algorithm. The result is an increase in total run-time rather than a decrease. We cannot simply reduce the overestimates by a constant factor, as we could no longer guarantee it being an overestimate. This could lead to false negatives – faulty assumptions that a neighborhood score is too low to make it to the *top K*, and thus return incorrect results in a *k*-nearest neighborhoods search.

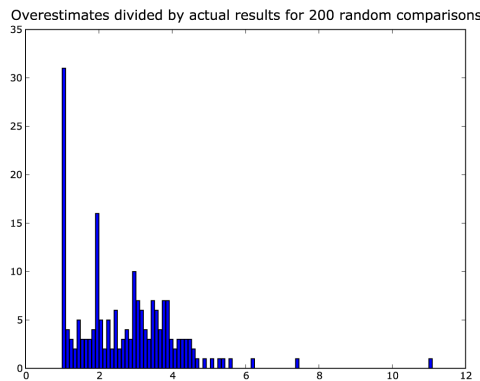


Figure 6.6: Histogram: Speedup/real score ratios

Unfortunately, our overestimation method will not help us speed up the methods at all. The only way it could help, is if we adjust the overestimates to line up better with the actual results and accept some missed results due to false negatives. We could in this case obtain some speedup as a tradeoff to some false negatives. We have not had the time to look into this matter during our thesis, and this will therefore be discussed briefly in section 7.2 on future work.

6.4.2 Memoization

Our algorithm has an advantage that we feel we cannot leave unexploited: When comparing one protein to a set of many proteins, many of the same calculations are repeated. Especially direct GO-similarity between neighbors (calculating the weights in the bipartite graph) is a task performed often, and we thought we could achieve a relevant speedup factor by memoizing results from these calculations. This is easily implemented by checking whether we have calculated this weight earlier. Temporary results are stored in a hash-structure, and retrieved if they exist.

We have not calculated a theoretical speedup factor for this approach at all, but we have run the algorithm both with and without the optimization. The results were, however, not happy reading. We could in most cases not detect any significant speedup-factor. This is probably because the calculations doing the weighing are already fast, and hash-lookups also take some time. We may have achieved some better results by denying the hash-structure to grow too large, as this may make it slightly slower. This could be done with a LRU cycling-approach, but we anticipate this will slow the memoizing down even more. We will therefore not do anything more with this idea.

Chapter 7

Discussion

People ask for criticism, but they only want praise.

– W. Somerset Maugham

The results from the previous chapter need to be analysed for us to produce a conclusion of our work. This chapter discusses the output from the tests performed, and suggests possible ways things could be made better, describing some future work we see the need for.

7.1 Discussion of Results

The correlation coefficients obtained from section 6.1 indicates that *something* is indeed working. However, one aspect of the scoring that may lead to bad results is that our scoring-algorithm accumulates weights from the bipartite graph to calculate the score for two neighborhoods. Larger neighborhoods will result in constantly higher scores. This is, to a certain extent, a positive effect – Larger neighborhoods have more information about the base than very small neighborhoods, but it leads us to the question of validity. Are we measuring the similarity of neighborhoods, or are we measuring the amount of research that has been done on a given protein? We cannot be sure if a protein is a part of few interactions because it is not very popular amongst researchers or if it is just not part of many interactions. The same goes for proteins that take part in a large number of interactions. For us to be able to tell these two cases apart, BIND would have to include some kind of score for the 'popularity' of each protein amongst researchers. BIND does currently support such a feature.

Another option would be to normalize scores by dividing scores on the size of the neighborhoods, but this would also be unjust. This way, very small, uninteresting neighborhoods could compete against large, meaningful neighborhoods. We therefore do not recommend doing this.

Another issue we feel the need to discuss is what to do with neighborhoods that have different sizes (at same depths). Given two neighborhoods $N^1(\alpha)$ (with three neighbors) and $N^1(\beta)$ (with four neighbors), we could after the weighted bipartite matching end up with a graph like figure 7.1.

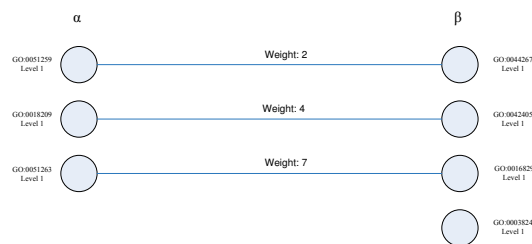


Figure 7.1: Two proteins with different size neighborhoods

We want to give a higher score to equal-size neighborhoods than to unequal size neighborhoods e.g. figure 7.1. So the question of penalizing different-size neighborhoods came up. After lengthy discussion we decided to ignore the problem, since a $4 \cdot 4$ neighborhood would have one more link than a $3 \cdot 4$ neighborhood, and this gives an implicit penalty as the $4 \cdot 4$ one would get a higher score than the 3×4 one.

Even after taking the problems of our method into account, we believe that it to a certain extent shows that neighborhood similarity can be used to predict protein-functions, and that our tool can do so within a reasonable amount of time.

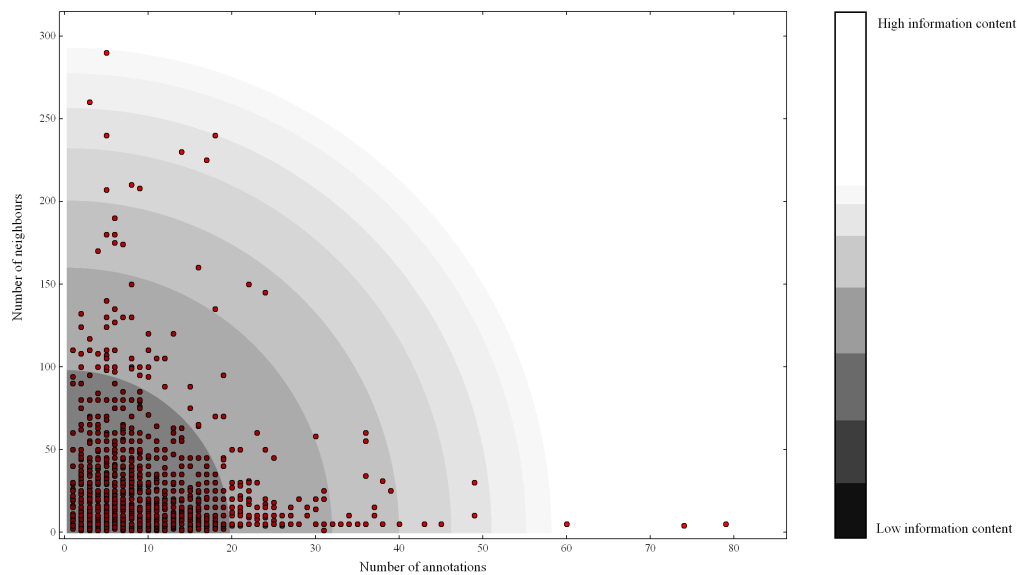


Figure 7.2: Neighborhood / annotation count plot

7.2 Future Work

Our work has been mainly concentrated on applying Gene Ontology protein similarity searches on a protein–protein interaction dataset. We see that there are many possibilities for future work on our and similar areas. Applying techniques like chromosomal proximity, phylogenetic proximity and coexpression analysis on protein–protein interaction networks instead of the gene ontology approach may very well be interesting. Also, testing our results for biological relevance has not been a major concern in our project, and may be a good starting point for others with similar ideas.

We feel that the results from our work are promising, and therefore hope that someone will pick up where we left off. Further research into how the scoring-algorithms can be improved is also interesting, not least by finding some way to solve the *neighborhood size – neighborhood score* correlation problem explained in section 7.1.

We also suggest trying to filter the data from BIND to see if this yields better results. Only including experimentally verified interactions data, and excluding predicted interactions could make it a more accurate and valuable input. Currently, we have found no way of differentiating between experimentally verified interactions and predicted ones, but what BIND does plan to offer, is a field called "Publication Quality". The field is part of the specification, but not yet implemented. Another possibility of excluding less interesting interactions, is the field "Publication Opinion". This is not implemented either, but will contain either *support*, *dispute* or *none*. Filtering out the disputed interactions could possibly make the network more accurate. Also, as figure 7.2 shows, a good portion of the proteins we have are both poorly annotated and have few neighbors. Washing these out of the input data might be a good approach to improving results, as these proteins are likely to be poorly researched and may result in misleading scores.

As noted earlier, merging the server and client could make the overall application more streamlined and easier to deploy.

A possibility for making the statistical selection (see section 4.4.3) better, is to increase the amount of data the statistics are based on. Instead of only using data from BIND, one could use a larger data set. Assuming these annotations are mostly correct, we could probably improve the statistical selection.

As noted in the license discussion (see section 5.4), a re-compilation of the CSA-code to remove the dependency on Cygwin is something that could be performed without too much hassle, and would remove the constraints imposed by the GPL [Free Software Foundation 1991a].

Wrapping CSA into a C-module that can be imported in Python would remove the command-line solution we use to make CSA work. It could also probably be useful for others looking for a solution to the assignment problem.

Removing the dependancy on NetworkX would make installation of our program easier. Either modifying the DAG library to make it do the job of NetworkX, or implementing a graph library to take the place of NetworkX would not require too much work. Either way we do not believe it would affect performace of our program.

Appendix A

Notation and Terminology

Symbol	Explanation
$\mathcal{A}, \mathcal{B}, \mathcal{C} \dots$	Genomes
\mathcal{Q}	Genome containing query protein
\mathcal{S}	Genome to be searched
$\alpha, \beta, \gamma \dots$	Proteins
φ	Query protein
\mathcal{A}_j	Gene ontology category j
$N^n(\alpha)$	n 'th degree neighborhood of protein α (The set of all nodes a maximum of n steps away from α)
$N_i^n(\alpha)$	i 'th neighbor in the n 'th degree neighborhood of protein α (Used for iterating over neighborhoods)
$ N^n(\alpha) $	Absolute value of $N^n(\alpha)$ (The number of n -degree neighbors to α)
$P(\alpha)$	Scoring for protein α in point-distribution system from section 4.2
$C_{\mathcal{A} \rightarrow \mathcal{B}}(\alpha)$	Putative ortholog of α (originally located in \mathcal{A}), located in \mathcal{B} We assume only one orthologous protein exists per genome \mathcal{B} .
$J(\alpha, \mathcal{A}_i)$	Distance in number of jumps from the protein α to the closest protein of category \mathcal{A}_i
$S_{level}(\mathcal{A})$	Score for the GO category \mathcal{A} using <i>level</i> scoring scheme.
$S_{statistics}(\mathcal{A})$	Score for the GO category \mathcal{A} using <i>statistical</i> scoring scheme.
$S(\alpha, \beta)$	Total similarity score for the neighborhoods of proteins α and β . Should include number of levels to search, and scoring scheme (statistical or level) This is omitted for simplicity.
$\hat{S}(\alpha, \beta)$	Overestimated similarity score for the neighborhoods of proteins α and β .

Table A.1: Notation

- Annotated
Annotated proteins are, in contrast to unannotated proteins, categorized in fixed ways we will describe later in the thesis.
- Assignment problem
The problem of finding a maximum (or minimum) weight matching in a weighted, bipartite graph.
- Base
By base, we mean the core of a neighborhood - the protein that the neighborhood is based on.
- Correlation Coefficient
A measure of the strength of the similarity between two sets of variables.
- DAG
Directed, acyclic graph
- GCC
Gnu Compiler Collection.
- Homolog
Indicates genetic relationship. See paralog and ortholog.
- In silico
Research performed via computer simulations
- In vivo
Research performed on living organisms (e.g. mice)
- Level
The level of a protein in a neighborhood denotes how many interaction steps we need to connect it to the base of the neighborhood.
- make-file
File that is shipped with source code meant to instruct the compiler on what to do.
- Ortholog
Orthologous proteins share a common ancestor, but are no longer contained in the same genome. See paralogs.
- Paralog
Paralogous proteins are, like orthologous proteins, mutated versions of the same ancestor, but paralogs are, unlike orthologs, contained in the same genome.
- Psyco
Just in time-compiler for Python, used to speed up Python applications.
- XML-RPC
Remote procedure calling using HTTP as the transport and XML as the encoding.

Table A.2: Terminology

Appendix B

Protein–protein interaction databases

There exists several databases for protein–protein interactions. These include the following: [Stacey]

1. BIND – Biomolecular Interaction Network Database
<http://www.blueprint.org/bind/bind.php>
2. DIP – Database of Interacting Proteins
<http://dip.doe-mbi.ucla.edu/>
3. PIM – Hybrigenics
<http://www.hybrigenics.fr/>
4. PathCalling Yeast Interaction Database
http://portal.curagen.com/pathcalling_portal/index.htm
5. MINT – a Molecular Interactions Database
<http://160.80.34.4/mint/>
6. GRID – The General Repository for Interaction Datasets
<http://biodata.mshri.on.ca/grid/servlet/Index>
7. InterPreTS - Protein–protein interaction prediction through tertiary structure
<http://www.russell.embl.de/interprets/>
8. STRING – predicted functional associations among genes/proteins
<http://www.bork.embl-heidelberg.de/STRING/>
9. Mammalian Protein–protein-protein interaction database (PPI)
<http://fantom21.gsc.riken.go.jp/PPI/>
10. InterDom – database of putative interacting protein domains
<http://interdom.lit.org.sg/>
11. FusionDB – database of bacterial and archaeal gene fusion events
<http://igs-server.cnrs-mrs.fr/FusionDB/>

12. IntAct Project
<http://www.ebi.ac.uk/intact/index.jsp>
13. HPID – The Human protein–protein interaction Database
<http://www.hpid.org/>
14. ADVICE – Automated Detection and Validation of Interaction by Co-Evolution
<http://advice.i2r.a-star.edu.sg/>
15. InterWeaver – Protein–protein interaction reports with online evidence
<http://interweaver.i2r.a-star.edu.sg/>
16. PathBLAST – alignment of protein–protein interaction networks
<http://www.pathblast.org/bioc/pathblast/blastpathway.jsp>
17. ClusPro – a fully automated algorithm for protein–protein docking
<http://nrc.bu.edu/cluster/>
18. HPRD – Human Protein Reference Database
<http://www.hprd.org/>

Appendix C

Basic graph concepts

Graphs theory is relied heavily upon throughout our thesis, and we therefore find it relevant to include a short introduction to graphs here. This appendix will shed light on the various types of graphs and properties that have proved relevant to our work.

A graph is a set of objects we call vertices and edges. The vertices are connected by edges. To be more formal, we can say that a graph is a tuple (V, E) , where V is a finite, non-empty set of vertices and E a set of unordered pairs of distinct vertices.

A typical graph can be seen in figure C.1

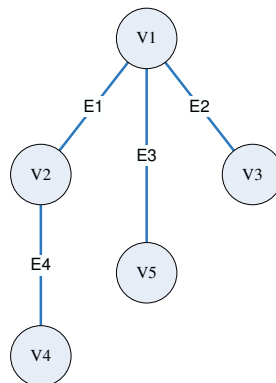


Figure C.1: An example graph.

C.1 Properties of Graphs

A *path* in a graph is a sequence of vertices such that from each of the vertices there is an edge to the previous vertex. The first vertex is called the start vertex, and the last the end vertex. A path in figure C.2 would for instance be $\{(V1, V2), (V2, V4)\}$. Paths can be *directed*, meaning it is a a

path where all the directed edges in the path point the same way. A *cycle* is a path where the start vertex is also the end vertex.

A graph is said to be *connected* if it is possible to establish a path from any one vertex to any other in the graph. A *small worldian network* or graph, is a connected graph with the property that any node can be reached from any other node via a short number of connections.

C.2 Classes of Graphs

A *directed graph* is a graph where the set of edges E is a set of ordered pairs, so that one is a start vertex and the other an end-vertex. Figure C.2 is a typical directed graph.

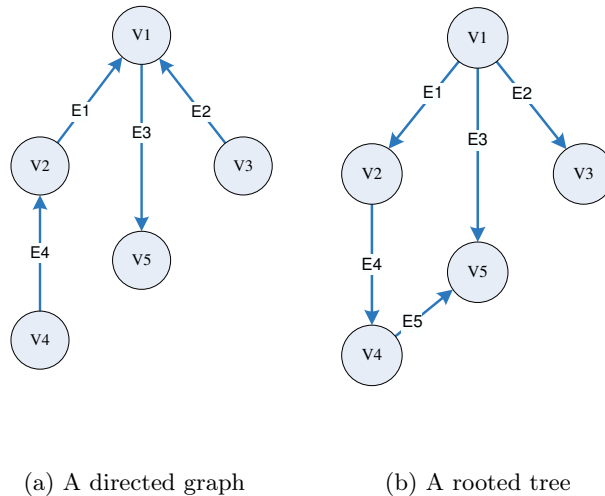


Figure C.2: Examples of graphs

A graph is *acyclic* if it contains no cycles. Figure C.2 is an example of an acyclic graph. An unordered graph is a *tree* if it is both *connected* and *acyclic*. The tree is *rooted* if one vertex has been designated as the root, and the edges have an orientation either away or towards the root. In figure C.2 we see a rooted tree, with V1 as the root. If the graph is directed and acyclic, we call it a DAG. DAGs have no paths starting at a given vertex, and somehow leading to the same vertex. See figure C.2 for reference.

Appendix D

User Manual for BioGraph and BioLogue

This appendix presents a user manual for our programs and their external dependencies.

D.1 Software and hardware requirements

This installation guide assumes a working knowledge of your platform of choice and experience with installing and running programs on it.

Both the client and the server have been tested and found to run on both Windows XP and Linux. It should be possible to make them both work on any platform which has access to Python 2.4 or later, Java 1.4 or later and a working C compiler, e.g gcc. The client requires Java 1.4 or later. It may work with earlier versions, but this has not been tested. The server requires Python 2.4 or later. Our test have been run on an Intel P4 3GHz with 512MB RAM. The client needs at least this much memory, but the server should run with 256MB, but this has not been tested.

D.1.1 Installing Java

If you do not have Java 1.4 or later on your computer, it can be downloaded from <http://java.com/en/download/manual.jsp>. Choose the package that corresponds to your operating system. We have for the sake of convenience included a Windows installer for Java 1.4 on the attached CD. Run the installer and follow its instructions.

D.1.2 Installing Python

The latest version of Python can be downloaded from <http://www.Python.org/download/>. Choose the right package corresponding to your platform and install Python. On Windows, the pre-compiled package should work like a charm. Most (if not all) Linux-distributions will have a pre-compiled Python-package as well. If you can't find a compiled version of Python for your platform, you will have to compile your own. Sources can be downloaded from the same location. We have used Python 2.4 (both on Windows and Linux) in our testing and development. Our code should work

well with newer versions of Python, although this has not been tested by us. We have for the sake of convenience included a Windows installer for Python 2.4, and a source distribution of Python 2.4 on the attached CD.

When Python is installed, Python programs can be run by opening the desired `.py`-file in a file explorer, or by typing

```
Python source.py
```

in a console window (granted, you have the Python installation directory in your `path`-variable).

D.1.3 Installing NetworkX

NetworkX can be downloaded from <http://sourceforge.net/projects/networkx/>. You can choose between a Windows installer or a source distribution. We have for the sake of convenience included both the Windows installer and the source distribution on the attached CD. Important: Install Python before you attempt to install NetworkX.

D.1.4 Installing our Software

Installing the software we have implemented is fairly straight-forward. Locate the directory `Application` on the accompanying CD, and copy all the files to a desired location on your hard drive.

D.1.5 Installing Data Sources

Our application currently uses data gathered from BIND[Bader et al. 2003] and GO[Ashburner et al. 2000]. The files are updated on a regular basis, and the user can download new versions of the files and manually import them into our application. However, this is not necessary, nor is it recommended, as the server automatically downloads the data sources it needs. Automatic updates can be configured using the servers configuration file (see below). The four files the server and the client use are:

- Annotations of proteins
`ftp://ftp.blueprint.org/pub/BIND/data/bindflatfiles/GO/bindgo.1.csv.gz`
- Protein-protein Interactions
`ftp://ftp.blueprint.org/pub/BIND/data/bindflatfiles/bindindex/YYYYMMDD.ints.txt`
- Taxonomy definitions
`ftp://ftp.blueprint.org/pub/BIND/data/bindflatfiles/bindindex/YYYYMMDD.taxon.txt`
- GO category definitions
`http://www.geneontology.org/ontology/gene_ontology.obo`

(YYYYMMDD represents the year, month and date the data was uploaded)

D.1.6 Configuring the Server

The server is configured using a file named `application/BioLogue/bioLogue.cfg`. The file contains several variables, and is read whenever the server is started. This means that you will need to restart the server for configuration changes to take effect. For a further description of the variables, see the file. This should be done before starting the server for the first time.

D.1.7 Configuring the Client

The client is configured using a file named `application/BioGraph/biograph.properties`. This file behaves similarly to that of the server. The most interesting variables are `rpchost` and `rpcport`, describing the address of the RPC server. Additionally, `input` locates the directory where BioGraph will look for its input data. If the user wants to run the server and client on separate machines, the input files (`ints.txt`, `taxon.txt` and `bindgo.1.csv`, per default located in `application/BioLogue/data/`), must be manually synchronized to a folder accessible to the client. Additionally, the address of the server must be updated accordingly in the properties file. This address is set to `localhost` as default.

D.1.8 Starting the RPC Server

The RPC server can be started by typing the command

```
python bioLogue.py
```

in a console window, optionally double-clicking on `bioLogue.py` from a file explorer. If the external data sources are missing, the files described in section D.1.5 will be downloaded automatically. The time this will take depends on your Internet connection. If the files exist, startup time will be few seconds. Note that network port 8000 must be open on the computer running the RPC server.

Important note if you are not running the server on Windows or Linux

The implementation of the CSA algorithm is as we have noted written in C. This means that it will have to be compiled for each platform and architecture you wish to run the server on. We have for the sake of convenience included a compiled version of CSA for Windows and Linux running on x86 architectures in our distribution. These are found in the `application/BioLogue/lib` directory (`csa_linux_x86` and `csa_win_x86.exe`).

D.1.9 Starting the Client

Start BioGraph by running the file `biograph.bat` (Windows systems only), or by typing into a console window

```
java -cp xmlrpc-1.2-b1.jar;. main.BioGraph
```

The application should be up and running after some seconds, and the user may refer to the next chapter for an introduction on using BioGraph.

D.2 Using BioGraph

BioGraph is the front-end for our application. It gathers interaction- and annotation-data, links the various modules together and displays the results for the user in an easy-to-understand manner. Reading the following instructions before using the software is recommended.

D.2.1 The Main Window

The main window is divided into three parts: (1) The *tools panel* is on the top of the window, (2) the *plug-ins panel* is on the right, and (3) the protein–protein interaction (PPI) explorer window is on the bottom left (See figure 5.3 (p. 47) for reference). The following instructions are mainly focused on the PPI explorer.

D.2.2 The PPI explorer

The protein–protein interaction explorer is the large, white window on the bottom left of the application window. It automatically shows the first protein read from the database when BioGraph starts. Proteins are depicted as circles with varying colours indicating which genome they belong to. You will notice that some circles are larger than others. Large circles indicate proteins where all available protein data has been read from the database: Annotations, genome data and interactions to other proteins. Small circles indicate proteins not fully read yet, or chosen not to be shown by the user. Figure D.1 shows an example of both.

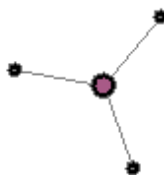


Figure D.1: One expanded protein connected to three collapsed proteins

Expanding and collapsing proteins

Click on a *small* (collapsed) protein with the left mouse button to read all its data. This will result in the protein expanding to a large circle, and its neighbors showing up as small circles. A tooltip will also show the corresponding genome, annotations and neighborhood information for the protein. The tooltip appears when the mouse cursor hovers over the protein. See figure D.2 for an example.

Left-click on the protein again to collapse it, hiding its information. When collapsing proteins, all neighboring proteins that are not (a) expanded or (b) connected to another expanded protein, will be hidden from view.

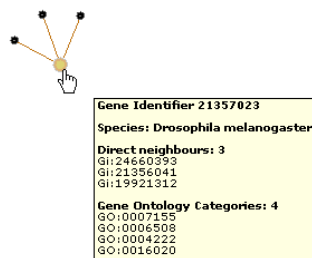


Figure D.2: Tooltip showing information about a protein

Moving proteins

For a better view of the proteins, they can be dragged around the explorer window. Click-and-drag the left mouse button to move proteins around. By simultaneously pressing the *shift* button on the keyboard, un-expanded neighboring proteins will move along with the protein you are moving. This is ideal for moving entire neighborhoods to another place in the explorer, giving a better view of the data.

Selecting proteins

Proteins can be *selected* by the user. Do this by clicking on them with the right mouse button. A blue circle will appear around all proteins that are selected. This command is also necessary for running some of the plug-ins described in section D.2.3. For instance, before comparing two neighborhoods for similarity, one must select the two proteins in the PPI explorer. Figure D.3 shows an example of three selected proteins.

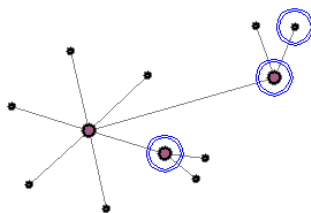






Figure D.3: Three selected proteins

One can also select several proteins simultaneously. Click-and-drag the right mouse button, creating a rectangle over the desired proteins. When releasing the button, the proteins will be selected. Hold the shift-button while doing this, and proteins will be un-selected instead.

Panning and zooming

The view can be panned and zoomed to give a better illustration of the data. Click-and-drag the middle mouse button on an empty portion of the explorer to pan the view. Zoom in and out by

rotating the mouse wheel up and down. The view will be centred on the area you are pointing on. Panning and zooming can also be used by clicking the ,  and  buttons above the explorer. To pan up, down, left and right, click on the corresponding position on the  button.

D.2.3 Running Plug-ins

All advanced functionality in BioGraph is implemented as plug-ins. All available plug-ins are listed to the right of the PPI explorer window, and started by clicking its corresponding button. Some plug-ins require the user to select proteins before starting, and will request the user to do so if it hasn't already been done. One example is the protein-protein comparing plug-in, which requires that two proteins are selected in the explorer. The following sections describe the plug-ins that come with the current version of BioGraph.

Find protein

The simplest plug-in is also a very helpful one. It finds any protein in the database based on the corresponding Gene Identifier (GI) number. Type in the GI and press the search button. The plug-in will find it, expand it and also select it, making it easy to notice for the user.

Compare protein to protein

This plug-in runs the algorithms described in section 4.4 and displays the score between two protein neighborhoods. Start off by selecting the two proteins whose neighborhoods you want to compare. After the plug-in is started, type the number of neighborhood levels you want to include in the search (1 by default). Also, choose which information content measurement method you want to apply (level or statistical method), and press the search button. If the RPC server is up and running, the result should come within a fraction of a second.

Compare protein to genome

The user selects one protein before starting this plug-in, selects a genome to search, and the number of neighborhood levels to include in the search. The plug-in then sequentially compares the selected protein's neighborhood to all the protein neighborhoods in the given genome, logging the similarities, and finally writing them to an output file. This is so far the most valuable function implemented in BioGraph, and gives researchers the possibility to discover unknown functional links between proteins.

Compare protein to list

Similar to the previous plug-in, but this plug-in allows the user to submit a comma-separated list of Gene Identifiers. The plug-in will sequentially compare the selected protein to the proteins in the list, and write the results to disk.

Bibliography

M. Ashburner, C. A. Ball, J. A. Blake, D. Botstein, H. Butler, J. M. Cherry, A. P. Davis, K. Dolinski, S. S. Dwight, J. T. Eppig, M. A. Harris, D. P. Hill, L. Issel-Tarver, A. Kasarskis, S. Lewis, J. C. Matese, J. E. Richardson, M. Ringwald, G. M. Rubin, and G. Sherlock. Gene ontology: Tool for the unification of biology. *Nature Genetics*, 25:25–29, may 2000.

V. Athitsos, J. Alon, S. Sclaroff, and G. Kollios. Boostmap: A method for efficient approximate similarity rankings. In *cvpr*, pages II:268–275, 2004.

G. D. Bader, D. Betel, and C. Hogue. BIND - The Biomolecular Interaction Network Database. *Nucleic Acids Research*, 31(1):248–50, 2003.

D. Beazley. Simplified wrapper and interface generator. URL <http://www.swig.org/>.

K. Bold, H. Rozenfeld, and B. Wohlberg. Networkx. URL <http://networkx.sourceforge.net/>.

F. M. Couto, M. J. Silva, and P. Coutinho. Implementation of a functional semantic similarity measure between gene-products. DI/FCUL TR 03–29, Department of Informatics, University of Lisbon, November 2003. URL <http://www.di.fc.ul.pt/tech-reports/03-29.pdf>.

M. Deng, Z. Tu, F. Sun, and T. Chen. Mapping gene ontology to proteins based on protein-protein interaction data. *Bioinformatics*, 20(6):895–902, 2004.

M. Deng, K. Zhang, S. Mehta, T. Chen, and F. Sun. Prediction of protein function using protein-protein interaction data. In *CSB '02: Proceedings of the IEEE Computer Society Conference on Bioinformatics*, page 197, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1653-X.

DIMACS. Dimacs implementation challenges. URL <http://dimacs.rutgers.edu/Challenges/>.

G. Ewing. Pyrex - a language for writing python extension modules. URL <http://www.cosc.canterbury.ac.nz/~greg/python/Pyrex/>.

C. Faylor and C. Vinschen. Cygwin. URL <http://cygwin.com/>.

I. Free Software Foundation. Gnu general public license, 1991a. URL <http://www.gnu.org/copyleft/gpl.html>.

I. Free Software Foundation. Gnu general public license, 1991b. URL <http://www.gnu.org/copyleft/lesser.html>.

- K. Fukuda and T. Matsui. Finding all minimum-cost perfect matchings in bipartite graphs. *Networks: An International Journal*, 22(1), 1992. URL <http://citeseer.ist.psu.edu/fukuda91finding.html>.
- A. Goldberg. Andrew goldberg's network optimization library, 2002. URL <http://www.avglab.com/andrew/soft.html>.
- A. V. Goldberg and R. Kennedy. An efficient cost scaling algorithm for the assignment problem. *Math. Program.*, 71(2):153–177, 1995. ISSN 0025-5610.
- P. Gomes, F. C. Pereira, N. Seco, J. L. Ferreira, and C. Bento. Supporting creativity in software design. In *Proceedings of the AISB'02 Symposium*, 2002. URL http://ailab.dei.uc.pt/dlfile.php?fn=220_extpaper_AISB2002_CRC.pdf&idp=220&ext=1.
- L. Gonick and M. Wheelis. *The Cartoon Guide to Genetics*. Harper Perennial, New York, updated edition, 1991.
- H. Hishigaki, K. Nakai, T. Ono, A. Tanigami, and T. Takagi. Assessment of prediction accuracy of protein function from protein-protein interaction data. *Yeast*, 18:523–31, 2001.
- A. Istvan. pygraphlib - a python graph library. URL <http://pygraphlib.sourceforge.net/doc/public/pygraphlib-module.html>.
- J. Jiang and D. Conrath. Semantic similarity based on corpus statistics and lexical taxonomy. In *Proceedings on International Conference on Research in Computational Linguistics, Taiwan*, 1997.
- B. F. J. Manly. *Randomization, Bootstrap and Monte Carlo Methods in Biology*. Chapman and Hall, 1997.
- B. P. Kelley, R. Sharan, R. M. Karp, T. S. and David E Root, B. R. Stockwell, and T. Ideker. Conserved pathways within bacteria and yeast as revealed by global protein network alignment. *PNAS*, 100(20):11394–11399, 2003. URL <http://www.pnas.org/cgi/content/abstract/100/20/11394>.
- J. Knowles and G. Gromo. Target selection in drug discovery. *Nature*, 2:63–69, january 2003.
- H. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2:83–97, 1955.
- D. Lin. An information-theoretic definition of similarity. In *Proceedings of International Conference on Machine Learning, Madison, Wisconsin*, August 1998.
- S. T. March and G. F. Smith. Design and natural science research on information technology. *Decision Support Systems*, 15(4):251–266, 1995.
- D. Mertz. Charming python b9: Making python run as fast as c, 2002. URL http://gnosis.cx/publish/programming/charming_python_b9.html.
- G. A. Miller. Wordnet: a lexical database for english. *Commun. ACM*, 38(11):39–41, 1995. ISSN 0001-0782.

- C. Peters, J.-J. van der Heijden, M. Khan, and A. Norlander. Mingw - minimalist gnu for windows. URL <http://www.mingw.org/>.
- P. Resnik. Using information content to evaluate semantic similarity in a taxonomy. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence, Montreal, Canada, 1995*.
- A. Rigo. Representation-based just-in-time specialization and the psyco prototype for python. In *PEPM '04: Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 15–26, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-835-0.
- J. Roskind. The python profiler. URL <http://docs.python.org/lib/profile.html>.
- B. Schwikowski, P. Uetz, and S. Fields. A network of protein-protein interactions in yeast. *Nature Biotechnology*, 18:1257–1261, 2000.
- B. Smith, J. Williams, and S. Schulze-Kremer. The ontology of the gene ontology. In *Proceedings of AMIA Symposium 2003*, pages 609–13, 2003.
- K. Stacey. Rosalind Franklin Centre for Genomics Research. URL <http://www.hgmp.mrc.ac.uk/GenomeWeb/prot-interaction.html>. Date of last access: June 12th, 2005.
- N. A. Streit, J. Geißler, T. Holmer, S. Konomi, C. Müller-Tomfelde, W. Reischl, P. Rexroth, P. Seitz, and R. Steinmetz. i-LAND: An interactive landscape for creativity and innovation. In *CHI*, pages 120–127, 1999. URL <http://citeseer.ist.psu.edu/streitz99iland.html>.
- R. L. Tatusov, E. V. Koonin, and D. J. Lipman. A genomic perspective on protein families. *Science*, 278:631–637, 1997.
- G. van Rossum. Python patterns - implementing graphs. URL <http://www.python.org/doc/essays/graphs.html>. Date of last access: June 12th, 2005.
- A. Warr and E. O’Neill. Understanding design as a social creative process. In *Proceedings of the 5th conference on Creativity & cognition*, pages 118–127. ACM Press, 2005. ISBN 1-59593-025-6.
- D. Winer. Xml-rpc specification, 1999. URL <http://www.xmlrpc.com/spec/>.
- T. Winston. Deltagen inc., 2005. URL <http://www.deltagen.com/target/invivo.html>.