Augmenting DIAMOnD:

A Method for Improving Disease Networks Among Human Genes

A Major Qualifying Project
submitted to the Faculty of
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Bachelor of Science,
Mathematical Sciences/Computer Science

Submitted by Kevin Specht

Advised by Dmitry Korkin and Zheyang Wu

ABSTRACT

One of the most significant advancements in modern medicine has been the discovery of relationships between diseases and certain genes. These "disease genes" enable medical professionals to pinpoint the exact cause and location of the disease. In 2015, a group of scientists developed an algorithm known as DIAMOnD (DIseAse MOdule Detection) in order to find more disease genes. DIAMOnD works by examining the significance of the known connections between genes in a biological network and the set of disease genes, and then adding the most significantly connected gene from the network to the set of disease genes. For this project, I sought to improve the performance of DIAMOnD to more accurately determine the next disease gene to be added. To do this, I changed the probability distribution used by the algorithm and tested both the original algorithm and the "augmented" algorithm on certain diseases and networks.

TABLE OF CONTENTS

ABSTRACT…… 2

TABLE OF CONTENTS…… 3

TABLE OF FIGURES…… 5

CHAPTER 1: INTRODUCTION…….. 7

CHAPTER 2: BACKGROUND……9

2.1 Components of Biological Processes…… 9

2.1.1 Types of Biological Molecules…….. 9

2.1.2 Biological Databases…… 13

2.2 Network Medicine Overview…….. 16

2.2.1 Properties of Biological Networks…….. 17

2.2.2 Types of Biological Networks……..20

2.2.3 Advances in Network Medicine…….. 24

CHAPTER 3: METHODOLOGY…… 27

3.1 The DIAMOnD Algorithm…….. 28

3.1.1 How DIAMOnD Works……..28

3.1.2 Analysis of DIAMOnD…….. 31

3.2 DIAMOnD Testing…….. 32

3.2.1 Testing Methods……32

3.2.2 Validation Methods……....33

3.3 Augmenting DIAMOnD…….. 35

3.3.1 The Augmented DIAMOnD Algorithm……....35

3.3.2 Comparing the Algorithms…….. 37

CHAPTER 4: RESULTS AND ANALYSIS……..38

4.1 DIAMOnD Results and Analysis…….. 38

4.2 Augmented DIAMOnD Results and Analysis……..46

CHAPTER 5: CONCLUSION AND RECOMMENDATIONS……57

ACKNOWLEDGEMENTS……...59

BIBLIOGRAPHY…….. 60

APPENDIX A: LIST OF FILES…….. 62

APPENDIX B: PROGRAMS…….. 70

B.1 DIAMOnD Algorithm…….. 70

B.2 Augmented DIAMOnD Algorithm……..82
B.3 Parse Annotations……...96
B.4 Random Selection……...98

TABLE OF FIGURES

Figure 1: Three-dimensional structure of a Myoglobin protein..... 10
Figure 2: Comparison of DNA and RNA structure..... 12
Figure 3: Manhattan plot for genome-wide association study..... 15
Figure 4: DAVID functional annotation tool..... 16
Figure 5: Models of a human disease network and disease gene network..... 19
Figure 6: Model of a PPI network..... 21
Figure 7: Model of a co-expression network..... 24
Figure 8: Process of finding disease modules..... 28
Figure 9: Progress of the DIAMOnD algorithm..... 30
Figure 10: Set of terms obtained by DAVID functional annotation tool..... 34
Figure 11: Table of final true positive values for DIAMOnD testing..... 39
Figure 12: Validation of DIAMOnD genes for Behcet disease with PPI network..... 41
Figure 13: Validation of DIAMOnD genes for Crohn's disease with PPI network..... 42
Figure 14: Validation of DIAMOnD genes for leukemia with PPI network..... 42
Figure 15: Validation of DIAMOnD genes for sarcoma with PPI network..... 43
Figure 16: Validation of DIAMOnD genes for vasculitis with PPI network..... 43
Figure 17: Validation of DIAMOnD genes for Behcet disease with co-expression network..... 44
Figure 18: Validation of DIAMOnD genes for Crohn's disease with co-expression network..... 44
Figure 19: Validation of DIAMOnD genes for leukemia with co-expression network…. 45
Figure 20: Validation of DIAMOnD genes for sarcoma with co-expression network….. 45
Figure 21: Validation of DIAMOnD genes for vasculitis with co-expression network…. 46
Figure 22: Table of final true positive values for augmented DIAMOnD testing……. 48
Figure 23: Comparison for Behcet disease of DIAMOnD with PPI network and augmented DIAMOnD with supplementary co-expression network..... 51
Figure 24: Comparison for Crohn's disease of DIAMOnD with PPI network and augmented DIAMOnD with supplementary co-expression network..... 52
Figure 25: Comparison for leukemia of DIAMOnD with PPI network and augmented DIAMOnD with supplementary co-expression network..... 53
Figure 26: Comparison for sarcoma of DIAMOnD with PPI network and augmented DIAMOnD with supplementary co-expression network..... 54
Figure 27: Comparison for vasculitis of DIAMOnD with PPI network and augmented DIAMOnD with supplementary co-expression network..... 55
Figure 28: Comparison for Behcet disease of DIAMOnD with co-expression network and augmented DIAMOnD with supplementary PPI network..... 56

Figure 29: Comparison for Crohn's disease of DIAMOnD with co-expression network and augmented DIAMOnD with supplementary PPI network..... 57
Figure 30: Comparison for leukemia of DIAMOnD with co-expression network and augmented DIAMOnD with supplementary PPI network..... 58
Figure 31: Comparison for sarcoma of DIAMOnD with co-expression network and augmented DIAMOnD with supplementary PPI network..... 59
Figure 32: Comparison for vasculitis of DIAMOnD with co-expression network and augmented DIAMOnD with supplementary PPI network..... 60
Figure 33: List of files used during testing..... 73

CHAPTER 1: INTRODUCTION

For the past several years, one of the most rapidly growing areas of medicine has been network medicine. This relatively new branch of healing involves discovering biological links between different genes in the human body. Thanks to the Human Genome Project, we know all of the genes that make up the human body and can tinker with them as needed, like fixing certain parts of a car. However, in this case we don't know how all of the parts are wired because we don't know what biological links exist between different genes or how they relate to diseases. Therefore, many biologists and data scientists have taken up the cause to discover as many biological links in the human body as possible in order to form a network. In the same way that a computer virus spreads over a network, so can a disease take over all genes in a network if one gene is affected. However, the genes most associated with a particular disease tend to be gathered together in "neighborhoods", forming smaller connected components known as disease modules. If one gene in a disease module is affected by a particular disease, doctors and scientists can track the disease's progress by looking at other genes in the disease module. For this reason, discovering more disease modules may become crucial to fighting diseases in the years to come.

In recent years, data scientists have developed several algorithms which can be used, with varying degrees of accuracy, to build disease modules from random links discovered between genes in the human body. One technique used by data scientists has been to determine the significance of links between genes using statistical analysis rather than observe how many links there are to a particular gene. To this end, several scientists, including Susan Ghiassian, Jorg Menche, and Albert-Laszlo Barabasi, developed an algorithm called DIseAse MOdule Detection (DIAMOnD) in order to determine additional genes to add to a disease module based on an initial set of genes (called seed genes) from the module and a biological network containing links between pairs of genes. The algorithm works by determining the significance of connections between genes in the network and the set of seed genes and adding the most significantly connected gene not currently in the disease module for each iteration. Once all iterations are complete, the user will have a set of new genes to add to the module. However, DIAMOnD does not guarantee that all of the added genes will be correct, which is why this project sought to improve its accuracy.

Throughout this project, I searched for ways to enhance DIAMOnd, firstly by conducting tests to replicate the performance of the original creators. This included testing the algorithm with different diseases and networks and then comparing the results to those of a random test. I changed the algorithm by altering the degree of probability with which two genes are associated. The new algorithm examines additional networks to determine if any of them contain any links that also exist in the original network, and gives these links greater weight if they do. The idea is to bring additional biological conditions and information into account when determining the probability of a biological link being in a random network, and then to change the probability distribution used by the algorithm accordingly. I tested this augmented algorithm against the original DIAMOnD algorithm to determine whether it provides more correct genes for a disease module than the original algorithm does. Both algorithms were developed in Python and tested using biological molecules and networks obtained from a few key databases. I also developed additional Python programs to find a random selection of genes to add to the disease module and to validate the genes discovered by each algorithm to see if they actually belong to that module.

This project helps open new doors of experimentation for finding genes in disease modules, which can help to advance network medicine. The more accurate disease module detection methods are, the more accurate disease related genes we will find. This process will only improve because as we discover more biological processes within the human body, we uncover new information about the functions each molecule connected by the process performs. This means that someday, we may be able to instantly identify the genes causing any disease, the same way we can identify the parts causing problems in a car. For this reason, the day may come when network medicine forms the backbone of all disease treatment.

CHAPTER 2: BACKGROUND

In order to improve the performance of DIAMOnD, it was necessary to do research on past work related to disease networks. Although network medicine is a relatively new field, it has already been used to make significant advances in disease detection and treatment. Using disease networks requires a good understanding of genetic biology, the underlying structure of general networks, and how to use statistics and data science to analyze said networks. Enhancing the algorithm also required the use of several major biological databases and testing methods used to aid researchers in expanding disease modules. Hopefully, this project will help to contribute to the ongoing search for better methods of disease network detection and construction.

2.1 Components of Biological Processes

In order to understand how network medicine works, it is necessary to first understand the different components that make up biological processes in the networks. These nodes can be a wide range of biological materials, from a single atom to an entire organism. The most common molecules in biological networks, however, include genes, proteins, and RNA. All of these molecules perform very important functions in the human body, and problems with any of them can lead to minor and sometimes major disorders. Because of this, a number of resources exist to store information on various biological molecules and their relationships.

2.1.1 Types of Biological Molecules

Proteins are some of the most important molecules in the human body, performing a wide variety of useful functions, including preventing diseases. They are made up of various smaller units known as amino acids, which are attached to each other in long chains to form proteins. The arrangement of amino acids determines the structure and function of each protein (Genetics Home Reference 2017), such as the one in Figure 1. The ubiquity of proteins in organisms makes it necessary for those organisms to create new ones when they run out. To do this, they take in new amino acids from other organisms and fitting them together using structures called ribosomes. There are several types of proteins that all perform essential functions, and problems with any of them can cause unfortunate complications. Antibodies are proteins that "bind to specific foreign

particles, such as viruses and bacteria, to help protect the body" (Genetics Home Reference 2017), so they play a big role in disease control. Enzymes, on the other hand, carry out cell chemical reactions and read genetic information in DNA to form new molecules (Genetics Home Reference 2017). However, they can still cause issues if they malfunction due to chemical reactions gone wrong or misreading of DNA sequences. Messenger proteins handle signals that "coordinate biological processes between different cells, tissues, and organs" (Genetics Home Reference 2017), so it is important for them to keep working in order to keep biological processes running on schedule. Structural components provide support for cells and allow the body to move, so without them, the organism may find it harder to function. Transport proteins "bind and carry atoms and small molecules within cells and throughout the body" (Genetics Home Reference 2017). Therefore, if one failed to take a molecule where it needed to go, the results could be troublesome for the organism. The essential nature of proteins in organisms is the reason for their prevalence in many biological networks, mainly protein-protein networks.

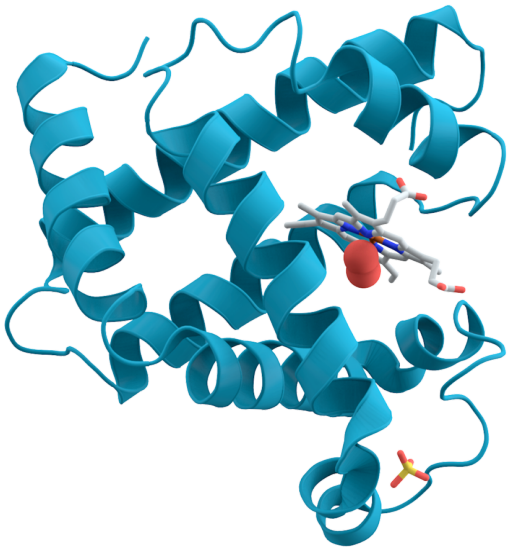


Figure 1: Three-dimensional structure of a Myoglobin protein.
Retrieved from https://en.wikipedia.org/wiki/Protein#/media/File:Myoglobin.png

Genes are the "basic physical and functional unit of heredity" (Genetics Home Reference 2017), the process of determining the traits of a new organism from its parents' genes. Genes are made up of many strands of DNA (deoxyribonucleic acid), the molecule responsible for storing

genetic information. This information is mainly instructions on how to create proteins that the organism needs. Any gene can have "from a few hundred DNA bases to more than 2 million bases" (Genetics Home Reference 2017), depending on the complexity of the genetic information. For a long time, scientists struggled to determine the number of genes in the human body and their various types. However, advances in computing technologies enabled them to create the Human Genome Project, whose mission is to document all genes in the human body. The Human Genome Project estimates that "humans have between 20,000 and 25,000 genes" (Genetics Home Reference 2017), and has documented much information about human genes. Therefore, scientists now have a wealth of genetic information at their disposal and can analyze genes more closely. All humans have roughly the same genes, with only a 1% difference from person to person. However, there are many different alleles, or "forms of the same gene with small differences in their sequence of DNA bases" (Genetics Home Reference 2017), throughout the human genome. These have created a large diversity in human traits, such as unique physical appearances, even though most of our core genes are the same. These differences can sometimes lead to mutations that either improve certain traits or create serious disorders, which is why genes are often involved in disease modules. Every person has two copies of each gene (one per parent), except for sex genes, and numerous genes are combined to form structures called chromosomes (Genetics Home Reference 2017). Each cell nucleus in the human body contains 46 chromosomes (23 from each parent, with one copy of each gene per parent). In order for these genes to express information, they require various interactions with both other genes and with molecules such as proteins which help them send information and maintain their structure. This is why genes have such a large role in many biological networks, such as regulatory networks and co-expression networks.

RNA (ribonucleic acid) is an important molecule whose main function is "to transfer the genetic code needed for the creation of proteins from the nucleus to the ribosome" (Mandal 2013), the structure in cells that makes proteins. It is similar to DNA, but it has different bases, a different structure, and performs different functions, as Figure 2 demonstrates. While DNA stores genetic information in the nucleus, RNA can move it around, which "keeps the DNA and genetic code protected from damage (Mandal 2013). RNA molecules can also act as enzymes to stimulate chemical reactions, which means that like proteins, they can act in many biological processes. RNA

is initially formed from DNA via a process called transcription (Mandal 2013), and from there it can fulfill a variety of functions. Since RNA serves as the bridge between genes and proteins, organisms can't function without it. Therefore, problems with RNA can contribute to many problems in organisms, including diseases. Messenger RNA (mRNA) "carries information from DNA to...ribosomes" (Mandal 2013), so if it experienced any problems, the information in genes could be rendered useless. Ribosomal RNA (rRNA) combine with proteins to build ribosomes, so without them, an organism would not be able to properly manufacture proteins. Transfer RNA (tRNA) is used to decode the message in mRNA into an actual protein sequence, so without them, all of the work of the mRNA would be for nothing because no one could read it. Due to all of these issues, RNA appears frequently throughout biological networks, such as regulatory networks.

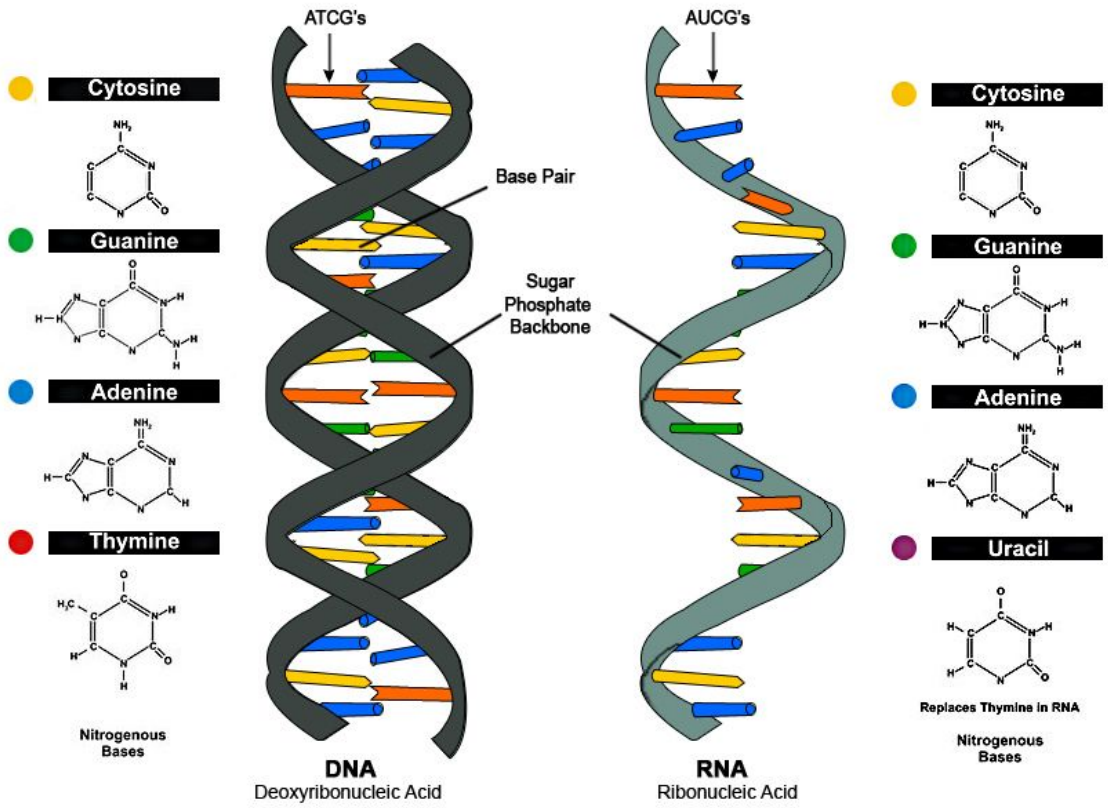


Figure 2: Comparison of DNA and RNA structure.

Retrieved from https://c1.staticflickr.com/8/7377/10082811755_d108c78942_b.jpg

I note that for this work, I have mostly referred to the search for disease modules in terms of genes. However, disease modules may contain genes, proteins, RNA, or other molecules depending on which network is used to obtain them. This is because all of these molecules play an essential role in the transfer of genetic information throughout the body, to which many diseases can be traced. Genes carry the DNA structures with all of the information necessary to create an organism and give it certain characteristics. RNA transports this information all over the cell and brings it to the ribosomes. Here, the information is used to build proteins, which then perform a variety of functions throughout the cell. Therefore, while I may only refer to genes when speaking of disease modules, all of these molecules can be a part of them.

2.1.2 Biological Databases

In order to improve the collection of biological information, several major databases have been set up to store information about various biological molecules. The main information stored in these databases is the names and characteristics of various proteins, genes, RNA molecules, and other materials that are necessary for organisms to function. However, some networks also contain information about biological processes, as well as the biological networks that result from them. Some can even be used to run tests on biological networks to learn more about them. Therefore, such databases make it possible to perform experiments like DIAMOnD, forming the foundation upon which this project is built.

One of the databases which proved most helpful for this project was OMIM, or Online Mendelian Inheritance in Man. OMIM is "a comprehensive, authoritative compendium of human genes and genetic phenotypes that is freely available and updated daily" (McCusick 1966). It contains information on various human genes obtained from the Human Genome Project, as well as some of the major functions and disorders that they are associated with. It was originally created "in the early 1960s by Dr. Victor A. McCusick as a catalog of mendelian traits and disorders" (McCusick 1966), mendelian referring to genetics pioneer Gregor Mendel. It focuses on the relationship between the genotype (the genetic information in the body) and the phenotype (the physical expression of that information), which includes physical traits, strengths, and disorders. For instance, the gene Interleukin 23 Receptor (IL23R) is found to be commonly associated with Crohn's Disease, which is

why in most biological networks, IL23R is part of a set of genes associated with Crohn. This information provides a good baseline for any scientists studying the genes that make up disease modules, and proved essential for this project.

Another important database for this project was the Genome-Wide Association (GWAS) Catalog. A genome-wide association study is "an approach that involves rapidly scanning markers across the complete sets of DNA, or genomes, of many people to find genetic variations associated with a particular disease" (National Human Genome Research Institute 2015). Rather than simply report any disorders that may result from problems with a particular gene, this study performs a comprehensive search to find issues caused by any genes that are associated with a particular disease, using statistical tools like the Manhattan Plot in Figure 3 on a large sample of people. This is possible because "With the completion of the Human Genome Project in 2003...researchers now have a set of research tools that make it possible to find the genetic contributions to common diseases" (National Human Genome Research Institute 2015). This test is particularly useful for building disease modules, since the genes found by this test are often confirmed to be associated with their given disease by many biological networks. To perform the test, researchers make one group of people with a disease and one without, and then "obtain DNA from each participant, usually by drawing a blood sample or by rubbing a cotton swab along the inside of the mouth to harvest cells" (National Human Genome Research Institute 2015). Then, "if certain genetic variants are found to be significantly more frequent in people with the disease compared to people without the disease, the variations are said to be "associated" with the disease" (National Human Genome Research Institute 2015). The GWAS Catalog is "provided jointly by the National Human Genome Research Institute (NHGRI) and the European Bioinformatics Institute (EMBL-EBI)" (GWAS Catalog 2017), so it provides comprehensive information on many GWAS studies. However, GWAS information is by no means complete, due to the intensive nature of the test, which is why algorithms like DIAMOnD are still necessary.

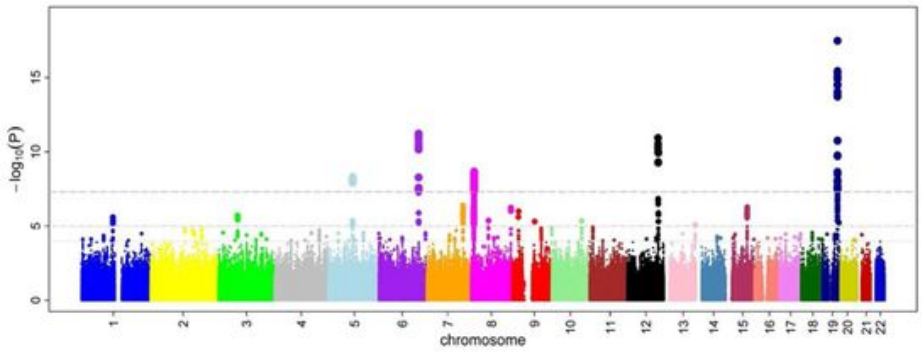


Figure 3: Manhattan plot for genome-wide association study.
Retrieved from https://upload.wikimedia.org/wikipedia/commons/1/12/Manhattan_Plot.png

The most useful database for this project was probably DAVID, which was designed to "validate" molecules within biological networks. It was originally developed by the Laboratory of Human Retrovirology and Immunoinformatics at Leidos Biomedical Research (DAVID 2009). Its main purpose is to gather all of the biological terms associated with a particular gene or set of genes, as part of a process called gene ontology. The "Gene Ontology project provides controlled vocabularies of defined terms representing gene product properties" (Gene Ontology Consortium 1999), meaning that it associates genes with their related biological terms. For instance, the myb gene is often associated with the term "cell growth". The first type of GO term is a cellular component, which describes "a component of a cell that is part of a larger object, such as an anatomical structure" (Gene Ontology Consortium 1999). The second type is a biological process, which "describes a series of events accomplished by one or more assemblies of molecular functions" (Gene Ontology Consortium 1999). The third type is molecular functions, which describe "activities that occur at the molecular level, such as "catalytic activity" or "binding activity"" (Gene Ontology Consortium 1999).

While GO information is centered at the Gene Ontology Consortium, DAVID provides a number of useful features for studying gene ontology. The best of these is the functional annotation tool, a statistical test that takes a list of genes and determines all of the GO terms that are associated with them, as shown in Figure 6. The DAVID KnowledgeBase contains "dozens of heterogeneous public databases that are comprehensively integrated by a unique single linkage method developed by DAVID team" (DAVID 2009), which means that it can pull information from various databases

into one source. This way, it can use multiple types of gene IDs in order to retrieve the corresponding GO terms from its data repository. Once the test has been run, DAVID compiles the GO terms into separate lists for cellular components, biological processes, and molecular functions. Each list contains a set of terms associated with the genes, as well as statistics regarding how many of the genes are associated with each term. Since biological networks tend to form modules of nodes that perform similar functions (and therefore share similar GO terms), DAVID is a great resource for validating such networks.

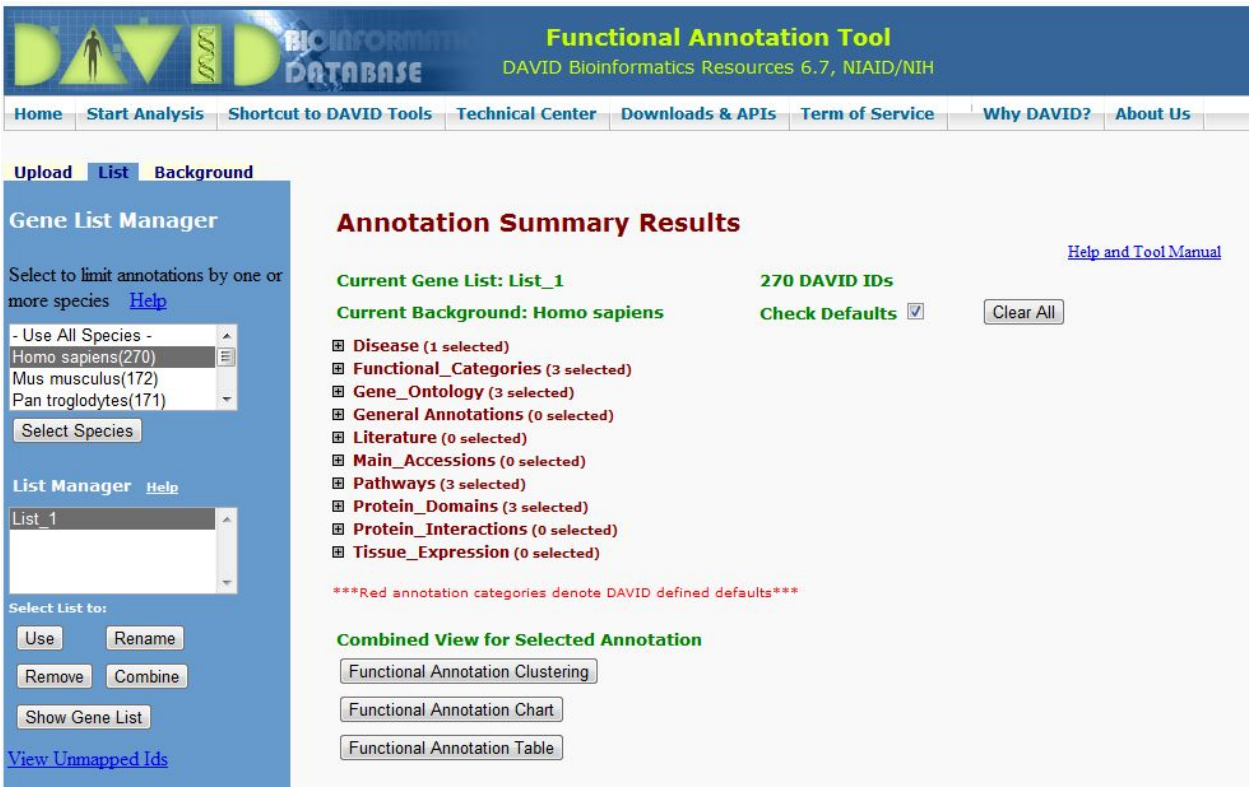


Figure 4: DAVID functional annotation tool.

Retrieved from https://david.ncifcrf.gov/home.jsp

2.2 Network Medicine Overview

The main idea behind network medicine is that we can learn a great deal about different components of the human body by observing how they interact with other components, since "Most cellular components exert their functions through interactions with other cellular components" (Barabasi 2011). These interactions between components of the human body,

including genes, proteins, and RNA molecules, serve as vertices or nodes for a vast network of connections known as the human interactome. Since "the number of cellular components that serve as the nodes of the human interactome easily exceeds 100,000" (Barabasi 2011), and "the number of functionally relevant interactions...is expected to be even larger" (Barabasi 2011), the task of determining links between these components is daunting. However, a number of advances in network theory and data science have made it possible for scientists to learn a lot more about the human interactome in recent years. These discoveries have already made several advances in medicine possible, and future discoveries will likely cause network medicine to advance at an exponential rate.

2.2.1 Properties of Biological Networks

Many of the recent advances in network medicine were only possible because of an understanding of the properties of the underlying networks. If a gene or protein is associated with a particular disease, then knowing about its interactions can provide valuable information about how the disease will spread. However, "only about 10% of human genes have a known disease association; thus, do disease genes have unique, quantifiable characteristics that distinguish them from other genes?" (Barabasi 2011). For answers, scientists have studied the arrangement of nodes and links inside biological networks, or network topology, to look for patterns. They found that several principles of network theory make it possible to analyze disease networks, which are biological networks that focus on nodes and links related to the spread of diseases.

Every network is made up of nodes and links between the nodes, forming a graph of interconnected points. Some examples of networks include the Internet (documents and URLs), power grids (generators and power transmission lines), and the United States Interstate Highway System (cities and roads). The degree of a node is the number of links it has, and the degree distribution is the number of nodes containing each degree (Barabasi 2013). When networks were first discovered, scientists assumed that the distribution of nodes and links in networks was random, and that there was no likelihood that one node would be more connected than another but that most nodes would have roughly the same number of connections. However, as more discoveries were made and networks came into clearer focus, scientists realized that networks were not

necessarily random and that some had nodes with much higher degrees than others. Since then, scientists have used the degree distributions inside of networks to determine the probability that one node is connected to another. This has led to the formation of several network probability distributions, such as the hypergeometric distribution used to calculate the connection probability in the DIAMOnD algorithm.

One aspect of network theory that pops up often in disease networks is the presence of highly connected nodes, or "hubs". For example, the Internet is a network containing nodes in the form of documents which are connected by links in the form of URLs. However, some documents are much more connected than others due to popularity or usefulness. Such networks are called scale-free networks, which means that they have a few highly connected nodes and a great many "smaller" nodes with only a few connections. Most biological networks are scale-free because there are a few genes or proteins that play a more essential role in biological processes than others, due to evolution or other factors, and form hubs in the networks. In disease networks, "hub proteins tend to be encoded by essential genes" (Barabasi 2011), and "genes encoding hubs are older and evolve more slowly than genes encoding non-hub proteins" (Barabasi 2011).

The tendency of disease networks to focus more links on a few nodes makes them more robust, which means that if one node is removed, the network will still be able to carry out most of its basic functions (Barabasi 2013). If a disease hits the human body, then this distribution of a few bigger nodes connected to many smaller nodes works to its advantage because the majority of nodes have only a few connections, so if one of them is hit, the network is less likely to go down. However, if the disease does hit a hub, then it will be able to spread to all of the nodes that the hub is connected to, which is why they are sometimes referred to as the "Achilles heel" of scale-free networks (Barabasi 2013). For a scale-free network, the probability that a node i is connected to another node in the network is the degree k of the node divided by the sum of the degrees of all other nodes in the network. Therefore, when a new node is added to the network, there is a higher probability that it will be connected to a hub because it has a higher degree, so the network retains its form. Figure 5 demonstrates the scale-free human-disease network and disease gene network.

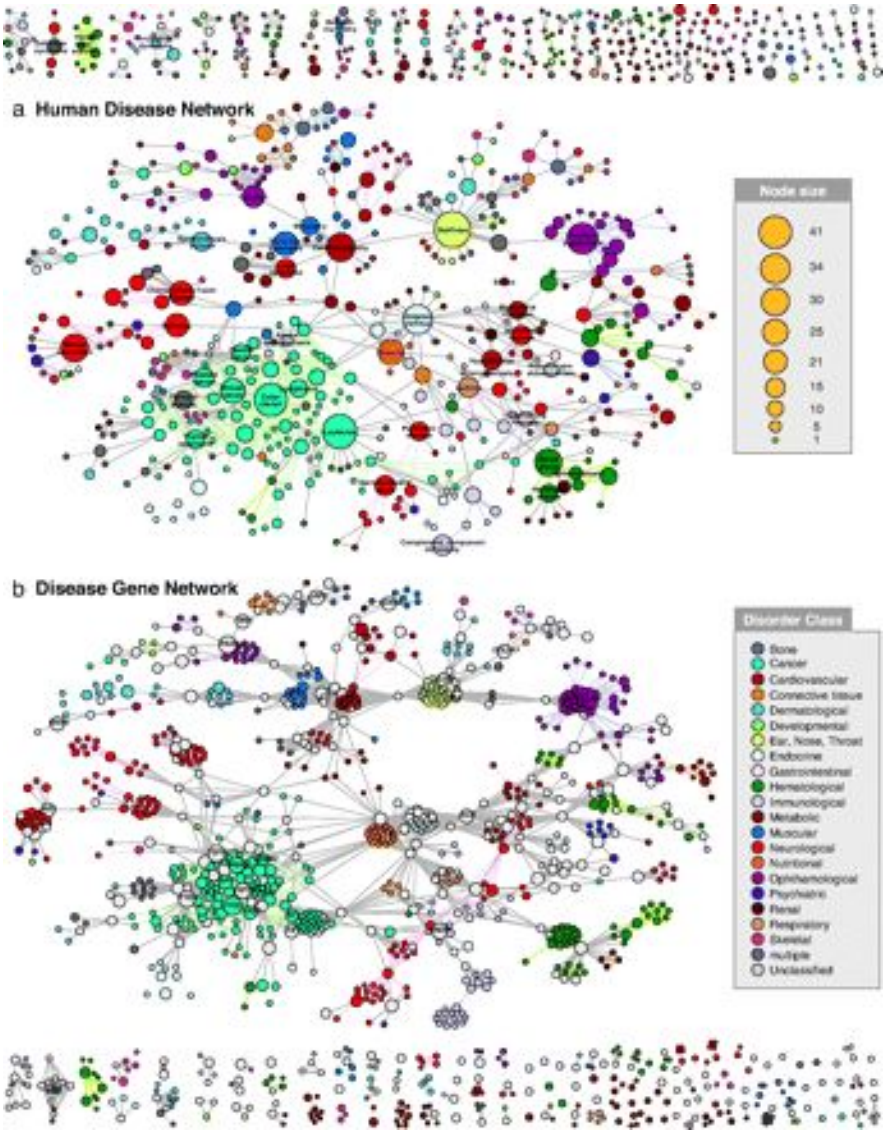


Figure 5: Models of a human disease network and disease gene network.
Adapted from (Goh, K.-I. The Human Disease Network. Proceedings of the National Academy of Sciences of the United States of America, 104(21)).

Disease networks also have the tendency to form communities or neighborhoods, which are groups of nodes which share certain characteristics, in the network (Barabasi 2013). One example of this clustering pattern is a staff network containing groups of teachers in the same department. This structuring into different modules of interconnected nodes is known as modularity, and it is one way to measure how structured a network is. These neighborhoods often have the characteristic that

their nodes have more links to nodes inside their community than they do to nodes outside the community. For instance, documents on the Internet tend to have more URLs to sites with information about their subject matter because that is what their readers will most likely be interested in (Barabasi 2013). This characteristic defines disease networks so that "proteins that are involved with the same disease show a high propensity to interact with each other" (Barabasi 2011). A topological module represents a locally dense neighborhood inside a network, while a functional module brings together nodes of similar or related biological function in the same neighborhood.

Using these concepts, scientists have created the "disease module", a group of interconnected disease network components that together contribute to a certain function in the body, which will cause a disease when it is disrupted (Barabasi 2011). For example, the disease module for Crohn's Disease contains all of the genes connected with the spread of Crohn's Disease, so if one of them stops functioning, Crohn's Disease could appear and spread to other genes in the network. Many disease modules can also overlap due to their genes playing a role in more than one disease. This creates a property called comorbidity, or habit of one disease to cause or be accompanied by another (Barabasi 2013). This in turn has led to the creation of the diseasome, or "disease maps whose nodes are diseases and whose links represent various molecular relationships between the disease-associated cellular components" (Barabasi 2013), to accompany the interactome. In general, a disease module is made up of the essential genes in the center and nonessential disease genes on the periphery. This way, the disease will hit the nonessential genes first before spreading to the essential genes which the body can't afford to lose. The interconnectedness among disease genes enables algorithms that use the probability of having a connection among disease module genes to determine what new genes should be added to the module.

2.2.2 Types of Biological Networks

In order to build disease modules, it is necessary to understand and observe the properties of different biological networks and determine which of their components and functions are connected to the spread of diseases. Every biological network represents the complete set of one particular type of interaction inside a particular organism, in this case humans. A few of the specific types of networks include protein-protein interaction networks, metabolic networks, regulatory networks,

and co-expression networks. These networks provide information about about interactions between two proteins, between molecules required for cellular metabolism, between molecules involved in sending regulatory signals to genes, and between genes with the same expression patterns, respectively. Observing these networks allows us to see which of their functions cause disease, allowing us to build disease networks and disease modules within them.

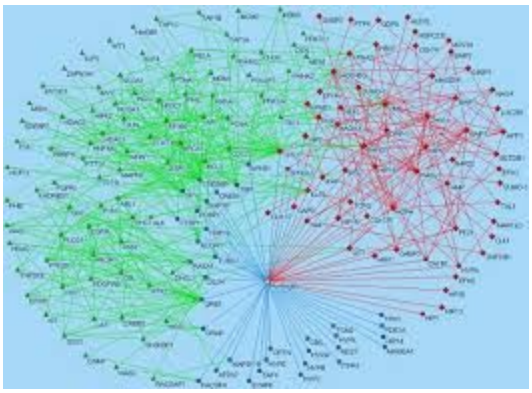


Figure 6: Model of a PPI network.

Retrieved from

https://www.mdc-berlin.de/10221541/de/research/research_teams/proteomics_and_molecular_mechanisms_of_neurodegenerative_diseases/research?cntx=40550219

Protein-protein interactions (PPI) like the one in Figure 6 are necessary for conducting many processes inside of cells, and refer to specific contacts that occur between two proteins. These interactions form a protein-protein interaction network (PPIN) in which the nodes represent proteins and the links represent undirected contacts between them. These contacts are "specific, occur between defined binding regions in the protein", and "have a particular biological meaning" (European Bioinformatics Institute 2017). For example, one protein can carry another protein from one region of a cell to another region where it is needed. The binding region refers to the section of the protein that has contact with another, which in many cases is specifically designed for that purpose. Protein-protein interactions can be either stable, meaning that they interact for a long time, or transient, meaning that they only interact briefly to prepare the protein for future operations (European Bioinformatics Institute 2017). Like most biological networks, PPINs are scale-free, so they have many proteins with only a few interactions and a small amount of hubs with many

interactions. They also tend to have a "small-world effect", which means that most nodes in the network are very closely linked, with only a few links separating them, allowing for greater speed in cellular processes. (European Bioinformatics Institute 2017). PPINs also tend to cluster proteins into neighborhoods based on what functions they perform, such as sending signals, transportation, or building compounds. An error in a protein-protein interaction or the removal of a protein from the network can be a catalyst for a disease, which means that PPINs and disease networks often overlap. However, many nodes and links in PPINs remain undiscovered, while for others, "only the existence of an interaction between two proteins is known, but the interaction type...remains unknown" (Zitnik 2016). In order to make an accurate prediction about the spread of diseases in the network, it is necessary to have a relatively good understanding of what interactions exist and how they occur.

Metabolic networks comprise all of the operations necessary for a cell's metabolism, or conversion of food into building blocks for the cell. In this case, the networks are "directed networks where each node represents a metabolite (a molecule) and an edge represents a metabolic reaction" (Zitnik 2016). For example, several metabolic processes are required to turn the sugars produced by photosynthesis into usable proteins and other cellular components. A metabolic pathway is "a connected sub-network of the metabolic network either representing specific processes or defined by functional boundaries" (Zitnik 2016). Therefore, the metabolic pathways divide the network into different neighborhoods based on their functionality, which generally means the transformation of a particular molecule into another. Like PPINs, metabolic networks are both scale-free, meaning that there are a few heavily connected metabolites and many loosely connected metabolites, and small-world, meaning that there are only a few links separating each node. Metabolic networks can also overlap with disease networks because any problems in the metabolic processes, like a missing metabolite or a failed transformation, can also lead to diseases. However, like PPINs, metabolic networks are missing some nodes and links, so it may be more difficult to track the progress of diseases until they are discovered.

Regulatory networks are made up of molecules, like proteins and RNA, that control the expression of genes in a genome. Gene regulation is usually "the response of a cell to an external stimulus" (Zitnik 2016). In this case, the nodes in the network are the genes expressed and their

regulators and the links (directed) are the actions of the regulatory molecules on them. For example, a protein called a transcription factor will "control gene expression by directly interacting with regulatory genomic DNA sequences that are usually located in or around their target genes" (MacNeil 2011). The main reason that these molecules are needed to regulate genes is so that organisms have the correct responses to certain situations. For example, "specific transcriptional programs are required for proper developmental patterning and to ensure appropriate responses to changing environmental conditions and stresses" (MacNeil 2011). However, these transcription factors and other molecules do not work alone and require a vast network of interactions to make all genes perform their functions when they need to. This is why "the TFs that control the expression of a gene often act together with other TFs" and "the TFs themselves are extensively regulated" (MacNeil 2011). Just as in other biological networks, GRNs contain many smaller modules based on a specific genetic function. In this case, "two types of modules can be identified: "gene modules" which are defined as sets of genes bound by similar TFs, and "TF modules" which are sets of TFs that share similar target genes" (MacNeil 2011). GRNs also share the properties of being small-world (only a few links between nodes) and scale-free (only a few links for normal nodes and many links for a small number of hub nodes). This means that "Biological systems are overall highly robust because most genes or gene products can be removed without compromising viability" (MacNeil 2011). The interconnectedness of transcription factors and other gene regulation molecules means that many genetic disorders can be traced to problems with GRNs, like a missing molecule or a failed transcription. However, as with other biological networks, the incompleteness of GRNs makes it more difficult to determine how diseases will act.

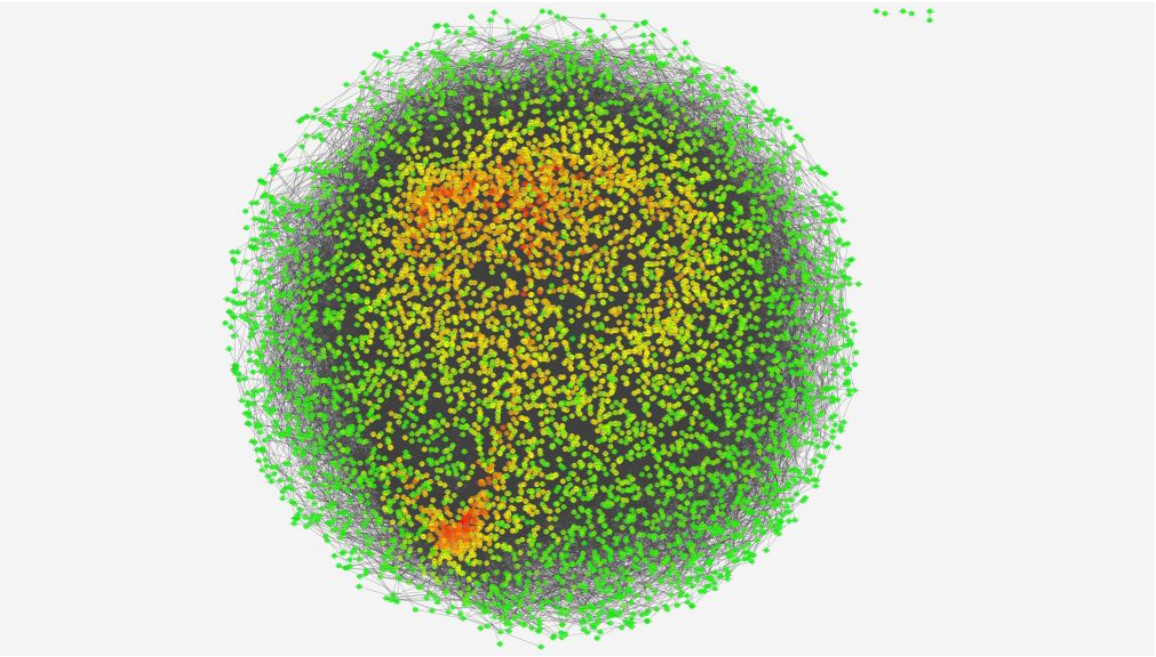


Figure 7: Model of a co-expression network.

Retrieved from https://en.wikipedia.org/wiki/Gene_co-expression_network

Co-expression networks, like the one in Figure 7 above, are made up of genes that have co-expression, or "the correlations between the expression of two genes knowing the expressions of all the other genes" (Villa-Vialaneix 2013). This means that co-expression represents a relationship between two genes so that when one is expressed in the body, the other is generally expressed also. For example, when examining eQTLs (genes that influence the expression levels of others), "a co-expression network is built from the first 272 gene expressions, and the structure of this network is highlighted, in terms of nodes of particular importance (hubs for instance), and in terms of decomposition into "communities" or "modules"" (Villa-Vialaneix 2013). In this case, the nodes in the network are the genes expressed and the links (undirected) signal that the two genes are expressed simultaneously. The correlation between the two genes can either be total or partial, meaning that they are co-expressed only some of the time. Co-expression networks share several similarities with other biological networks, including the tendency of nodes to cluster into specific neighborhoods. For instance, in the eQTL experiment, "seven clusters were identified that contained from 28 to 58 genes" (Villa-Vialaneix 2013), in this case relating to certain phenotypes. Co-expression networks also tend to be small-world (only a few links separate each node) and

scale-free (each network contains a few highly-connected hubs and many loosely connected regular nodes). In the eQTL experiment, "the network contained 21 hubs having a degree larger than 26", all of which were viewed as very important nodes. Co-expression networks can provide valuable information as to which genes are closely linked based on how they are expressed, which can be very useful when one of the genes has a problem. However, as with other biological networks, co-expression networks are limited by a lack of information, including "a major lack of annotation of the genomes, and the fact that most associated literature is devoted mainly to only a few mammalian species" (Villa-Vialaneix 2013).

2.2.3 Advances in Network Medicine

By studying the properties of biological networks to find disease modules, scientists have already made several advances in the field of network medicine. These include isolating the causes of disease, providing better targeting for drug development, and making a map of how a disease will spread throughout the body. Currently, the advances in network medicine fall mainly under network pharmacology and disease classification. Network pharmacology is the process of designing new drugs to treat diseases by using biological networks. Disease classification is the process of grouping diseases into certain categories in order to determine how they are connected so they can be dealt with simultaneously. Thanks to network medicine, these two fields of study will both play a larger role in the study of diseases in days to come.

In recent years, scientists have made several advances in network pharmacology, using biological networks more and more in the treatment of diseases. The first step of this process is to understand why a disease occurs by determining what is going wrong in each cell. Thanks to network medicine, we now know that "this dysfunction is limited to the disease module, which means that one can reduce the search for therapeutic agents to those that induce detectable changes in module activity" (Barabasi 2011). This is useful to pharmacists and biologists because it allows them to focus on drugs that specifically affect genes and other molecules within the disease module. Another discovery is that "a drug might have more than one binding partner such that its efficacy is determined by its multiple interactions, leading to unwanted side effects" (Barabasi 2011). This means that by using the processes that form the links in biological networks, scientists can determine

how drugs affect not just the molecules they are targeting, but also other molecules throughout the network. Therefore, scientists will know in advance, through this *in vivo* testing, if the drugs have any unintended side effects, via their biological connections, on other parts of the network. According to researchers, "the promise of network-based approaches in drug discovery is best illustrated in the area of bacterial and human metabolism" (Barabasi 2011). This research uses metabolic networks to predict "flux" changes that occur due to enzymes in bacteria when they are exposed to a particular drug. It has "led to the identification and testing of potential new antibacterial agents and the complex system-based responses that they produce (Barabasi 2011). Scientists are also experimenting with drugs that target multiple nodes in the network in order to bypass network side effects and just take care of all of the affected nodes, unlike so-called "single-target" drugs. These studies have led to "combinatorial therapies for AIDS, cancer, and depression" (Barabasi 2011) and "potentially safer multi-target combinations for inflammatory conditions and to the optimization of anti-cancer drug combinations" (Barabasi 2011). If advances in network medicine continue at this rate, then many more discoveries like these may become possible throughout network pharmacology.

Scientists have had some success in classifying diseases into certain groups, but there are some fundamental problems with their methodology. It reflects "a lack of sensitivity in identifying preclinical disease and a lack of specificity in defining disease unequivocally". This means that disease classifications often fail to identify characteristics of diseases at different stages and just lump them together. Classification methodology also tends to "neglect the interconnected nature of many diseases" (Barabasi 2011). As a counterpoint, some researchers have developed an alternative, network-based approach to disease classification. This approach starts with "the primary disease-causing gene, which contains a mutation" (Barabasi 2011), and then examines the other genes it interacts with, as well as the environmental factors that affect the disease. These factors combine to "give rise to clinical phenotypes that are highly individual" (Barabasi 2011), providing a more specific classification for each disease. However, network-based classification is still difficult to implement since "many of the factors affecting the disease module remain unknown or poorly mapped" (Barabasi 2011). Nevertheless, as network medicine becomes more advanced, our ability to model and classify diseases by network will improve.

CHAPTER 3: METHODOLOGY

The experimental work of this project begins by testing the DIAMOnD algorithm on a small number of diseases (seed genes are provided by the creators) and seeing how it performs, first by using a protein-protein network and then by using other networks. The next component is to build a new algorithm by changing the probability distribution used to find new genes in order to improve performance and running the same tests on the new algorithm. The objectives of the project are stated as follows:

Mission Statement:

The goal of this project is to investigate the DIAMOnD algorithm and work to improve its performance in terms of finding the correct genes to go into disease modules. For the first part, I will test the original DIAMOnD algorithm on sets of seed genes and interactions to determine how many of correct genes it finds for the module and then test it again using different networks. For the second part, I will modify the probability distribution used in the algorithm and test the new algorithm on sets of seed genes and interactions to determine how many of the correct genes it finds for the module. A series of graphs and charts will be made to display the performance of the algorithms for each data set.

Objectives:

1. Run initial tests on 5 diseases using the DIAMOnD algorithm, using sets of seed genes provided by the creators and an initial biological network, and see how they perform in terms of correctness.
2. Run tests on these diseases again using the DIAMOnD algorithm, but this time use different networks with the same seed genes.
3. Create a new algorithm by changing the probability distribution used in the DIAMOnD algorithm to improve the performance.
4. Run tests on 5 diseases using the new algorithm, using the same seed genes and networks, and see how its performance compares to that of DIAMOnD in terms of correctness.

3.1 The DIAMOnD Algorithm

The DIAMOnD algorithm uses advanced statistical methods to determine new genes and proteins to be added to a disease module. This involves taking a series of "seed genes", or genes that are known to have a significant connection to a particular disease and therefore should be in the disease module, and using their connections within a biological network (in this case a protein-protein network) to determine the next genes to be added to the module. The original creators of the algorithm have provided an implementation in Python that can be used to test it out (DIAMOnD.py). However, in order to work, the algorithm needs certain information as input, such as the current network and the list of seed genes. It also needs a way to analyze the genes that are added to the network and determine which ones are "correct", meaning that they are believed to have a strong connection to the disease represented by this module.

3.1.1 How DIAMOnD Works

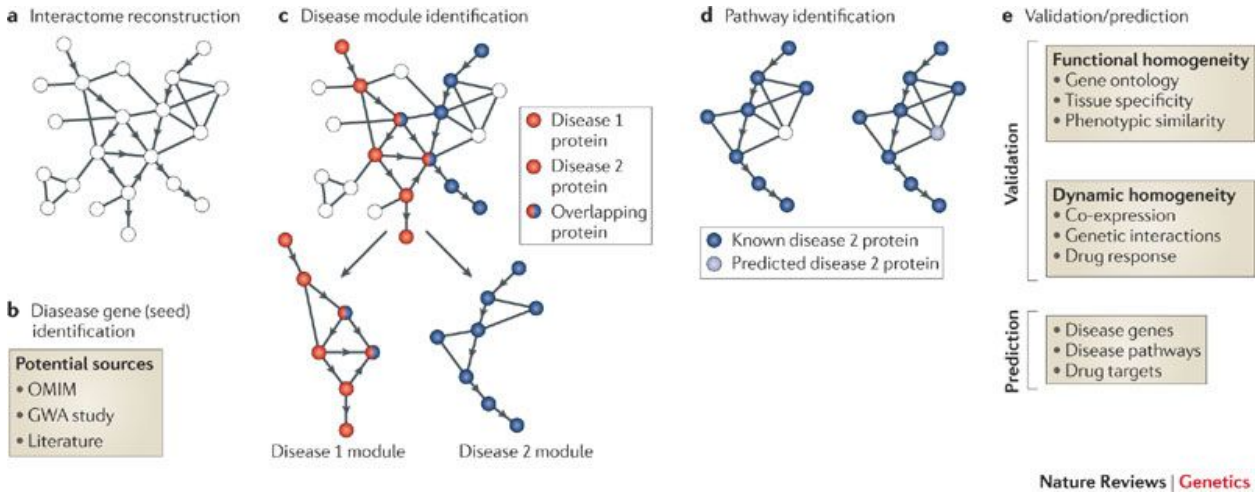


Figure 8: Process of finding disease modules.

Adapted from (Barabasi, A.-L. L., Joseph; Gulbahce, Natali. (2011). Network Medicine: A Network-Based Approach to Human Disease. Nature Reviews Genetics, 12(1)).

Most disease network detection algorithms try to pinpoint the causes of diseases by finding the genes related to a disease and then using the network to find more, since genes with similar functions tend to cluster in the same neighborhood within the network. This is demonstrated in

Figure 8 above. However, data scientists have had difficulty in determining the exact connectivity patterns of disease modules, with some methods of detection having more success than others. The creators of DIAMOnD analyzed several communities inside biological networks to determine whether or not they were significantly enriched with genes that are often associated with diseases, but discovered that they generally were not. The creators also found that disease modules had a low modularity, meaning that "while topological communities may often represent meaningful functional modules, they are not able to capture disease modules" (Ghiassian 2015), since disease modules are not as clustered as other modules can be. However, they did discover that "disease associated proteins do not reside within locally dense networks and instead identify connectivity significance as the most predictive quality" (Ghiassian 2015). Therefore, instead of evaluating the number of connections that each gene has with seed genes in the network, the algorithm focuses on the statistical significance of these connections.

To determine the significance of seed connections, the algorithm's creators calculated the p-value, or probability that a gene with k connections to genes in the network has k_s connections to seed genes specifically, for every potential gene in the network. They then summed these p-values to form the connectivity p-value, or probability that a gene with k connections to genes in the network has k_s or greater connections to seed genes, for every potential gene. To determine the significance of these values, the creators of the algorithm performed a Kolmogorov-Smirnov test, which "tries to determine if two datasets differ significantly" (Kirkman 1996). According to their results, "the connectivity p-values within the sets of known disease modules are very significantly shifted towards smaller values when compared to the distributions expected for randomly scattered proteins" (Ghiassian 2015). The Kolmogorov-Smirnov results produced a very small p-value, which "reports if the numbers differ significantly" (Kirkman 1996), meaning that there is a significant difference between the random distribution and the disease modules. Therefore, the creators determined that disease genes are more likely to have a low connectivity p-value because it is less likely to be obtained by chance and is therefore more significant. This unprecedented conclusion forms the crux of the DIAMOnD algorithm.

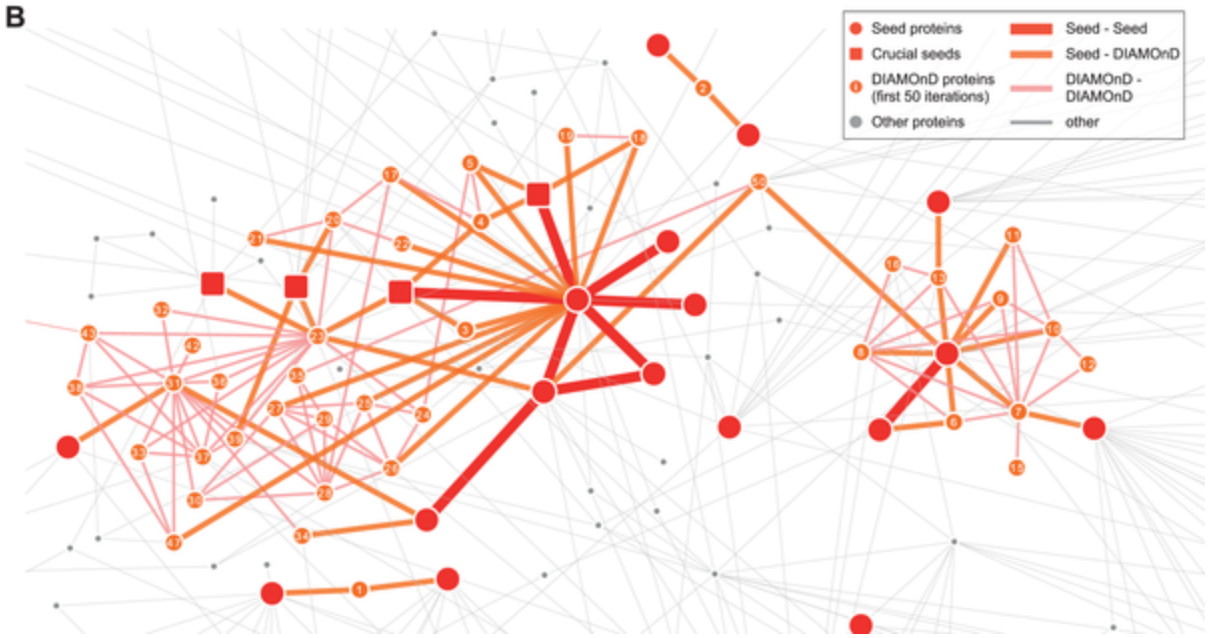


Figure 9: Progress of the DIAMOnD algorithm.

Adapted from (Ghiassian, S. D. M., Jorg; Barabasi, Albert-Laszlo. (2015). A DIseAse MOdule Detection (DIAMOnD) Algorithm Derived from a Systemic Analysis of Disease Proteins in the Human Interactome. PLoS Computational Biology, 11(4)).

The first steps taken by the DIAMOnD algorithm are to read in a set of input arguments, including a file containing the links in the biological network being used, a file containing the list of seed genes, and a number specifying the number of genes to be added to the disease module. Additional options include giving genes a certain weight in the probability distribution, which can impact the results in some cases, and providing a name for the output file where the new genes are listed (otherwise it defaults). Once all of the inputs have been read, the algorithm determines which seed genes should be included in the probability distribution by determining whether they have any connections inside the input network, and ignoring them entirely if they don't. The algorithm then executes its main loop, which determines the next gene to be added to the disease module by finding the gene in the network with the lowest connectivity p-value, making it the most significantly connected gene. The algorithm then adds this gene to the set of seed genes, as well as a list of the new disease genes, and then moves on to the next iteration, using the revised set of seed genes to

find the next gene in the module. This is repeated until the algorithm hits the maximum number of iterations specified in the input, and then the algorithm finishes by outputting the list of new genes.

DIAMOnD calculates the connectivity p-value for every gene in the network by calling a function that determines the degrees and neighbors of every gene. In addition, it reduces time by taking proteins with the same degree and eliminating the one that will produce a larger p-value (which can be determined from the probability distribution used). The p-value for a gene with n connections and x seed connections is determined using a hypergeometric distribution. The hypergeometric experiment is a statistical experiment in which a sample size of n items is selected from a total of N items without replacement (Ghiassian 2015). In this experiment, k of the N items are considered successes and N-k items are considered failures, based on a true/false random variable. The probability that the experiment will have x successes is $p={}_kC_x * {}_{(N-k)}C_{(n-x)} / {}_NC_n$ (C=choose function). This distribution fits the DIAMOnD p-values perfectly because we start with N genes in the network, then for each gene, we select a sample size of n genes that represent the genes connected to that gene in the network. In this case, our random variable considers a connection to a seed gene a success and a normal connection a failure, so we can use the hypergeometric distribution to determine the probability that a gene has x links to seed genes if it has n connections and there are N genes and k seed genes in the network. The algorithm further reduces time by storing all p-values in an array and simply using them if the p-value with the specified x, k, n, and N values has already been discovered.

3.1.2 Analysis of DIAMOnD

One of the biggest problems posed to scientists studying network medicine is how to determine whether a particular gene actually belongs to a certain cluster or not. This is the main issue faced after DIAMOnD is run, for there must be a way to determine whether the new genes are actually part of the disease module or not. Fortunately, thanks to the modularity of many biological networks, disease modules tend to be made up of genes that perform similar functions, and are therefore associated with similar biological terms. Therefore, the best way to validate the genes obtained by DIAMOnD as being part of the disease module is to run a gene ontology test on them. As mentioned in the background, gene ontology is the process of determining which terms are

associated with particular genes. To validate the DIAMOnD genes, the original creators found the GO terms associated with the seed genes and then looked through the GO terms of the DIAMOnD genes to see if they share any of the same terms as the seed genes. If a DIAMOnD gene is associated with at least one of the GO terms, then it is considered a "true positive", meaning that it actually belongs in the disease module.

Based on the creator's original experiments, it appears that the DIAMOnD algorithm is very successful at finding new genes to add to biological networks. Based on a graph they made comparing the number of true positives to the number of DIAMOnD iterations, it appears that there is a sharp increase in the number of true positives obtained by DIAMOnD compared to those obtained by a random search. However, they also found that the number of true positives obtained levels off after about 200 iterations, so there is a limit to how many disease genes DIAMOnD can find. The creators performed a number of additional tests with DIAMOnD, such as limiting certain GO terms, removing some of the seed genes from the network to see if DIAMOnD could find them and adding weights to links connected to seed genes. However, this project was unable to replicate some of these tests due to time and manpower constraints.

3.2 DIAMOnD Testing

The first step in improving the correctness of DIAMOnD was to replicate the results of the original experiment in order to confirm that the algorithm actually worked. This also provided an opportunity to see how the algorithm performs depending on the initial information given, the seed genes and the the network interactions. Therefore, I collected sets of data from my advisors, the original experiment, and certain databases in order to test the algorithm. I also ran some additional tests by changing some of the parameters of the experiment. The algorithm runs fairly quickly, and appears to perform just as well as described.

3.2.1 Testing Methods

To start out, I chose a set of diseases to serve as test modules: Crohn's Disease, Behcet Disease, Leukemia B-cell, Sarcoma, and Vasculitis. Next, I ran a DIAMOnD test on each of these diseases using the Python code (DIAMOnD.py) provided by the original experiment. Each test used

200 iterations, since the cutoff for finding more true positives in the original experiment was around this point. The network used for the initial tests was a PPI network provided by my advisors. The seed genes for each disease were obtained from OMIM and GWAS and used in the original experiment, so by using them, I ensured that I would get similar results. I ran each test using a weight of 1 for each connection and saved each set of new disease genes to a different output file. Therefore, after performing these tests, each disease module had 200 potential new genes.

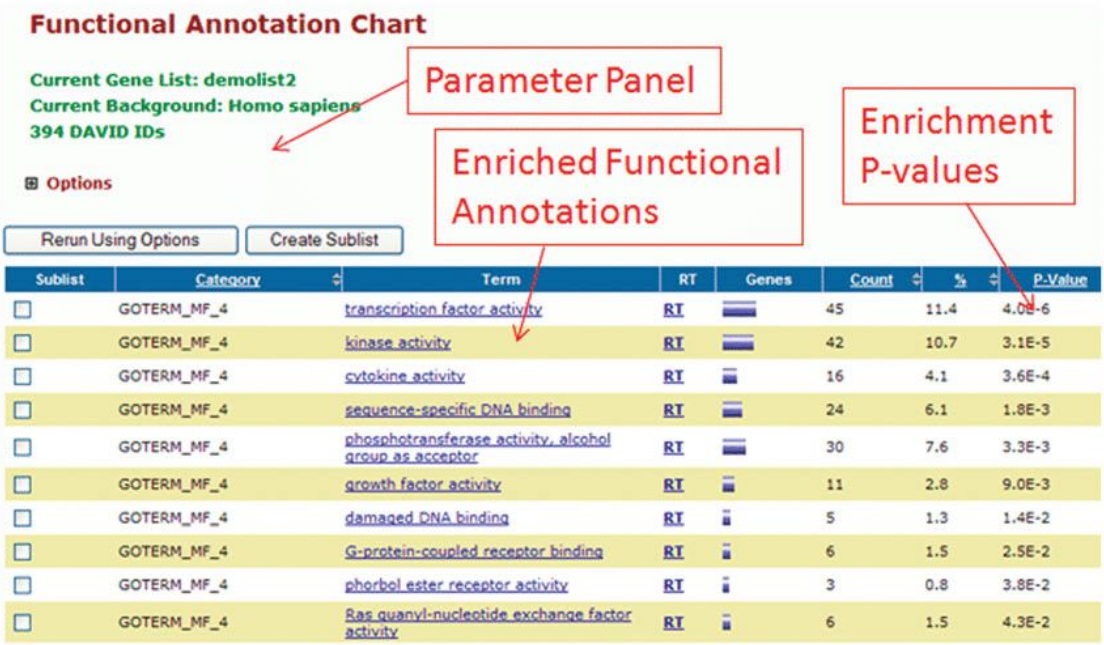
Once the tests on the PPI network were complete, I ran an additional set of tests using a different network, this time a co-expression network (also provided by my advisors). The purpose of this was to determine whether or not DIAMOnD produces the same level of performance for different types of biological networks. The set of seed genes for each of the five diseases was the same, but running them on a different network produced a different set of DIAMOnD genes for each disease. This process can be repeated for any type of biological network, although the best results come from networks that are more complete. Once again, I used a cutoff point of 200 genes and saved each test to a different outfile.

The final step in testing DIAMOnD was to compare it to a random sampling, so as to confirm that it performs better than the norm. In order to do this, I designed additional program called random_walk.py to select a random set of genes from the network used for finding DIAMOnD genes. I did this by simply making a loop in which a random gene is selected from the network to be part of the disease module, and then removed from the network so it is not selected again. This way, the random selection can be iterated as many times as DIAMOnD, so as to provide a fitting comparison (200 times in this case). The genes for DIAMOnD and the random search were validated the same way for every disease and network used.

3.2.2 Validation Methods

As in the original experiment, this experiment validated the DIAMOnD genes by running a gene ontology test on them. Therefore, I conducted a search in DAVID to find the GO terms associated with the seed genes for each disease, as shown in Figure 10. I did this by plugging the list of seed genes into DIAMOnD (in this case using the regular gene name) and running the test on them. I then looked through each of the GO lists for cellular components, biological processes, and

molecular functions to find all of the GO terms that were strongly associated with each disease. In this case, a term is considered strongly associated with a set of genes if it has a Bonferroni correction value (an additional test designed to reduce insignificant data from the GO terms) of less than 0.05 (Ghiassian 2015).



Sublist	Category	Term	RT	Genes	Count	%	P-Value
☐	GOTERM_MF_4	transcription factor activity	RT		45	11.4	4.0E-6
☐	GOTERM_MF_4	kinase activity	RT		42	10.7	3.1E-5
☐	GOTERM_MF_4	cytokine activity	RT		16	4.1	3.6E-4
☐	GOTERM_MF_4	sequence-specific DNA binding	RT		24	6.1	1.8E-3
☐	GOTERM_MF_4	phosphotransferase activity, alcohol group as acceptor	RT		30	7.6	3.3E-3
☐	GOTERM_MF_4	growth factor activity	RT		11	2.8	9.0E-3
☐	GOTERM_MF_4	damaged DNA binding	RT		5	1.3	1.4E-2
☐	GOTERM_MF_4	G-protein-coupled receptor binding	RT		6	1.5	2.5E-2
☐	GOTERM_MF_4	phorbol ester receptor activity	RT		3	0.8	3.8E-2
☐	GOTERM_MF_4	Ras guanyl-nucleotide exchange factor activity	RT		6	1.5	4.3E-2

Figure 10: Set of terms obtained by DAVID functional annotation tool. Retrieved from https://david.ncifcrf.gov/home.jsp

In order to determine whether the DIAMOnD genes had any of these terms, I downloaded a complete list of genes in the human genome and their GO terms from the Gene Ontology Consortium. I also designed additional program in Python called parse_annotations.py to parse the list and determine if each DIAMOnD gene had any of the same GO terms as the seed genes (by taking the list of seed gene GO terms and the list of DIAMOnD genes as inputs). The program finds a list of all the DIAMOnD genes containing at least one of the seed GO terms, which can be used to determine the number of true positives among the DIAMOnD genes. It outputs the number of true positives obtained by each iteration of the algorithm by first starting with one DIAMOnD gene and determining whether it is a true positive. It then adds the next DIAMOnD gene to the list and determines the number of true positives between the two genes, and continues adding more until all the iterations are complete. This program works for both the DIAMOnD algorithm and the

random selection. Thanks to the data analysis from the original experiment, a reasonable comparison exists for how many of the DIAMOnD genes should be true positives, so as to verify that the experiment is working properly. Once all of the new DIAMOnD genes were validated, it became possible to graph the number of true positives for each of the 5 diseases used.

3.3 Augmenting DIAMOnD

The next major part of improving the correctness of DIAMOnD was to find a way to upgrade the existing algorithm in order to increase its chances of finding the correct genes. In order to do this, I modified the probability distribution used to calculate the p-values for each of the network genes. The original DIAMOnD algorithm calculates the p-values by simply using the number of links each gene is connected to in the network and the total number of links to determine the probability that the gene is connected. However, the augmented algorithm challenges this assumption by considering whether that gene's links also exist in other biological networks. The hope was that including more biological information in the algorithm would provide a more realistic idea of how likely each gene is to be connected, since a link that appears in several networks is considered more likely to exist in a random network. Therefore, I gave such links a greater weight in the hypergeometric distribution and then normalized the distribution, hoping to provide a more realistic probability distribution for calculating the p-values.

3.3.1 The Augmented DIAMOnD Algorithm

The augmented DIAMOnD algorithm (augmented_DIAMOnD.py) works by using one biological network to obtain DIAMOnD genes, but supplementing it with additional networks used to indicate how rigid the biological links of those genes are. For instance, if one gene in a PPI network has four connections to other genes in the network, and twenty one genes total, then the probability that it is connected to any gene is four/twenty, or one fifth. However, in a co-expression network which contains that same gene, there may be connections to the first two genes but not the last two. Therefore, in the original PPI network, one might consider the original gene to have a greater chance (ex. eight/twenty) of being connected to the first two genes and a smaller chance (ex. two/twenty) of being connected to the last two genes. By taking this information into account, the

algorithm indicates how likely one gene is to be connected to another across all biological networks, not just the one being used to obtain the DIAMOnD genes.

For my augmented DIAMOnD algorithm, I took advantage of the algorithm's functionality for adding weights to certain links in the network. The original creators left an option for users to give additional weight to links connected to seed genes by specifying the weight in a command line argument. The algorithm then normalizes the hypergeometric distribution for the p-value by adding this weight to the values of N, n, k, and x for each gene connected to seed genes. It does this by adding a number of links equal to the additional weight to N, n, k, and x for each seed link. For example, if one were to specify a weight of 2, then the algorithm would add (2-1)=1 to N and k for every seed link and to n and x for every seed link connected to the current gene. For the augmented algorithm, I changed this scheme so that instead of adding additional weight for every seed link, the algorithm adds additional weight for every link that exists in an additional supplementary network of my choosing. After determining which links exist in both networks, I went through every gene as usual, but this time added additional weights to N, n, k, and x if the gene was connected to a link that existed in both networks. For instance, if I tested the algorithm using a PPI network and used a regulatory network as the supplementary network, and then specified a weight of 2, then the algorithm would add (2-1)=1 to N for every link that exists in both networks, to k if that link contains seeds, and to the n and x values for each gene in the original network depending on whether the link contains a seed.

For my testing, I used an augmented algorithm that obtains new DIAMOnD genes from the original network but includes a supplementary network and gives greater weight to links that appear in both networks. I also used a weight of 2 for links that exist in both networks and a link of 1 for genes that exist in just the first network. However, since the two networks are specified as parameters, one could use any network as the main network or the supplementary network. It also wouldn't be too difficult to add functionality to consider multiple networks in the algorithm and then add additional weight to links that exist in three or more networks. One could also experiment with the weighting scheme by giving even higher weights to genes that exist in multiple networks or lower ones to genes that exist in just the first network, although this may require altering the way the algorithm calculates the p-value, since it currently doesn't accept fractions.

3.3.2 Comparing the Algorithms

In order to compare the augmented DIAMOnD algorithm to the original DIAMOnD algorithm, I used the same tests that I used for the original DIAMOnD algorithm. I ran the new algorithm using the sets of seed genes for Crohn's disease, Behcet disease, leukemia, sarcoma, and vasculitis. For the first set of tests, I used the PPI network as the main network and the co-expression network as the supplementary network for comparing links. For the second set of tests, I reversed the process by using the co-expression network as the main network and the PPI network as the supplementary network for comparing links. Once again, I used a cutoff point of 200 iterations for each test and saved each of the results to a different outfile.

Once I tested the augmented DIAMOnD algorithm, I saw that it produced different results than the original DIAMOnD algorithm. I also verified that the new algorithm was in fact finding links that existed in both networks and adding the appropriate weights when computing the p-value by testing on a simple network of only a few genes. This way, I could calculate for myself what the values for N, k, etc. were supposed to be and verify that the algorithm was actually getting them. Therefore, as long as the new algorithm uses two networks that have some of the same links, it should produce different results than DIAMOnD. If not, then it will perform the same way as the original DIAMOnD algorithm. In order to verify the results of the new algorithm, I used the same gene ontology test that I used for the original DIAMOnD algorithm. Therefore, I ran each set of new disease genes through my GO parsing program to determine the number of true positives for each DIAMOnD test. I then compared them to the number of true positives obtained for the original DIAMOnD algorithm to determine if it indeed performs better.

CHAPTER 4: RESULTS AND ANALYSIS

In order to determine how successful each algorithm was, I made several graphs to show the validation of each of the new disease genes. In particular, I made a set of graphs to model the number of true positives found for each disease module compared to the number of iterations used for the algorithm. I also included a straight y=x line in the graph to represent the performance of the seed genes, since any seed genes are already in the disease module and are therefore already validated. I fit an additional line to the scatter plot representing the DIAMOnD gene data, so that one can compare the number of DIAMOnD true positives to the number of seed genes for the same number of iterations (which are all true positives). This comparison of the two lines demonstrates how successful DIAMOnD was in finding the correct genes for each disease module. In addition, I included a line representing random gene selection to prove that DIAMOnD performs better than the norm. I repeated all of these steps when graphing the progress of the augmented algorithm. I also made a table to compare the final numbers of true positives obtained by each algorithm.

4.1 DIAMOnD Results and Analysis

Primary Network	Disease	Algorithm	Final True Positive Count (200 iterations)
PPI	Behcet	DIAMOnD	160
		Random	88
	Crohn	DIAMOnD	137
		Random	37
	Leukemia	DIAMOnD	184
		Random	80

	Sarcoma	DIAMOnD	195
		Random	127
	Vasculitis	DIAMOnD	161
		Random	91
Co-expression	Behcet	DIAMOnD	116
		Random	82
	Crohn	DIAMOnD	88
		Random	31
	Leukemia	DIAMOnD	50
		Random	74
	Sarcoma	DIAMOnD	184
		Random	142
	Vasculitis	DIAMOnD	129
		Random	93

Figure 11: Table of final true positive values for DIAMOnD testing. Each test has a network (PPI or co-expression), disease (Behcet, Crohn, leukemia, sarcoma, or vasculitis), algorithm (DIAMOnD or random) and final count of correct "true positive" disease genes (0 to 200).

In order to analyze the results of the original DIAMOnD algorithm, I took the results of the gene ontology test performed by my parse_annotations.py program and used them to graph the number of true positives with a scatter plot. The program calculates the number of true positives for each iteration of the algorithm, so I graphed using number of iterations as the independent variable

and the number of true positives as the dependent variable. Finally, I found a best fit line for each of the scatter plots and used these to graph the performance of the algorithm. One graph was made for each of the diseases used (Behcet, Crohn, Leukemia, Sarcoma, and Vasculitis) for each of the biological networks used (PPI and co-expression) for a total of 10 graphs (see Figures 12-21). Each graph includes a scatter plot best fit line for seed, DIAMOnD, and random genes, based on the results I got from parse_annotations.py for all of them. I also made a table comparing the final true positive values for the DIAMOnD and random genes (see Figure 11).

Based on the graph and table results, it appears that all of the DIAMOnD tests were fairly successful in finding new genes for all of the disease modules. In almost all of the graphs, the best fit line for DIAMOnD is higher than the best fit line for random selection, indicating that DIAMOnD finds more correct disease genes than random selection for almost every iteration. The one exception is the test for leukemia with the co-expression network, which yielded results worse than those of the random selection (50 final true positives compared to 74). It is possible that the gene ontology test for this disease is not entirely accurate, or that the list of terms obtained from DAVID for the seed genes is incomplete. It also appears that in most cases, the best fit line for the DIAMOnD algorithm is fairly close to that of the best fit line for seed genes (100% correct). This means that for most of the tests, the majority of the genes that DIAMOnD obtains have a strong probability of belonging to the disease module of the disease tested. However, it is possible that the results would vary somewhat for a different validation test, such as PASCAL.

When comparing the results that DIAMOnD produces between the PPI network and the co-expression network, it appears that DIAMOnD performs better on the PPI network. For every disease, the algorithm yields a higher number of true positives for the PPI network than for the co-expression network for most of the graph iterations. The reasons for this are uncertain, but it could be that the PPI network is simply more complete than the co-expression network, since many links are still undiscovered. For the PPI network, the disease for which the algorithm performs best is sarcoma (195 true positives), and the disease for which the algorithm performs worst is Crohn's disease (137 true positives). For the co-expression network, the disease for which the algorithm performs best is sarcoma (184 true positives), and the disease for which the algorithm performs worst is leukemia (50 true positives). For the PPI network, the total number of true positives ends

up being between 150 to 200 out of 200 for most of the tests, which is an amazing performance. For the co-expression network, the number of validated genes ends up being somewhat lower, around 50 to 150 out of 200 for most of the tests. However, for both of the networks, the algorithm seems to be fairly successful at finding new disease genes.

There are a few major findings we can determine from the DIAMOnD tests. Based on the true positive graphs and table, it appears that DIAMOnD is in fact successful at disease module detection, since it performs better than the random distribution in most cases. It also seems that DIAMOnD has an exceptionally high performance for some diseases and networks, since some of the tests, like the ones for sarcoma, produced a best fit line that was close to 100% successful. DIAMOnD also performs better on the PPI network than the co-expression network. Based on the random selections for each of the diseases, it appears that DIAMOnD was successful in improving disease module detection for all of them, aside from the co-expression test for leukemia. Therefore, we can conclude that the DIAMOnD algorithm is the algorithm to beat in terms of disease module detection, which puts a lot of pressure on the augmented algorithm.

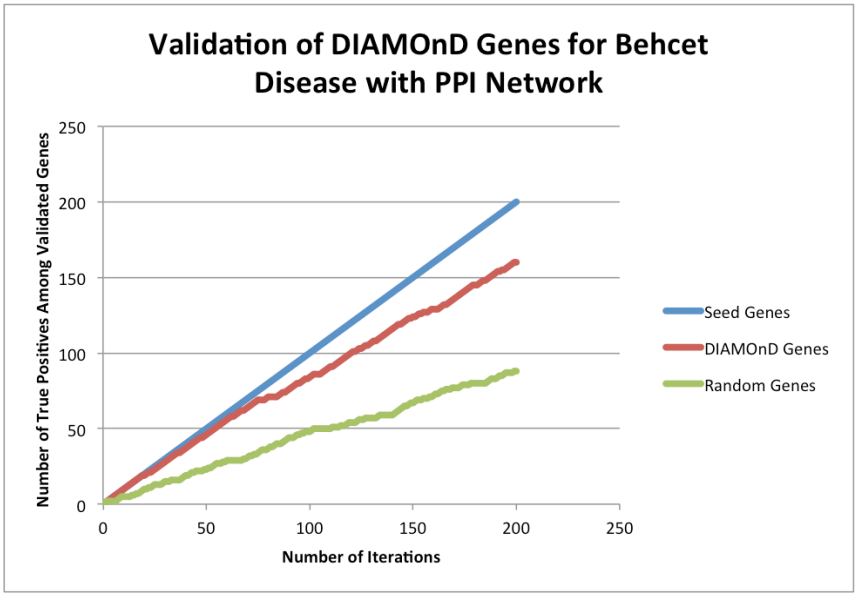


Figure 12: Validation of DIAMOnD genes for Behcet disease with PPI network. Each line (for seed genes, DIAMOnD genes, and random genes) graphs the number of algorithm iterations against the number of "true positive" Behcet disease genes obtained by comparing the GO terms of the seeds genes and algorithm genes.

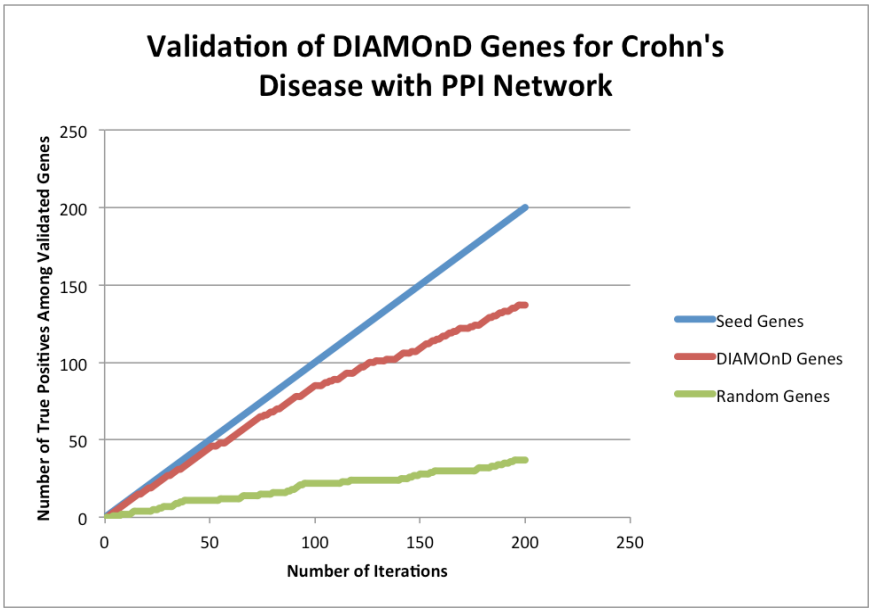


Figure 13: Validation of DIAMOnD genes for Crohn's disease with PPI network. Each line (for seed genes, DIAMOnD genes, and random genes) graphs the number of algorithm iterations against the number of "true positive" Crohn disease genes obtained by comparing the GO terms of the seeds genes and algorithm genes..

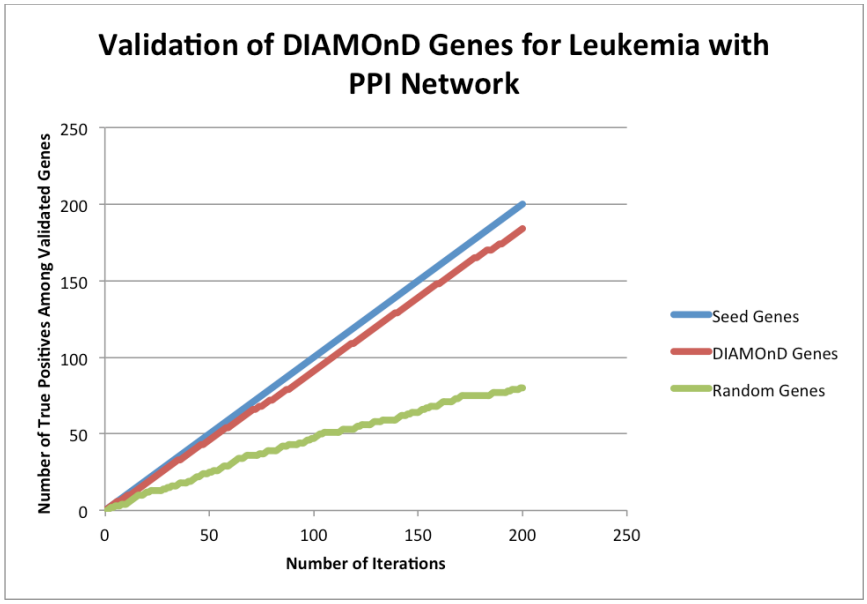


Figure 14: Validation of DIAMOnD genes for leukemia with PPI network. Each line (for seed genes, DIAMOnD genes, and random genes) graphs the number of algorithm iterations against the number of "true positive" leukemia disease genes obtained by comparing the GO terms of the seeds genes and algorithm genes.

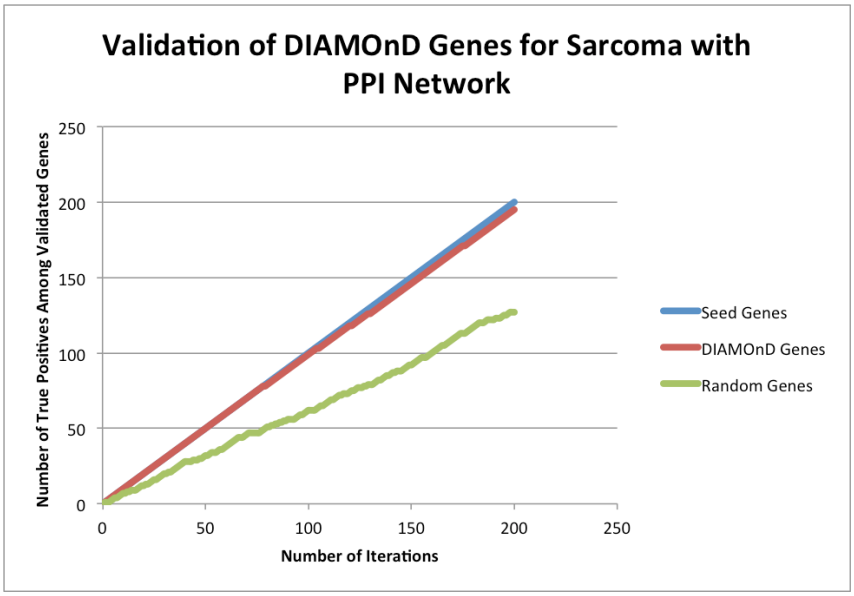


Figure 15: Validation of DIAMOnD genes for sarcoma with PPI network. Each line (for seed genes, DIAMOnD genes, and random genes) graphs the number of algorithm iterations against the number of "true positive" sarcoma disease genes obtained by comparing the GO terms of the seeds genes and algorithm genes.

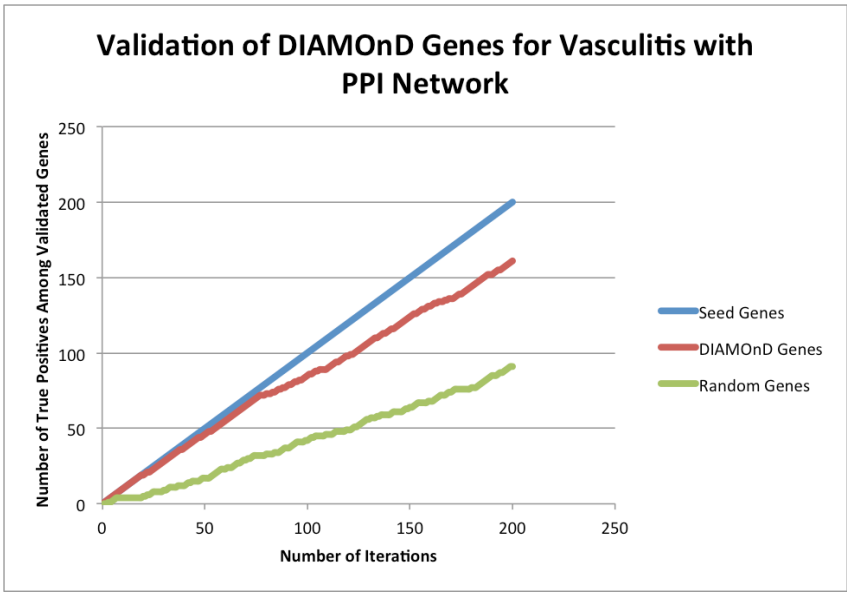


Figure 16: Validation of DIAMOnD genes for vasculitis with PPI network. Each line (for seed genes, DIAMOnD genes, and random genes) graphs the number of algorithm iterations against the number of "true positive" vasculitis disease genes obtained by comparing the GO terms of the seeds genes and algorithm genes.

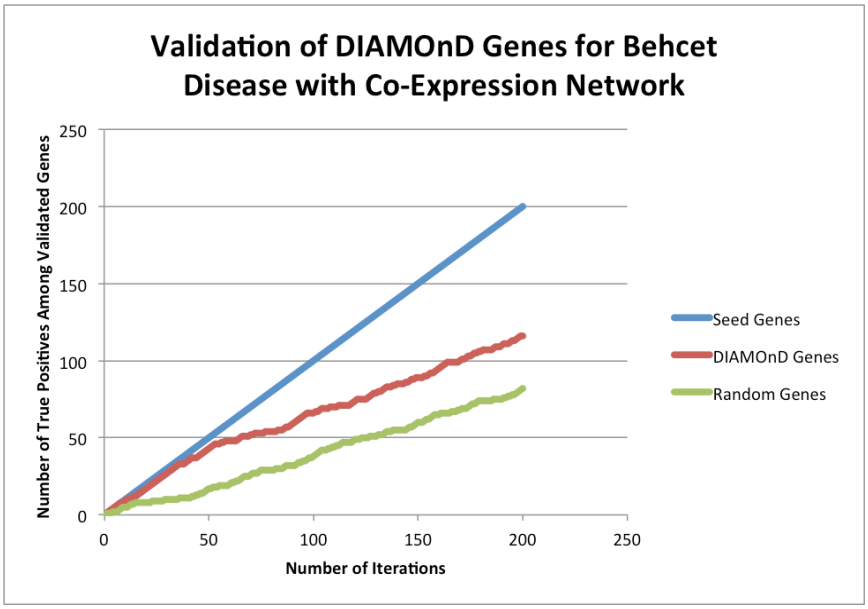


Figure 17: Validation of DIAMOnD genes for Behcet disease with co-expression network. Each line (for seed genes, DIAMOnD genes, and random genes) graphs the number of algorithm iterations against the number of "true positive" Behcet disease genes obtained by comparing the GO terms of the seeds genes and algorithm genes.

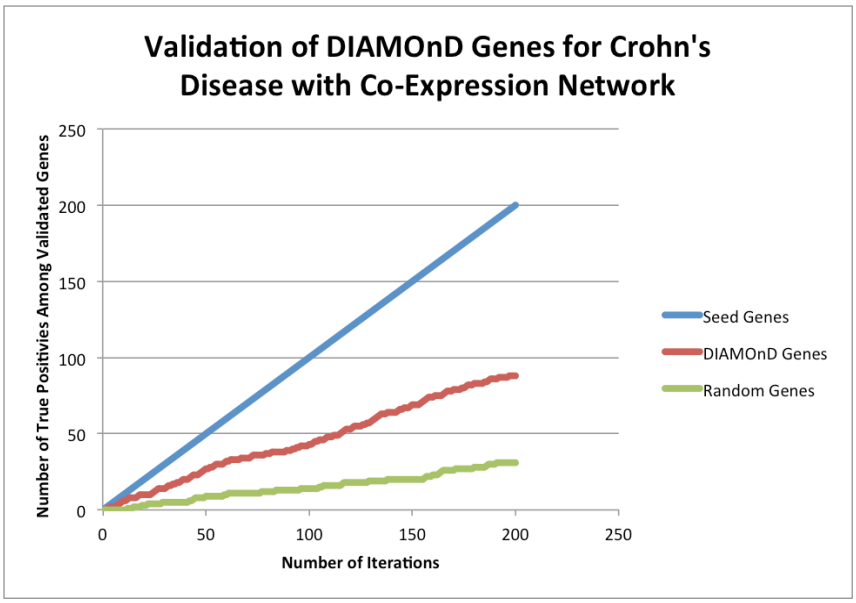


Figure 18: Validation of DIAMOnD genes for Crohn's disease with co-expression network. Each line (for seed genes, DIAMOnD genes, and random genes) graphs the number of algorithm iterations against the number of "true positive" Crohn disease genes obtained by comparing the GO terms of the seeds genes and algorithm genes.

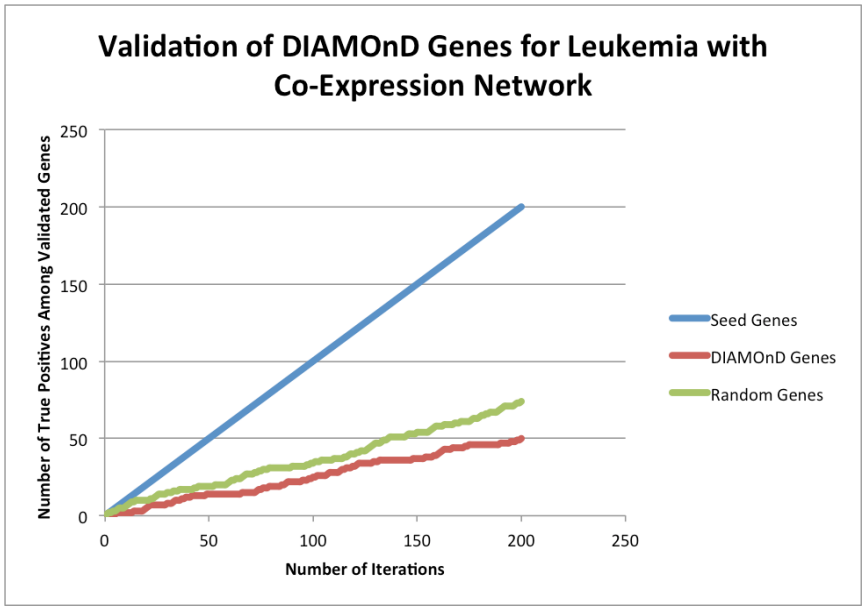


Figure 19: Validation of DIAMOnD genes for leukemia with co-expression network. Each line (for seed genes, DIAMOnD genes, and random genes) graphs the number of algorithm iterations against the number of "true positive" leukemia disease genes obtained by comparing the GO terms of the seeds genes and algorithm genes.

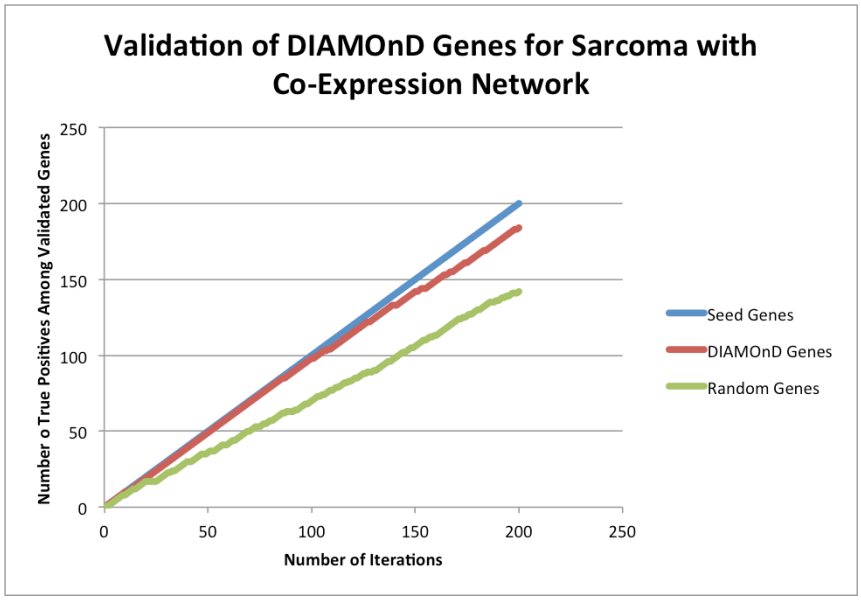


Figure 20: Validation of DIAMOnD genes for sarcoma with co-expression network. Each line (for seed genes, DIAMOnD genes, and random genes) graphs the number of algorithm iterations against the number of "true positive" sarcoma disease genes obtained by comparing the GO terms of the seeds genes and algorithm genes.

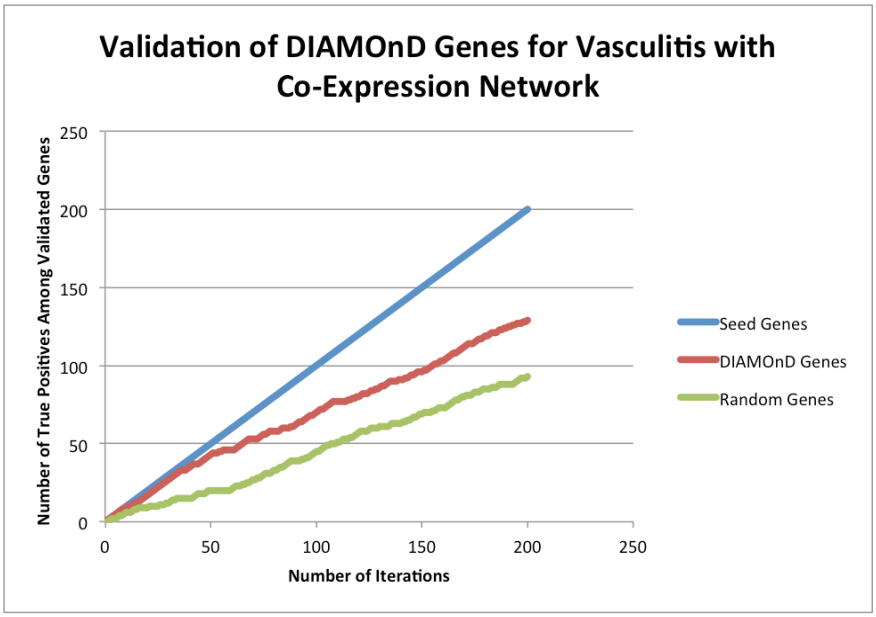


Figure 21: Validation of DIAMOnD genes for vasculitis with co-expression network. Each line (for seed genes, DIAMOnD genes, and random genes) graphs the number of algorithm iterations against the number of "true positive" vasculitis disease genes obtained by comparing the GO terms of the seeds genes and algorithm genes.

4.2 Augmented DIAMOnD Results and Analysis

Primary Network	Disease	Algorithm	Final True Positive Count (200 iterations)
PPI	Behcet	DIAMOnD	160
		Augmented DIAMOnD	155
		Random	88
	Crohn	DIAMOnD	137
		Augmented	155

		DIAMOnD	
		Random	37
	Leukemia	DIAMOnD	184
		Augmented DIAMOnD	182
		Random	80
	Sarcoma	DIAMOnD	195
		Augmented DIAMOnD	196
		Random	127
	Vasculitis	DIAMOnD	161
		Augmented DIAMOnD	155
		Random	91
Co-expression	Behcet	DIAMOnD	116
		Augmented DIAMOnD	132
		Random	82
	Crohn	DIAMOnD	88
		Augmented DIAMOnD	81

		Random	31
	Leukemia	DIAMOnD	50
		Augmented DIAMOnD	183
		Random	74
	Sarcoma	DIAMOnD	184
		Augmented DIAMOnD	188
		Random	142
	Vasculitis	DIAMOnD	129
		Augmented DIAMOnD	132
		Random	93

Figure 22: Table of final true positive values for augmented DIAMOnD testing. Each test has a network (PPI or co-expression), disease (Behcet, Crohn, leukemia, sarcoma, or vasculitis), algorithm (DIAMOnD, augmented DIAMOnD, or random) and final count of correct "true positive" disease genes (0 to 200).

In order to analyze the results from the augmented DIAMOnD algorithm, I used the same graph structure that was used for the regular DIAMOnD algorithm. I also used the same parse_annotations.py program to validate the augmented DIAMOnD genes, so I could graph the number of iterations against the number of true positives for the new algorithm, just as I did for the original. However, this time I included scatter plot best fit lines for both the original DIAMOnD algorithm and the augmented DIAMOnD algorithm, so I could compare the performance of the two. I kept the lines for seed genes and random selection also, just to see how the augmented

algorithm does compared to them. Once again, I performed a test for each of the 5 diseases with two different networks, for a total of 10 tests. For the first set of tests, I used the PPI network as my main network and used the co-expression network as a supplementary network to increase the weights. For the second set of tests, I used the co-expression network as my main network and used the PPI network as a supplementary network to increase the weights. I made an additional set of 10 graphs (see Figures 23-32) and another table showing the final numbers of true positives among the DIAMOnD, augmented DIAMOnD, and random genes (see Figure 22).

Based on the graph and table results, it appears that all of the augmented DIAMOnD tests were also fairly successful in finding new genes for all of the disease modules. All of the augmented DIAMOnD tests produced a higher best fit line than the random selection of genes. This includes the test for leukemia with the co-expression network, which produced a best fit line below the one for the random selection during the original algorithm tests. In addition, it appears that the new algorithm does outperform the original algorithm for some of the tests. There are several tests for which the augmented best fit line is higher than that of the original best fit line. It also appears that in most cases, the best fit line for the augmented algorithm is fairly close to the best fit line for the seed genes (100% correct). This means that the majority of the genes collected by the augmented DIAMOnD algorithm for each disease have a strong chance of belonging to the disease module for this disease, although more validation tests may be needed to verify this.

It appears that just as with the original DIAMOnD algorithm, the augmented algorithm performs better when using the PPI network as the main network than it does when using the co-expression network as the main network. For most of the diseases, the best fit line for the augmented algorithm is higher for the PPI network than it is for the co-expression network. The disease with the greatest number of true positives for the new algorithm with the PPI network was sarcoma (196 ture positives), while the one with the smallest number of true positives was a three-way tie between Behcet disease, Crohn's disease, and vasculitis (155 true positives). The disease with the greatest number of true positives for the new algorithm with the co-expression network was sarcoma (188 true positives), while the one with the smallest number of true positives was Crohn's disease (81 true positives). For both networks, the total number of true positives mostly ends up being between 100 and 200 genes out of 200, suggesting more parity between the networks

than with the original algorithm. Therefore, for both orderings of the main and supplementary networks, the algorithm appears to be fairly successful at finding disease genes.

It appears that in general, the augmented algorithm does produce a slightly higher number of true positives than the original algorithm does. Of the 10 tests with the augmented algorithm, 6 of them produce final true positive numbers that are higher than those of the original algorithm. In addition, in all of the tests, the best fit line for the augmented algorithm is either much higher than that of the original algorithm or runs along roughly the same points that it does. However, the co-expression network tests showed a greater improvement, in general, in the number of true positives discovered from the old algorithm to the new one than the PPI network did. The greatest change was in the number of true positives for leukemia when tested with the co-expression network, which rocketed upwards from 50 to 183 after being tested with the new algorithm. Therefore, it appears that the new algorithm does indeed perform better than the original algorithm, or at least as well as it does.

There are a few major findings we can determine from the augmented DIAMOnD tests. Based on the random selections for each of the diseases, it appears that the augmented DIAMOnD algorithm was successful in improving disease module detection for all of them. In addition, based on the true positive graphs, it appears that the augmented algorithm performs at least as well as, or better than, the original algorithm does, since its best fit line is close to or higher than that of the original algorithm for all of the diseases. It also appears that the augmented algorithm has an exceptionally high performance for some of the diseases, like leukemia, and performs significantly better than the original algorithm in these cases. Furthermore, it appears that while the augmented algorithm performs better on the PPI network than the co-expression network, it produces a higher improvement over the original algorithm for the co-expression network than the PPI network. Based on this information, we can conclude that changing the probability distribution of DIAMOnD to add weights to links that exist in additional networks does improve the algorithm's ability to find correct genes to add to the disease network.

While these results provide good support for the augmented algorithm, it is important to consider their limitations. I only tested the algorithms on five diseases, so further testing may be needed to see if the results hold for a larger sample size. I also only used two types of biological

networks, so it is possible that other types will produce different results. To further complicate matters, I didn't have time to fully experiment with the augmented DIAMOnD algorithm, as I only tested it with a weight of 2 for genes in both networks and a weight of 1 for genes in one network. In addition, it is difficult to determine how accurate the gene ontology test is, since it relies on having a fixed statistical significance limit for biological terms connected to the seed genes. This significance restriction can be overestimated or underestimated, and the lists of biological terms in DAVID and other databases may not be entirely accurate. It is also possible that my own program to validate the genes overlooks certain factors when determining the number of true positives. However, these results do at least suggest that the augmented algorithm can sometimes perform better than the original algorithm.

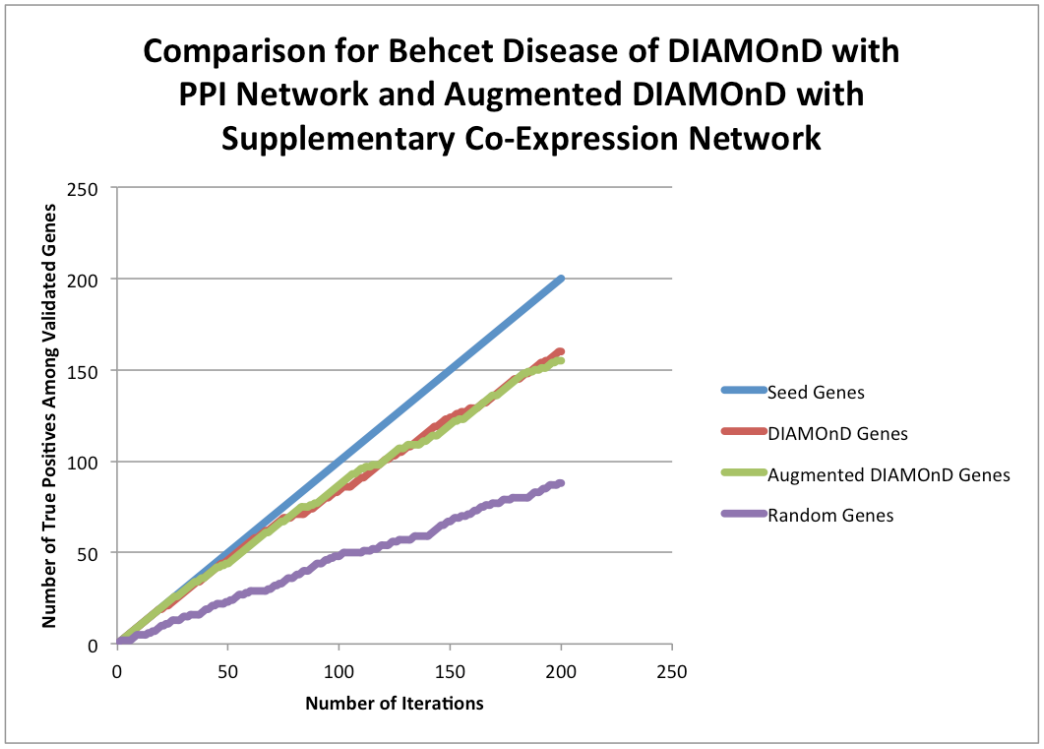


Figure 23: Comparison for Behcet disease of DIAMOnD with PPI network and augmented DIAMOnD with supplementary co-expression network. Each line (for seed genes, DIAMOnD genes, augmented DIAMOnD genes, and random genes) graphs the number of algorithm iterations against the number of "true positive" Behcet disease genes obtained by comparing the GO terms of the seeds genes and algorithm genes. This allows us to see whether the new algorithm, which compares links in the main and supplementary networks, has a better performance.

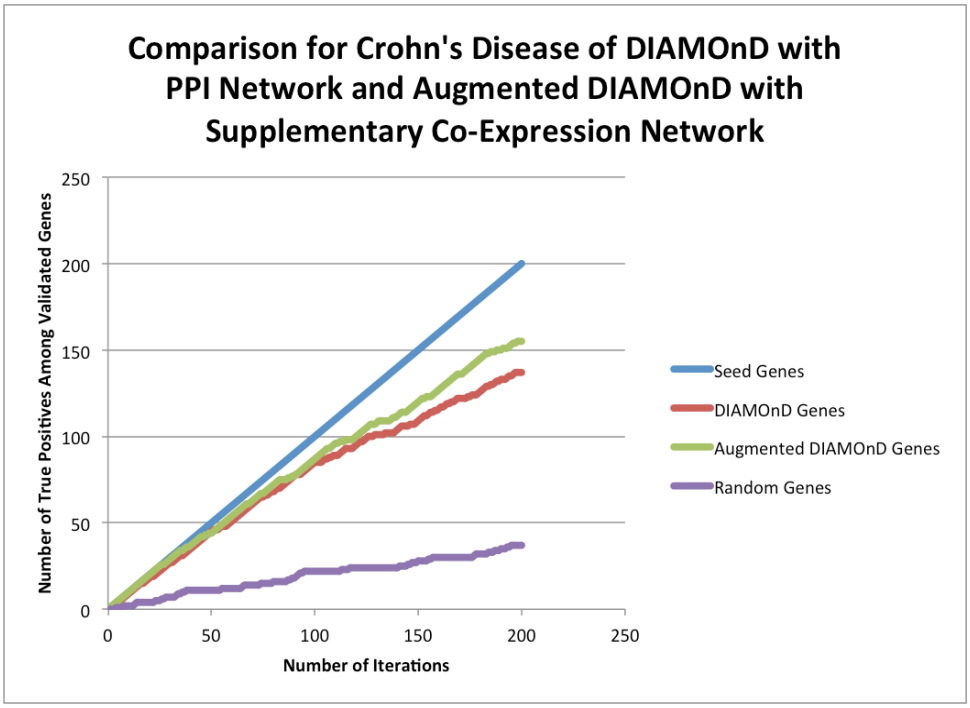


Figure 24: Comparison for Crohn's disease of DIAMOnD with PPI network and augmented DIAMOnD with supplementary co-expression network. Each line (for seed genes, DIAMOnD genes, augmented DIAMOnD genes, and random genes) graphs the number of algorithm iterations against the number of "true positive" Crohn disease genes obtained by comparing the GO terms of the seeds genes and algorithm genes. This allows us to see whether the new algorithm, which compares links in the main and supplementary networks, has a better performance.

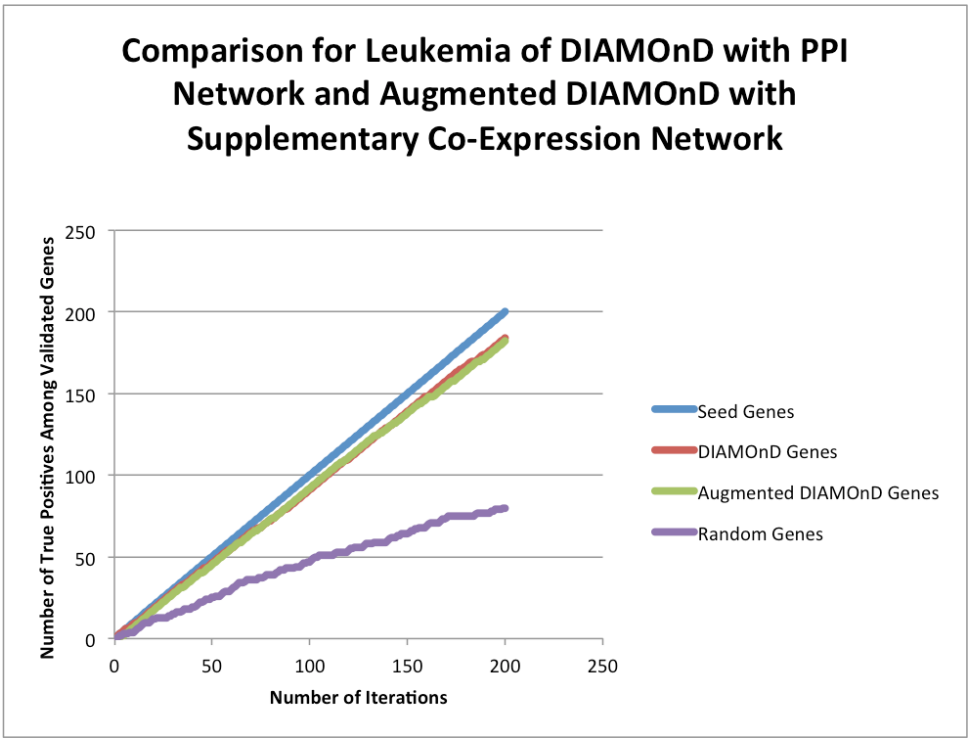


Figure 25: Comparison for leukemia of DIAMOnD with PPI network and augmented DIAMOnD with supplementary co-expression network. Each line (for seed genes, DIAMOnD genes, augmented DIAMOnD genes, and random genes) graphs the number of algorithm iterations against the number of "true positive" leukemia disease genes obtained by comparing the GO terms of the seeds genes and algorithm genes. This allows us to see whether the new algorithm, which compares links in the main and supplementary networks, has a better performance.

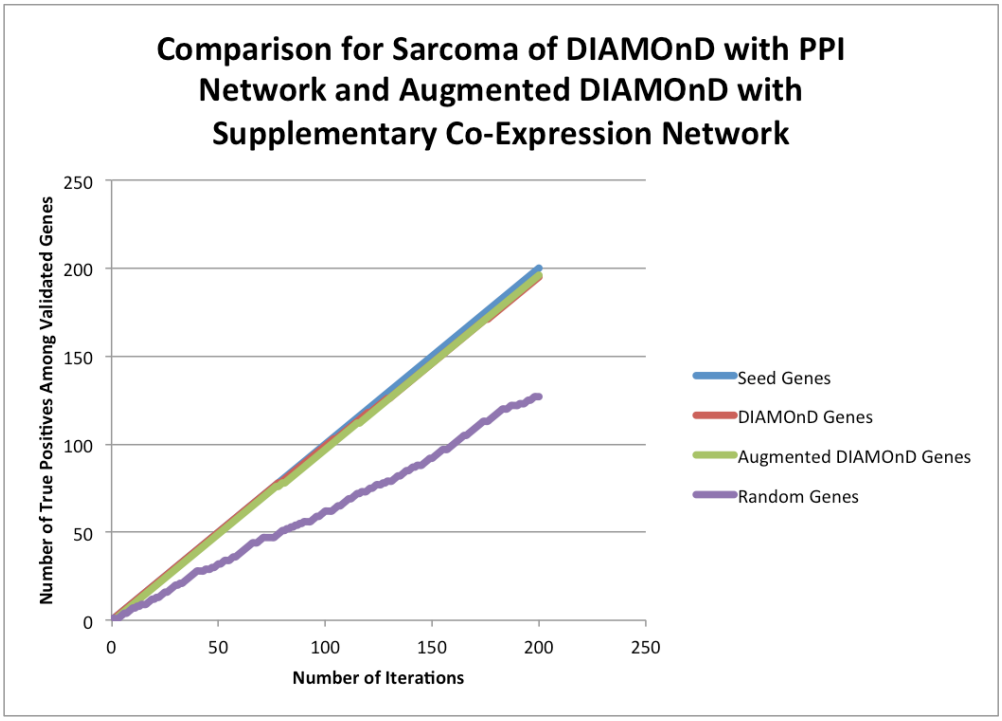


Figure 26: Comparison for sarcoma of DIAMOnD with PPI network and augmented DIAMOnD with supplementary co-expression network. Each line (for seed genes, DIAMOnD genes, augmented DIAMOnD genes, and random genes) graphs the number of algorithm iterations against the number of "true positive" sarcoma disease genes obtained by comparing the GO terms of the seeds genes and algorithm genes. This allows us to see whether the new algorithm, which compares links in the main and supplementary networks, has a better performance.

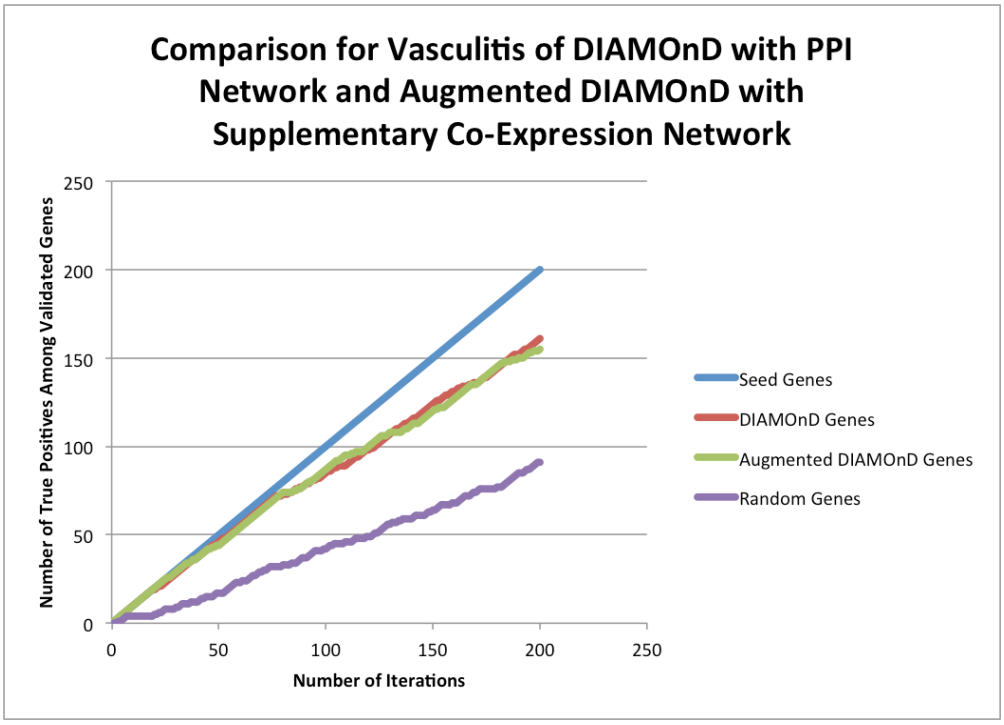


Figure 27: Comparison for vasculitis of DIAMOnD with PPI network and augmented DIAMOnD with supplementary co-expression network. Each line (for seed genes, DIAMOnD genes, augmented DIAMOnD genes, and random genes) graphs the number of algorithm iterations against the number of "true positive" vasculitis disease genes obtained by comparing the GO terms of the seeds genes and algorithm genes. This allows us to see whether the new algorithm, which compares links in the main and supplementary networks, has a better performance.

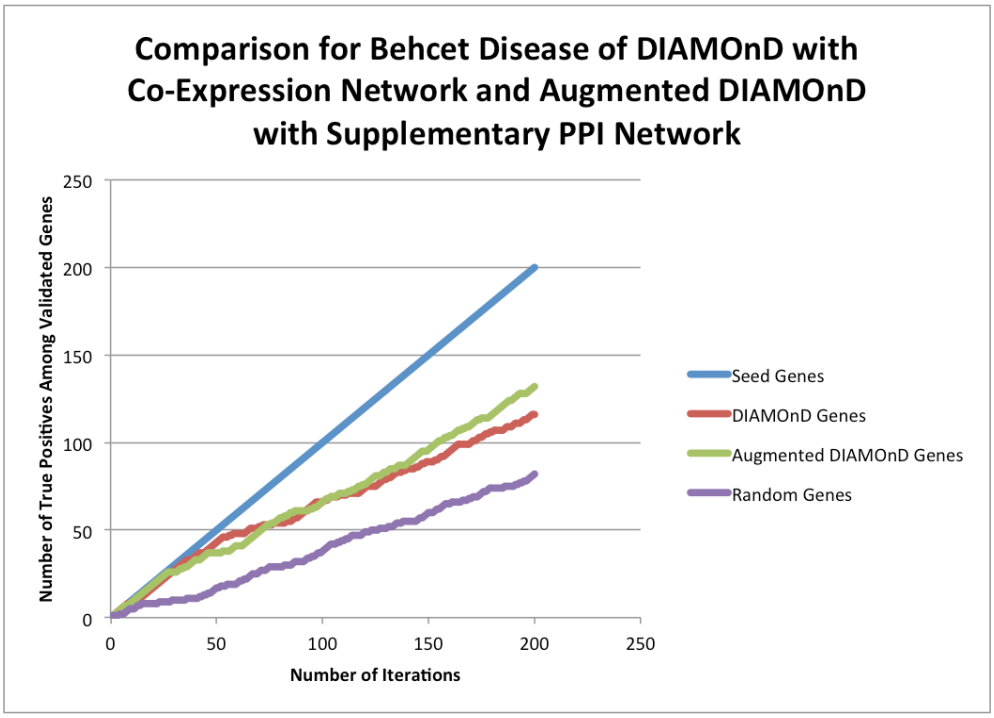


Figure 28: Comparison for Behcet disease of DIAMOnD with co-expression network and augmented DIAMOnD with supplementary PPI network. Each line (for seed genes, DIAMOnD genes, augmented DIAMOnD genes, and random genes) graphs the number of algorithm iterations against the number of "true positive" Behcet disease genes obtained by comparing the GO terms of the seeds genes and algorithm genes. This allows us to see whether the new algorithm, which compares links in the main and supplementary networks, has a better performance.

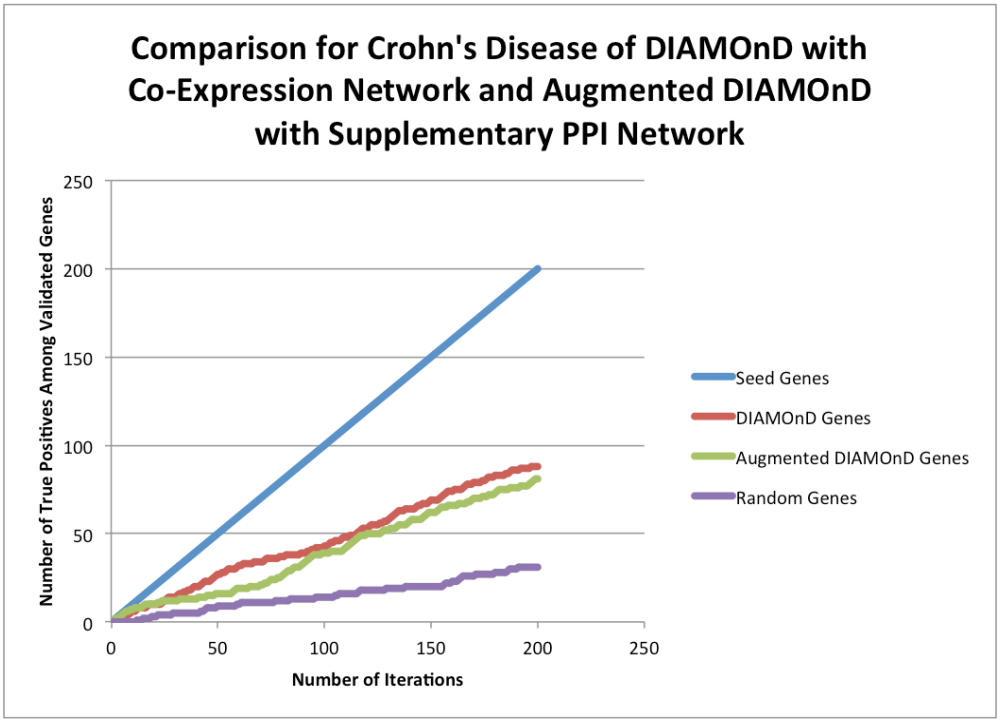


Figure 29: Comparison for Crohn's disease of DIAMOnD with co-expression network and augmented DIAMOnD with supplementary PPI network. Each line (for seed genes, DIAMOnD genes, augmented DIAMOnD genes, and random genes) graphs the number of algorithm iterations against the number of "true positive" Crohn disease genes obtained by comparing the GO terms of the seeds genes and algorithm genes. This allows us to see whether the new algorithm, which compares links in the main and supplementary networks, has a better performance.

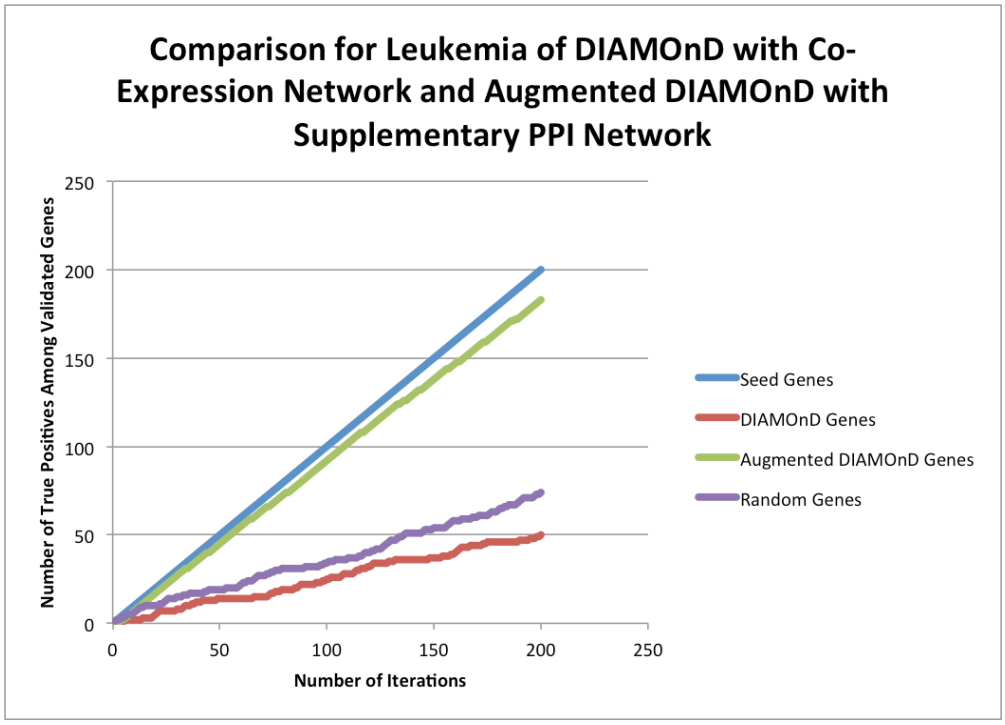


Figure 30: Comparison for leukemia of DIAMOnD with co-expression network and augmented DIAMOnD with supplementary PPI network. Each line (for seed genes, DIAMOnD genes, augmented DIAMOnD genes, and random genes) graphs the number of algorithm iterations against the number of "true positive" leukemia disease genes obtained by comparing the GO terms of the seeds genes and algorithm genes. This allows us to see whether the new algorithm, which compares links in the main and supplementary networks, has a better performance.

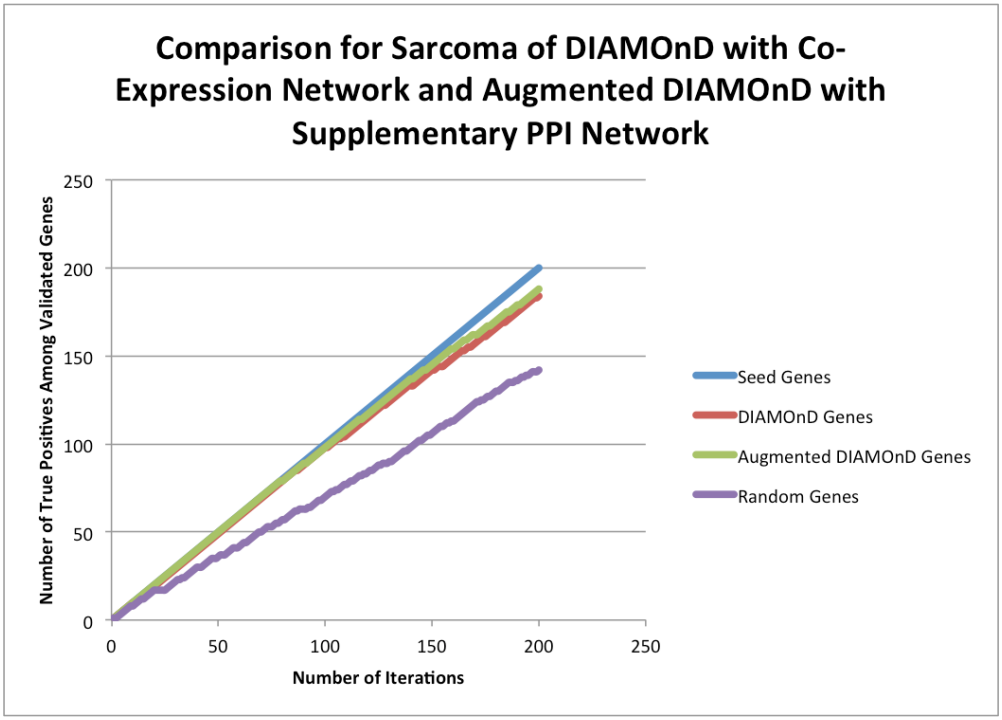


Figure 31: Comparison for sarcoma of DIAMOnD with co-expression network and augmented DIAMOnD with supplementary PPI network. Each line (for seed genes, DIAMOnD genes, augmented DIAMOnD genes, and random genes) graphs the number of algorithm iterations against the number of "true positive" sarcoma disease genes obtained by comparing the GO terms of the seeds genes and algorithm genes. This allows us to see whether the new algorithm, which compares links in the main and supplementary networks, has a better performance.

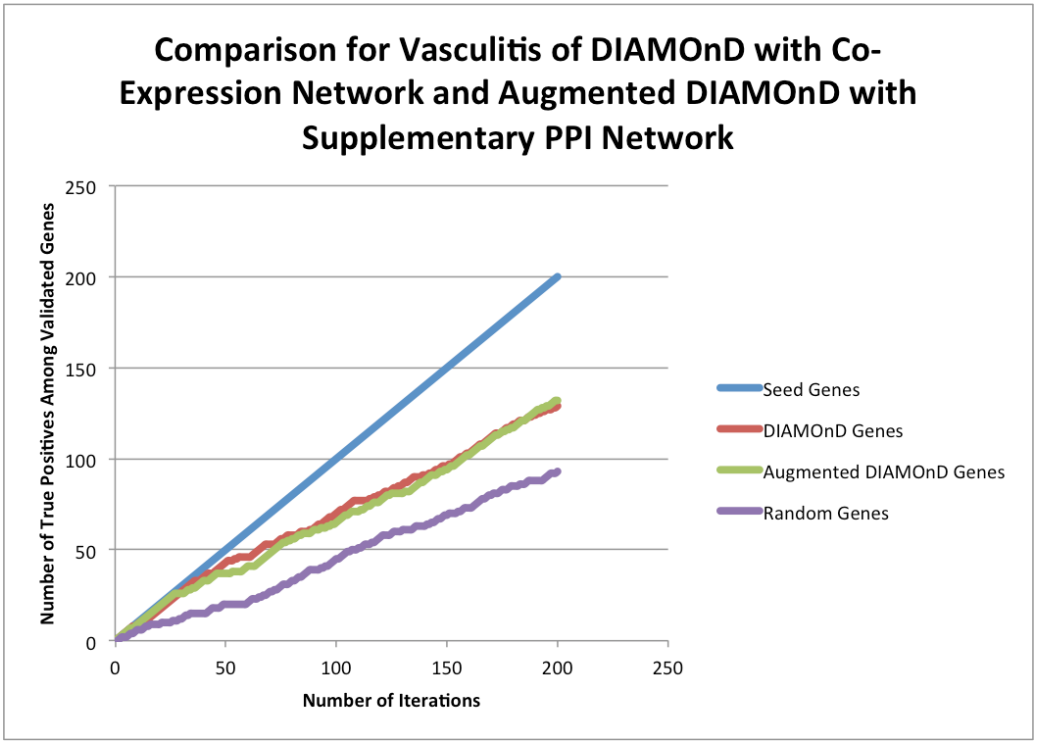


Figure 32: Comparison for vasculitis of DIAMOnD with co-expression network and augmented DIAMOnD with supplementary PPI network. Each line (for seed genes, DIAMOnD genes, augmented DIAMOnD genes, and random genes) graphs the number of algorithm iterations against the number of "true positive" vasculitis disease genes obtained by comparing the GO terms of the seeds genes and algorithm genes. This allows us to see whether the new algorithm, which compares links in the main and supplementary networks, has a better performance.

CHAPTER 5: CONCLUSION AND RECOMMENDATIONS

When I began this project, I knew very little about bioinformatics and biological networks. However, over the course of the project, I have learned a great deal about biological structures and how interconnected they are. I realized that networks and probability both played a major role in how our genes function, and sometimes go wrong. This understanding has enabled biologists and data scientists to make great strides in linking diseases to genes, including the DIAMOnD algorithm. Improving this algorithm mixed my biological knowledge, my programming skills, and my mathematical analysis to apply probabilistic methods to bioinformatics. As far as I know, this augmented algorithm represents one of the first major attempts to improve DIAMOnD. By comparing it to the original DIAMOnD algorithm, I can start to push the algorithm further and continue to explore the interconnected networks of disease genes.

After analyzing both the DIAMOnD algorithm and the upgraded algorithm, I can draw several conclusions about both algorithms. It appears that the DIAMOnD algorithm works as well as in the original experiment, always performing better than the random distribution and almost as well as the seed genes (a perfect score). For the augmented algorithm, it appears that the performance is somewhat better than the original algorithm, although the results are somewhat inconclusive. It also appears that the DIAMOnD algorithm and augmented algorithm both perform better when tested using the PPI network then they do when tested using the co-expression network. However, the new algorithm produces a greater improvement in the co-expression network than in the PPI network. Therefore, it appears that changing the weighting scheme of DIAMOnD could potentially improve its performance. However, due to time and resource constraints, I was unable to fully experiment with the parameters of the new algorithm.

In the future, there are a number of steps that can be taken to improve the results of this project. One step would be to test the DIAMOnD algorithm with additional diseases and networks, such as metabolic or regulatory networks, in order to determine how it performs on those. There are also a number of ways in which future experiments could improve the augmented algorithm. They could try experimenting with the weighting scheme used when calculating the p-values, such as giving genes in both networks a weight of 3 and genes only in the first network a weight of 0.5 (the current setup of the algorithm makes it difficult to use fractions). It also wouldn't be too difficult to

add a functionality to include additional supplementary networks in the algorithm, which would allow for even more experimentation with the weighting scheme. For instance, if I tested the algorithm with a PPI network but also included supplementary co-expression and regulatory networks, then I could assign a link that exists in all three networks a weight of 3. Furthermore, the gene ontology test that I used may not be entirely accurate, so future experiments could try using a test with less error. For instance, PASCAL (Pathway Scoring Algorithm) is used to analyze the validity of disease genes obtained by the GWAS database. Any of these steps could provide new information that would improve the scope of the experiment.

As there are still many questions around the performance of this augmented algorithm, it is difficult to determine how stable my conclusions are. However, one thing that this project definitively confirms is that it is possible to affect the performance of DIAMOnD by changing the weights of its network links based on the information available in additional networks. Therefore, even if this project doesn't conclusively prove that the new algorithm is an improvement over DIAMOnD, it at least demonstrates that messing with its probability distribution can produce a higher number of correct disease genes than the original network. This means that future experiments can build on what I have done with better testing and methodology, determining conclusively whether the algorithm has been improved and continuing to enhance it. Hopefully, these discoveries will contribute to the field of network medicine and make it easier to associate diseases with genes. That way, a day may come when I or someone I know will get a disease and a doctor will, just like finding a malfunctioning car part, simply point out the gene causing the problem and fix it.

ACKNOWLEDGEMENTS

I would like to thank certain people for their help throughout this project. Thank you to Professors Dmitry Korkin and Zheyang Wu for sponsoring this project and giving me the opportunity to fulfill the MQP requirement for both of my majors. You introduced me to the field of bioinformatics and helped me grasp the major concepts enough for me to do something meaningful with this project. Thank you to Hongzhu Cui for helping me throughout this project with advice and instructions about how to run the tests and search the databases, among other things. Without your help, I would not have been able to design a coherent experiment. Finally, I would like to thank my academic advisors for helping me to get involved in this project and for guiding me throughout my academic career at WPI.

BIBLIOGRAPHY

Barabasi, A.-L. L., Joseph; Gulbahce, Natali. (2011). Network Medicine: A Network-Based Approach to Human Disease. Nature Reviews Genetics, 12(1).

Barabasi, A.-L. (2013). Network Medicine: From Cellular Interactions to Human Diseases. Retrieved from https://www.youtube.com/watch?v=qedoIdZrlDM

DAVID. (2009). Retrieved from https://david.ncifcrf.gov/home.jsp

European Bioinformatics Institute. Protein-Protein Interaction Networks. (2017). Retrieved from https://www.ebi.ac.uk/training/online/course/network-analysis-protein-interaction-data-introduction/protein-protein-interaction-networks

Gene Ontology Consortium. (1999). Retrieved from http://www.geneontology.org/

Genetics Home Reference. What Are Proteins and What Do They Do? (2017). Retrieved from https://ghr.nlm.nih.gov/primer/howgeneswork/protein

Genetics Home Reference. What Is a Gene? (2017). Retrieved from https://ghr.nlm.nih.gov/primer/basics/gene

Ghiassian, S. D. M., Jorg; Barabasi, Albert-Laszlo. (2015). A DIseAse MOdule Detection (DIAMOnD) Algorithm Derived from a Systemic Analysis of Disease Proteins in the Human Interactome. PLoS Computational Biology, 11(4).

GWAS Catalog. (2017). Retrieved from https://www.ebi.ac.uk/gwas/

Kirkman, T. W. (1996). Kolmogorov-Smirnov Test. Retrieved from http://www.physics.csbsju.edu/stats/KS-test.html

MacNeil, L. T. W., Albertha J. M. (2011). Gene Regulatory Networks and the Role of Robustness and Stochasticity in the Control of Gene Expression. Genome Research, 21(5), 645-657.

Mandal, A. (2013). What is RNA? Retrieved from http://www.news-medical.net/life-sciences/What-is-RNA.aspx

McKusick, V. (1966). OMIM - Online Mendelian Inheritance in Man. Retrieved from https://www.omim.org/

National Human Genome Research Institute. Genome-Wide Association Studies. (2015). Retrieved from https://www.genome.gov/20019523/

Villa-Vialaneix, N. L., Laurence; Laurent, Thibault; Cherel, Pierre; Gamot, Adrien; SanChristobal, Magali. (2013). The Structure of A Gene Co-Expression Network Reveals Biological Functions Underlying eQTLs. PLoS One, 8(4).

Zitnik, M. (2016). Types of Biological Networks. Retrieved from https://web.stanford.edu/class/cs224w/slides/handout-bionets.pdf

APPENDIX A: LIST OF FILES

In this appendix, I included a list of all the files in my project folder and their uses. The files can all be found in DIAMOnD-master.

File	Purpose
all_seed_genes.txt	List of seed genes used by original DIAMOnD creators
augmented_DIAMOnD.py	Implementation of the upgraded DIAMOnD algorithm
behcet_correct_genes_augmented_coex.txt	Set of true positive numbers for each iteration of augmented algorithm for Behcet disease with main co-expression network
behcet_correct_genes_augmented_ppi.txt	Set of true positive numbers for each iteration of augmented algorithm for Behcet disease with main PPI network
behcet_correct_genes_coex.txt	Set of true positive numbers for each iteration of DIAMOnD algorithm for Behcet disease with co-expression network
behcet_correct_genes_ppi.txt	Set of true positive numbers for each iteration of DIAMOnD algorithm for Behcet disease with PPI network
behcet_correct_random_coex.txt	Set of true positive numbers for each iteration of random selection for Behcet disease with co-expression network
behcet_correct_random_ppi.txt	Set of true positive numbers for each iteration of random selection for Behcet disease with PPI network
behcet_genes_augmented_coex.txt	Set of genes obtained by augmented algorithm for Behcet disease with main PPI network
behcet_genes_augmented_ppi.txt	Set of genes obtained by augmented algorithm for Behcet disease with main co-expression network
behcet_genes_coex.txt	Set of genes obtained by DIAMOnD algorithm

	for Behcet disease with co-expression network
behcet_genes_ppi.txt	Set of genes obtained by DIAMOnD algorithm for Behcet disease with PPI network
behcet_go_terms.txt	Set of GO terms associated with Behcet seed genes
behcet_random_walk_coex.txt	Set of genes obtained by random selection for Behcet disease with co-expression network
behcet_random_walk_ppi.txt	Set of genes obtained by random selection for Behcet disease with PPI network
behcet_seed_genes.txt	Set of seed genes for Behcet disease
coex_network.txt	Co-expression network used for testing
crohn_correct_genes_augmented_coex.txt	Set of true positive numbers for each iteration of augmented algorithm for Crohn's disease with main co-expression network
crohn_correct_genes_augmented_ppi.txt	Set of true positive numbers for each iteration of augmented algorithm for Crohn's disease with main PPI network
crohn_correct_genes_coex.txt	Set of true positive numbers for each iteration of DIAMOnD algorithm for Crohn's disease with co-expression network
crohn_correct_genes_ppi.txt	Set of true positive numbers for each iteration of DIAMOnD algorithm for Crohn's disease with PPI network
crohn_correct_random_coex.txt	Set of true positive numbers for each iteration of random selection for Crohn's disease with co-expression network
crohn_correct_random_ppi.txt	Set of true positive numbers for each iteration of random selection for Crohn's disease with PPI network
crohn_genes_augmented_coex.txt	Set of genes obtained by augmented algorithm for Crohn's disease with main PPI network
crohn_genes_augmented_ppi.txt	Set of genes obtained by augmented algorithm for Crohn's disease with main co-expression

	network
crohn_genes_coex.txt	Set of genes obtained by DIAMOnD algorithm for Crohn's disease with co-expression network
crohn_genes_ppi.txt	Set of genes obtained by DIAMOnD algorithm for Crohn's disease with PPI network
crohn_go_terms.txt	Set of GO terms associated with Crohn seed genes
crohn_random_walk_coex.txt	Set of genes obtained by random selection for Crohn's disease with co-expression network
crohn_random_walk_ppi.txt	Set of genes obtained by random selection for Crohn's disease with PPI network
crohn_seed_genes.txt	Set of seed genes for Crohn's disease
DIAMOnD.py	Implementation of the DIAMOnD algorithm
first_200_for_behcet_augmented_coex.txt	Results of augmented DIAMOnD test with 200 iterations for Behcet disease with main co-expression network
first_200_for_behcet_augmented_ppi.txt	Results of augmented DIAMOnD test with 200 iterations for Behcet disease with main PPI network
first_200_for_behcet_coex.txt	Results of DIAMOnD test with 200 iterations for Behcet disease with co-expression network
first_200_for_behcet_ppi.txt	Results of DIAMOnD test with 200 iterations for Behcet disease with PPI network
first_200_for_crohn_augmented_coex.txt	Results of augmented DIAMOnD test with 200 iterations for Crohn's disease with main co-expression network
first_200_for_crohn_augmented_ppi.txt	Results of augmented DIAMOnD test with 200 iterations for Crohn's disease with main PPI network
first_200_for_crohn_coex.txt	Results of DIAMOnD test with 200 iterations for Crohn's disease with co-expression network
first_200_for_crohn_ppi.txt	Results of DIAMOnD test with 200 iterations

	for Crohn's disease with PPI network
first_200_for_leukemia_augmented_coex.txt	Results of augmented DIAMOnD test with 200 iterations for leukemia with main co-expression network
first_200_for_leukemia_augmented_ppi.txt	Results of augmented DIAMOnD test with 200 iterations for leukemia with main PPI network
first_200_for_leukemia_coex.txt	Results of DIAMOnD test with 200 iterations for leukemia with co-expression network
first_200_for_leukemia_ppi.txt	Results of DIAMOnD test with 200 iterations for leukemia with PPI network
first_200_for_sarcoma_augmented_coex.txt	Results of augmented DIAMOnD test with 200 iterations for sarcoma with main co-expression network
first_200_for_sarcoma_augmented_ppi.txt	Results of augmented DIAMOnD test with 200 iterations for sarcoma with main PPI network
first_200_for_sarcoma_coex.txt	Results of DIAMOnD test with 200 iterations for sarcoma with co-expression network
first_200_for_sarcoma_ppi.txt	Results of DIAMOnD test with 200 iterations for sarcoma with PPI network
first_200_for_vasculitis_augmented_coex.txt	Results of augmented DIAMOnD test with 200 iterations for vasculitis with main co-expression network
first_200_for_vasculitis_augmented_ppi.txt	Results of augmented DIAMOnD test with 200 iterations for vasculitis with main PPI network
first_200_for_vasculitis_coex.txt	Results of DIAMOnD test with 200 iterations for vasculitis with co-expression network
first_200_for_vasculitis_ppi.txt	Results of DIAMOnD test with 200 iterations for vasculitis with PPI network
goa_human.txt	Complete set of human GO terms and their associated genes (used in parse_annotations.py)
leukemia_correct_genes_augmented_coex.txt	Set of true positive numbers for each iteration of augmented algorithm for leukemia with main co-expression network

leukemia_correct_genes_augmented_ppi.txt	Set of true positive numbers for each iteration of augmented algorithm for leukemia with main PPI network
leukemia_correct_genes_coex.txt	Set of true positive numbers for each iteration of DIAMOnD algorithm for leukemia with co-expression network
leukemia_correct_genes_ppi.txt	Set of true positive numbers for each iteration of DIAMOnD algorithm for leukemia with PPI network
leukemia_correct_random_coex.txt	Set of true positive numbers for each iteration of random selection for leukemia with co-expression network
leukemia_correct_random_ppi.txt	Set of true positive numbers for each iteration of random selection for leukemia with PPI network
leukemia_genes_augmented_coex.txt	Set of genes obtained by augmented algorithm for leukemia with main PPI network
leukemia_genes_augmented_ppi.txt	Set of genes obtained by augmented algorithm for leukemia with main co-expression network
leukemia_genes_coex.txt	Set of genes obtained by DIAMOnD algorithm for leukemia with co-expression network
leukemia_genes_ppi.txt	Set of genes obtained by DIAMOnD algorithm for leukemia with PPI network
leukemia_go_terms.txt	Set of GO terms associated with leukemia seed genes
leukemia_random_walk_coex.txt	Set of genes obtained by random selection for leukemia with co-expression network
leukemia_random_walk_ppi.txt	Set of genes obtained by random selection for leukemia with PPI network
leukemia_seed_genes.txt	Set of seed genes for leukemia
parse_annotations.py	Program that parses through complete set of gene annotations and discovers which genes obtained by algorithm have the same gene annotations as a set of seed genes

ppi_network.txt	PPI network used for testing
PPI.txt	PPI network used by original DIAMOnD creators
random_walk.py	Program that selects a certain number of random genes from a network
README.md	Readme file for programs
sarcoma_correct_genes_augmented_coex.txt	Set of true positive numbers for each iteration of augmented algorithm for sarcoma with main co-expression network
sarcoma_correct_genes_augmented_ppi.txt	Set of true positive numbers for each iteration of augmented algorithm for sarcoma with main PPI network
sarcoma_correct_genes_coex.txt	Set of true positive numbers for each iteration of DIAMOnD algorithm for sarcoma with co-expression network
sarcoma_correct_genes_ppi.txt	Set of true positive numbers for each iteration of DIAMOnD algorithm for sarcoma with PPI network
sarcoma_correct_random_coex.txt	Set of true positive numbers for each iteration of random selection for sarcoma with co-expression network
sarcoma_correct_random_ppi.txt	Set of true positive numbers for each iteration of random selection for sarcoma with PPI network
sarcoma_genes_augmented_coex.txt	Set of genes obtained by augmented algorithm for sarcoma with main PPI network
sarcoma_genes_augmented_ppi.txt	Set of genes obtained by augmented algorithm for sarcoma with main co-expression network
sarcoma_genes_coex.txt	Set of genes obtained by DIAMOnD algorithm for sarcoma with co-expression network
sarcoma_genes_ppi.txt	Set of genes obtained by DIAMOnD algorithm for sarcoma with PPI network
sarcoma_go_terms.txt	Set of GO terms associated with Behcet seed

	genes
sarcoma_random_walk_coex.txt	Set of genes obtained by random selection for Behcet disease with co-expression network
sarcoma_random_walk_ppi.txt	Set of genes obtained by random selection for Behcet disease with PPI network
sarcoma_seed_genes.txt	Set of seed genes for Behcet disease
test_augDIAMOnD.txt	Results of augmented DIAMOnD with test data
test_mainList.txt	Test main network for augmented algorithm
test_otherList.txt	Test supplementary network for augmented algorithm
test_seed.txt	Test seed genes for augmented algorithm
vasculitis_correct_genes_augmented_coex.txt	Set of true positive numbers for each iteration of augmented algorithm for vasculitis with main co-expression network
vasculitis_correct_genes_augmented_ppi.txt	Set of true positive numbers for each iteration of augmented algorithm for vasculitis with main PPI network
vasculitis_correct_genes_coex.txt	Set of true positive numbers for each iteration of DIAMOnD algorithm for vasculitis with co-expression network
vasculitis_correct_genes_ppi.txt	Set of true positive numbers for each iteration of DIAMOnD algorithm for vasculitis with PPI network
vasculitis_correct_random_coex.txt	Set of true positive numbers for each iteration of random selection for vasculitis with co-expression network
vasculitis_correct_random_ppi.txt	Set of true positive numbers for each iteration of random selection for vasculitis with PPI network
vasculitis_genes_augmented_coex.txt	Set of genes obtained by augmented algorithm for vasculitis with main PPI network

vasculitis_genes_augmented_ppi.txt	Set of genes obtained by augmented algorithm for vasculitis with main co-expression network
vasculitis_genes_coex.txt	Set of genes obtained by DIAMOnD algorithm for vasculitis with co-expression network
vasculitis_genes_ppi.txt	Set of genes obtained by DIAMOnD algorithm for vasculitis with PPI network
vasculitis_go_terms.txt	Set of GO terms associated with vasculitis seed genes
vasculitis_random_walk_coex.txt	Set of genes obtained by random selection for vasculitis with co-expression network
vasculitis_random_walk_ppi.txt	Set of genes obtained by random selection for vasculitis with PPI network
vasculitis_seed_genes.txt	Set of seed genes for vasculitis

Figure 33: List of files used during testing. This includes the original DIAMOnD algorithm and its test files, the main Python programs, the lists of seed genes for each disease, the lists of gene ontology terms obtained by DAVID for the sets of seed genes, the output files showing the results of each algorithm, the output files showing the number of true positives for the algorithms, and a few test files which I used to make sure the algorithms worked.

APPENDIX B: PROGRAMS

In this appendix, I included the Python code for the programs I used: the DIAMOnD algorithm, the augmented DIAMOnD algorithm, the program for parsing the gene ontology terms, and the program that chooses a random selection of genes. It should be noted that the DIAMOnD algorithm was originally created by Susan Ghiassian, Jorg Menche, and Albert-Laszlo Barabasi, and that my code for the augmented DIAMOnD algorithm is based on it. Also, note that the augmented algorithm has to compare the links in two different networks and then determine what the weighted links are in the main algorithm, so it may take several hours to run. I also included a Readme file for the programs in the list of files.

B.1 DIAMOnD Algorithm

```
#! /usr/bin/env python

"""
# ----------------------------------------------------------------------
# encoding: utf-8

# DIAMOnD.py
# Joerg Menche, Susan D. Ghiassian
# Last Modified: 2014-12-05

# This code runs the DIAMOnD algorithm as described in
#
# A DIseAse MOdule Detection (DIAMOnD) Algorithm derived from a
# systematic analysis of connectivity patterns of disease proteins in
# the Human Interactome
#
# by Susan Dina Ghiassian, Joerg Menche & Albert-Laszlo Barabasi
#
#
# ----------------------------------------------------------------------
"""

import time
import cPickle
import networkx as nx
import numpy as np
import copy
```

```
import scipy.stats
from collections import defaultdict
import csv
import sys


#
================================================================================
=====
def print_usage():

        print ' '
        print '         usage: ./DIAMOnD network_file seed_file n alpha(optional) outfile_name
(optional)'
        print '         ----------------------------------------------------------------'
        print '         network_file : The edgelist must be provided as any delimiter-separated'
        print '                 table. Make sure the delimiter does not exit in gene IDs'
        print '                 and is consistent across the file.'
        print '                 The first two columns of the table will be'
        print '                 interpreted as an interaction gene1 <==> gene2'
        print '         seed_file       : table containing the seed genes (if table contains'
        print '                 more than one column they must be tab-separated;'
        print '                 the first column will be used only)'
        print '         n       : desired number of DIAMOnD genes, 200 is a reasonable'
        print '                 starting point.'
        print '         alpha           : an integer representing weight of the seeds,default'
        print '                 value is set to 1'
        print '         outfile_name : results will be saved under this file name'
        print '                 by default the outfile_name is set to
"first_n_added_nodes_weight_alpha.txt"'
        print ' '


#
================================================================================
=====
def check_input_style(input_list):
        try:
        network_edgelist_file = input_list[1]
        seeds_file = input_list[2]
        max_number_of_added_nodes = int(input_list[3])
        # if no input is given, print out a usage message and exit
        except:
```

```
        print_usage()
        sys.exit(0)
        return

        alpha = 1
        outfile_name =
'first_%d_added_nodes_weight_%d.txt'%(max_number_of_added_nodes,alpha)

        if len(input_list)==5:
        try:
        alpha = int(input_list[4])
        outfile_name = 'first_%d_added_weight_%d.txt'%(max_number_of_added_nodes,alpha)
        except:
        outfile_name = input_list[4]

        if len(input_list)==6:
        try:
        alpha = int(input_list[4])
        outfile_name = input_list[5]
        except:
        print_usage()
        sys.exit(0)
        return
        return
network_edgelist_file,seeds_file,max_number_of_added_nodes,alpha,outfile_name

#
==========================================================================
=====
def read_input(network_file,seed_file):
        """
        Reads the network and the list of seed genes from external files.

        * The edgelist must be provided as a tab-separated table. The
        first two columns of the table will be interpreted as an
        interaction gene1 <==> gene2

        * The seed genes mus be provided as a table. If the table has more
        than one column, they must be tab-separated. The first column will
        be used only.

        * Lines that start with '#' will be ignored in both cases
        """
```

```
sniffer = csv.Sniffer()
line_delimiter = None
for line in open(network_file,'r'):
if line[0]=='#':
continue
else:
dialect = sniffer.sniff(line)
line_delimiter = dialect.delimiter
break
if line_delimiter == None:
print 'network_file format not correct'
sys.exit(0)


# read the network:
G = nx.Graph()
for line in open(network_file,'r'):
# lines starting with '#' will be ignored
if line[0]=='#':
continue
# The first two columns in the line will be interpreted as an
# interaction gene1 <=> gene2
#line_data   = line.strip().split('\t')
line_data = line.strip().split(line_delimiter)
node1 = line_data[0]
node2 = line_data[1]
G.add_edge(node1,node2)

# read the seed genes:
seed_genes = set()
for line in open(seed_file,'r'):
# lines starting with '#' will be ignored
if line[0]=='#':
continue
# the first column in the line will be interpreted as a seed
# gene:
line_data = line.strip().split('\t')
seed_gene = line_data[0]
seed_genes.add(seed_gene)

return G,seed_genes
```

```
#
================================================================================
========
def compute_all_gamma_ln(N):
        """
        precomputes all logarithmic gammas
        """
        gamma_ln = {}
        for i in range(1,N+1):
        gamma_ln[i] = scipy.special.gammaln(i)

        return gamma_ln

#
================================================================================
=====
def logchoose(n, k, gamma_ln):
        if n-k+1 <= 0:
        return scipy.infty
        lgn1  = gamma_ln[n+1]
        lgk1  = gamma_ln[k+1]
        lgnk1 = gamma_ln[n-k+1]
        return lgn1 - [lgnk1 + lgk1]

#
================================================================================
=====
def gauss_hypergeom(x, r, b, n, gamma_ln):
        return np.exp(logchoose(r, x, gamma_ln) +
                logchoose(b, n-x, gamma_ln) -
                logchoose(r+b, n, gamma_ln))

#
================================================================================
=====
def pvalue(kb, k, N, s, gamma_ln):
        """
        ---------------------------------------------------------------
        Computes the p-value for a node that has kb out of k links to
        seeds, given that there's a total of s sees in a network of N nodes.

        p-val = \sum_{n=kb}^{k} HypergemetricPDF(n,k,N,s)
```

```
        ---------------------------------------------------------------
        """
        p = 0.0
        for n in range(kb,k+1):
        if n > s:
        break
        prob = gauss_hypergeom(n, s, N-s, k, gamma_ln)
        # print prob
        p += prob

        if p > 1:
        return 1
        else:
        return p

#
==========================================================================
=====
def get_neighbors_and_degrees(G):

        neighbors,all_degrees = {},{}
        for node in G.nodes():
        nn = set(G.neighbors(node))
        neighbors[node] = nn
        all_degrees[node] = G.degree(node)

        return neighbors,all_degrees

#
==========================================================================
=====
# Reduce number of calculations
#
==========================================================================
=====
def reduce_not_in_cluster_nodes(all_degrees,neighbors,G,not_in_cluster,cluster_nodes,alpha):
        reduced_not_in_cluster = {}
        kb2k = defaultdict(dict)
        for node in not_in_cluster:

        k = all_degrees[node]
        kb = 0
        # Going through all neighbors and counting the number of module neighbors
```

```
        for neighbor in neighbors[node]:
        if neighbor in cluster_nodes:
                kb += 1

        #adding wights to the the edges connected to seeds
        k += (alpha-1)*kb
        kb += (alpha-1)*kb
        kb2k[kb][k] =node

        # Going to choose the node with largest kb, given k
        k2kb = defaultdict(dict)
        for kb,k2node in kb2k.iteritems():
        min_k = min(k2node.keys())
        node = k2node[min_k]
        k2kb[min_k][kb] = node

        for k,kb2node in k2kb.iteritems():
        max_kb = max(kb2node.keys())
        node = kb2node[max_kb]
        reduced_not_in_cluster[node] =(max_kb,k)

        return reduced_not_in_cluster

#============================================================================
===============
#  C O R E    A L G O R I T H M
#============================================================================
===============
def diamond_iteration_of_first_X_nodes(G,S,X,alpha):

        """

        Parameters:
        ----------
        - G:    graph
        - S:    seeds
        - X:    the number of iterations, i.e only the first X gened will be
        pulled in
        - alpha: seeds weight

        Returns:
        --------
```

```
- added_nodes: ordered list of nodes in the order by which they
are agglomerated. Each entry has 4 info:

* name : dito
* k       : degree of the node
* kb   : number of +1 neighbors
* p       : p-value at agglomeration

"""

N = G.number_of_nodes()

added_nodes = []


# ------------------------------------------------------------------
# Setting up dictionaries with all neighbor lists
# and all degrees
# ------------------------------------------------------------------
neighbors,all_degrees = get_neighbors_and_degrees(G)

# ------------------------------------------------------------------
# Setting up initial set of nodes in cluster
# ------------------------------------------------------------------

cluster_nodes = set(S)
not_in_cluster = set()
s0 = len(cluster_nodes)

s0 += (alpha-1)*s0
N +=(alpha-1)*s0

# ------------------------------------------------------------------
# precompute the logarithmic gamma functions
# ------------------------------------------------------------------
gamma_ln = compute_all_gamma_ln(N+1)

# ------------------------------------------------------------------
# Setting initial set of nodes not in cluster
# ------------------------------------------------------------------
for node in cluster_nodes:
not_in_cluster |= neighbors[node]
not_in_cluster -= cluster_nodes
```

```
# ------------------------------------------------------------------
#
# M A I N        L O O P
#
# ------------------------------------------------------------------

all_p = {}

while len(added_nodes) < X:

# ------------------------------------------------------------------
#
# Going through all nodes that are not in the cluster yet and
# record k, kb and p
#
# ------------------------------------------------------------------

info = {}

pmin = 10
next_node = 'nix'
reduced_not_in_cluster = reduce_not_in_cluster_nodes(all_degrees,
                                    neighbors,G,
                                    not_in_cluster,
                                    cluster_nodes,alpha)

for node,kbk in reduced_not_in_cluster.iteritems():
# Getting the p-value of this kb,k
# combination and save it in all_p, so computing it only once!
kb,k = kbk
try:
        p = all_p[(k,kb,s0)]
except KeyError:
        p = pvalue(kb, k, N, s0, gamma_ln)
        all_p[(k,kb,s0)] = p

# recording the node with smallest p-value
if p < pmin:
        pmin = p
        next_node = node
```

```
        info[node] = (k,kb,p)

        # ----------------------------------------------------------------
        # Adding node with smallest p-value to the list of aaglomerated nodes
        # ----------------------------------------------------------------
        added_nodes.append((next_node,
                            info[next_node][0],
                            info[next_node][1],
                            info[next_node][2]))

        # Updating the list of cluster nodes and s0
        cluster_nodes.add(next_node)
        s0 = len(cluster_nodes)
        not_in_cluster |= ( neighbors[next_node] - cluster_nodes )
        not_in_cluster.remove(next_node)

        return added_nodes

#
================================================================================
===
#
#  M A I N      D I A M O n D  A L G O R I T H M
#
#
================================================================================
===
def DIAMOnD(G_original,seed_genes,max_number_of_added_nodes,alpha,outfile = None):

        """
        Runs the DIAMOnD algorithm

        Input:
        ------
        - G_original :
        The network
        - seed_genes :
        a set of seed genes
        - max_number_of_added_nodes:
        after how many added nodes should the algorithm stop
        - alpha:
        given weight to the sees
        - outfile:
```

```
filename for the output generates by the algorithm,
if not given the program will name it 'first_x_added_nodes.txt'

Returns:
--------
- added_nodes: A list with 4 entries at each element:
* name : name of the node
* k       : degree of the node
* kb   : number of neighbors that are part of the module (at agglomeration)
* p       : connectivity p-value at agglomeration
-
"""

# 1. throwing away the seed genes that are not in the network
all_genes_in_network = set(G_original.nodes())
seed_genes = set(seed_genes)
disease_genes = seed_genes & all_genes_in_network

if len(disease_genes) != len(seed_genes):
print "DIAMOnD(): ignoring %s of %s seed genes that are not in the network" %(
len(seed_genes - all_genes_in_network), len(seed_genes))




# 2. agglomeration algorithm.
added_nodes = diamond_iteration_of_first_X_nodes(G_original,
                                    disease_genes,
                                    max_number_of_added_nodes,alpha)
# 3. saving the results
with open(outfile,'w') as fout:
print>>fout,'\t'.join(['#rank','DIAMOnD_node'])
rank = 0
for DIAMOnD_node_info in added_nodes:
rank += 1
DIAMOnD_node = DIAMOnD_node_info[0]
p = float(DIAMOnD_node_info[3])
print>>fout,'\t'.join(map(str,([rank,DIAMOnD_node])))

return added_nodes
```

```
#
================================================================================
===
#
# "Hey Ho, Let's go!" -- The Ramones (1976)
#
#
================================================================================
===


if __name__ == '__main__':


        # -----------------------------------------------------
        # Checking for input from the command line:
        # -----------------------------------------------------
        #
        # [1] file providing the network in the form of an edgelist
        #       (tab-separated table, columns 1 & 2 will be used)
        #
        # [2] file with the seed genes (if table contains more than one
        #       column they must be tab-separated; the first column will be
        #       used only)
        #
        # [3] number of desired iterations
        #
        # [4] (optional) seeds weight (integer), default value is 1
        # [5] (optional) name for the results file

        #check if input style is correct
        input_list = sys.argv
        network_edgelist_file,seeds_file,max_number_of_added_nodes,alpha,outfile_name=
check_input_style(input_list)

        # read the network and the seed genes:
        G_original,seed_genes = read_input(network_edgelist_file,seeds_file)

        # run DIAMOnD
        added_nodes = DIAMOnD(G_original,
                seed_genes,
                max_number_of_added_nodes,alpha,
                outfile=outfile_name)
```

```
        print "\n results have been saved to '%s' \n" %outfile_name
```

B.2 Augmented DIAMOnD Algorithm

```
#! /usr/bin/env python


"""
# ----------------------------------------------------------------------
# encoding: utf-8

# augmented_DIAMOnD.py
# Kevin Specht
# Last Modified: 2017-04-27

# This code runs the augmented DIAMOnD algorithm
#  as described in Augmenting DIAMOnD: A Method for Improving
#  Disease Networks Among Human Genes

# Based on
#
# A DIseAse MOdule Detection (DIAMOnD) Algorithm derived from a
# systematic analysis of connectivity patterns of disease proteins in
# the Human Interactome
#
# by Susan Dina Ghiassian, Joerg Menche & Albert-Laszlo Barabasi
#
#
# ----------------------------------------------------------------------
"""

import time
import cPickle
import networkx as nx
import numpy as np
import copy
import scipy.stats
from collections import defaultdict
import csv
import sys
```

```
#
================================================================================
=====
def print_usage():

        print ' '
        print '         usage: ./DIAMOnD network_file seed_file n alpha(optional) outfile_name
(optional)'
        print '         ----------------------------------------------------------------'
        print '         network_edgeList_file1 : The edgelist must be provided as any
delimiter-separated'
        print '                 table. Make sure the delimiter does not exit in gene IDs'
        print '                 and is consistent across the file.'
        print '                 The first two columns of the table will be'
        print '                 interpreted as an interaction gene1 <==> gene2'
        print '         network_edgeList_file2 : The edgelist must be provided as any
delimiter-separated'
        print '                 table. Make sure the delimiter does not exit in gene IDs'
        print '                 and is consistent across the file.'
        print '                 The first two columns of the table will be'
        print '                 interpreted as an interaction gene1 <==> gene2'
        print '         seed_file       : table containing the seed genes (if table contains'
        print '                 more than one column they must be tab-separated;'
        print '                 the first column will be used only)'
        print '         n       : desired number of augmented DIAMOnD genes, 200 is a
reasonable'
        print '                 starting point.'
        print '         alpha           : an integer representing weight of the links in multiple
networks,default'
        print '                 value is set to 1'
        print '         outfile_name : results will be saved under this file name'
        print '                 by default the outfile_name is set to
"first_n_added_nodes_weight_alpha.txt"'
        print ' '


#
================================================================================
=====
def check_input_style(input_list):
        try:
        network_edgelist_file1 = input_list[1]
        network_edgelist_file2 = input_list[2]
```

```
        seeds_file = input_list[3]
        max_number_of_added_nodes = int(input_list[4])
        # if no input is given, print out a usage message and exit
        except:
        print_usage()
        sys.exit(0)
        return

        alpha = 1
        outfile_name =
'first_%d_added_nodes_weight_%d.txt'%(max_number_of_added_nodes,alpha)

        if len(input_list)==6:
        try:
        alpha = int(input_list[5])
        outfile_name = 'first_%d_added_weight_%d.txt'%(max_number_of_added_nodes,alpha)
        except:
        outfile_name = input_list[5]

        if len(input_list)==7:
        try:
        alpha = int(input_list[5])
        outfile_name = input_list[6]
        except:
        print_usage()
        sys.exit(0)
        return
        return
network_edgelist_file1,network_edgelist_file2,seeds_file,max_number_of_added_nodes,alpha,o
utfile_name

#
================================================================================
=====
def read_input(network_file,seed_file):
        """
        Reads the main network and the list of seed genes from external files.

        * The edgelist must be provided as a tab-separated table. The
        first two columns of the table will be interpreted as an
        interaction gene1 <==> gene2

        * The seed genes mus be provided as a table. If the table has more
```

```
than one column, they must be tab-separated. The first column will
be used only.

* Lines that start with '#' will be ignored in both cases
"""

sniffer = csv.Sniffer()
line_delimiter = None
for line in open(network_file,'r'):
if line[0]=='#':
continue
else:
dialect = sniffer.sniff(line)
line_delimiter = dialect.delimiter
break
if line_delimiter == None:
print 'network_file format not correct'
sys.exit(0)


# read network:
G = nx.Graph()
for line in open(network_file,'r'):
# lines starting with '#' will be ignored
if line[0]=='#':
continue
# The first two columns in the line will be interpreted as an
# interaction gene1 <=> gene2
#line_data   = line.strip().split('\t')
line_data = line.strip().split(line_delimiter)
node1 = line_data[0]
node2 = line_data[1]
G.add_edge(node1,node2)

# read the seed genes:
seed_genes = set()
for line in open(seed_file,'r'):
# lines starting with '#' will be ignored
if line[0]=='#':
continue
# the first column in the line will be interpreted as a seed
# gene:
line_data = line.strip().split('\t')
```

```
        seed_gene = line_data[0]
        seed_genes.add(seed_gene)

        return G,seed_genes

#
================================================================================
========
def compare_lists(network1, network2, seedsFile):
        """
        Determines which links from the main network also exist in the
        secondary network in order to determine the weights of each link
        """

        mainList=set() #links in main network
        otherList=set() #links in supplementary network
        seedList=set() #seed genes
        all_genes=set() #all eligible disease genes in main network (not seeds)

        main=[] #temporary main network
        other=[] #temporary supplementary network
        for line in open(network1):
        main.append(line.rstrip()) #get main links
        for line in open(network2):
        other.append(line.rstrip()) #get supplementary lins
        for line in open(seedsFile):
        seedList.add(line.rstrip()) #get seed genes
        for line in main: #get all eligible disease genes from main network
        genes=line.split()
        if genes[0] not in all_genes:
        if genes[0] not in seedList:
                all_genes.add(genes[0])
        if genes[1] not in all_genes:
        if genes[1] not in seedList:
                all_genes.add(genes[1])

        good_links=[] #links in both networks (weighted)

        #sort genes in each link of main list alphabetically
        for link in main:
        genes=link.split()
        newlink=sorted([genes[0],genes[1]])
        str=''
```

```
        for i in newlink:
        str+=i
        str+='\t'
        mainList.add(str)

        #sort genes in each link of supplementary list alphabetically
        for link in other:
        genes=link.split()
        newlink=sorted([genes[0],genes[1]])
        str=''
        for i in newlink:
        str+=i
        str+='\t'
        otherList.add(str)

        #find all links that appear in both lists
        good_links=mainList.intersection(otherList)

        #return list of weighted links
        return good_links

#
================================================================================
========
def compute_all_gamma_ln(N):
        """
        precomputes all logarithmic gammas
        """
        gamma_ln = {}
        for i in range(1,N+1):
        gamma_ln[i] = scipy.special.gammaln(i)

        return gamma_ln

#
================================================================================
=====
def logchoose(n, k, gamma_ln):
        if n-k+1 <= 0:
        return scipy.infty
        lgn1  = gamma_ln[n+1]
        lgk1  = gamma_ln[k+1]
        lgnk1 = gamma_ln[n-k+1]
```

```
        return lgn1 - [lgnk1 + lgk1]

#
==========================================================================
=====
def gauss_hypergeom(x, r, b, n, gamma_ln):
        return np.exp(logchoose(r, x, gamma_ln) +
                logchoose(b, n-x, gamma_ln) -
                logchoose(r+b, n, gamma_ln))

#
==========================================================================
=====
def pvalue(kb, k, N, s, gamma_ln):
        """
        ---------------------------------------------------------------
        Computes the p-value for a node that has kb out of k links to
        seeds, given that there's a total of s sees in a network of N nodes.

        p-val = \sum_{n=kb}^{k} HypergemetricPDF(n,k,N,s)
        ---------------------------------------------------------------
        """
        p = 0.0
        for n in range(kb,k+1):
        if n > s:
        break
        prob = gauss_hypergeom(n, s, N-s, k, gamma_ln)

        p += prob

        if p > 1:
        return 1
        else:
        return p

#
==========================================================================
=====
def get_neighbors_and_degrees(G):

        neighbors,all_degrees = {},{}
        for node in G.nodes():
        nn = set(G.neighbors(node))
```

```
        neighbors[node] = nn
        all_degrees[node] = G.degree(node)

        return neighbors,all_degrees

#
================================================================================
=====
# Reduce number of calculations
#
================================================================================
=====
def
reduce_not_in_cluster_nodes(all_degrees,neighbors,G,not_in_cluster,cluster_nodes,alpha,goo
d_links):
        reduced_not_in_cluster = {}
        kb2k = defaultdict(dict)
        i=0
        for node in not_in_cluster:

        k = all_degrees[node]
        kb = 0
        # Going through all neighbors and counting the number of module neighbors
        for neighbor in neighbors[node]:
        if neighbor in cluster_nodes:
                kb += 1

        #loop through the weighted links
        for link in good_links:
        #if either gene is the current gene in the network
        #  add extra weight to its connections, if the other
        #  gene is a seed gene add extra weight to seed connections
        genes=link.split()
        if node==genes[0]:
                k += (alpha-1)
                if genes[1] in cluster_nodes:
                kb += (alpha-1)
        elif node==genes[1]:
                k += (alpha-1)
                if genes[0] in cluster_nodes:
                kb += (alpha-1)
        kb2k[kb][k] =node
        i=i+1
```

```
        # Going to choose the node with largest kb, given k
        k2kb = defaultdict(dict)
        for kb,k2node in kb2k.iteritems():
        min_k = min(k2node.keys())
        node = k2node[min_k]
        k2kb[min_k][kb] = node

        for k,kb2node in k2kb.iteritems():
        max_kb = max(kb2node.keys())
        node = kb2node[max_kb]
        reduced_not_in_cluster[node] =(max_kb,k)

        return reduced_not_in_cluster

#=========================================================================
===============
#  C O R E    A L G O R I T H M
#=========================================================================
===============
def diamond_iteration_of_first_X_nodes(G,S,X,alpha,good_links):

        """

        Parameters:
        ----------
        - G:     graph
        - S:     seeds
        - X:     the number of iterations, i.e only the first X genes will be
        pulled in
        - alpha: seeds weight
        - good_links: set of weighted links in the main network

        Returns:
        --------

        - added_nodes: ordered list of nodes in the order by which they
        are agglomerated. Each entry has 4 info:

        * name : dito
        * k      : degree of the node
        * kb   : number of +1 neighbors
        * p      : p-value at agglomeration
```

```
"""

N = G.number_of_nodes()
added_nodes = []


# ------------------------------------------------------------------
# Setting up dictionaries with all neighbor lists
# and all degrees
# ------------------------------------------------------------------
neighbors,all_degrees = get_neighbors_and_degrees(G)

# ------------------------------------------------------------------
# Setting up initial set of nodes in cluster
# ------------------------------------------------------------------

cluster_nodes = set(S)
not_in_cluster = set()
s0 = len(cluster_nodes)

#loop through the weighted links
for link in good_links:
genes=link.split()
#if either gene is a seed gene add extra weight to seed genes
if genes[0] in cluster_nodes:
s0 += (alpha-1)
if genes[1] in cluster_nodes:
s0 += (alpha-1)
#add extra weight to total genes
N +=(alpha-1)

# ------------------------------------------------------------------
# precompute the logarithmic gamma functions
# ------------------------------------------------------------------
gamma_ln = compute_all_gamma_ln(N+1)

# ------------------------------------------------------------------
# Setting initial set of nodes not in cluster
#   (non-seed nodes connected to seed nodes)
# ------------------------------------------------------------------
for node in cluster_nodes:
not_in_cluster |= neighbors[node]
not_in_cluster -= cluster_nodes
```

```
# ------------------------------------------------------------------
#
# M A I N        L O O P
#
# ------------------------------------------------------------------

all_p = {}

while len(added_nodes) < X:

# ------------------------------------------------------------------
#
# Going through all nodes that are not in the cluster yet and
# record k, kb and p
#
# ------------------------------------------------------------------

info = {}

pmin = 10
next_node = 'nix'
reduced_not_in_cluster = reduce_not_in_cluster_nodes(all_degrees,
                                        neighbors,G,
                                        not_in_cluster,
                                        cluster_nodes,alpha,good_links)

for node,kbk in reduced_not_in_cluster.iteritems():
# Getting the p-value of this kb,k
# combination and save it in all_p, so computing it only once!
#print(node)
kb,k = kbk
try:
        p = all_p[(k,kb,s0)]
except KeyError:
        p = pvalue(kb, k, N, s0, gamma_ln)
        all_p[(k,kb,s0)] = p

# recording the node with smallest p-value
if p < pmin:
        pmin = p
        next_node = node
```

```
        info[node] = (k,kb,p)

        # ------------------------------------------------------------------
        # Adding node with smallest p-value to the list of aaglomerated nodes
        # ------------------------------------------------------------------
        added_nodes.append((next_node,
                            info[next_node][0],
                            info[next_node][1],
                            info[next_node][2]))

        # Updating the list of cluster nodes and s0
        cluster_nodes.add(next_node)
        s0 = len(cluster_nodes)
        not_in_cluster |= ( neighbors[next_node] - cluster_nodes )
        not_in_cluster.remove(next_node)

        return added_nodes

#
=============================================================================
===
#
#  M A I N    A U G M E N T E D    D I A M O n D    A L G O R I T H M
#
#
=============================================================================
===
def
augDIAMOnD(G_original,seed_genes,max_number_of_added_nodes,alpha,good_links,outfile =
None):

        """
        Runs the augmented DIAMOnD algorithm

        Input:
        ------
        - G_original :
        The network
        - seed_genes :
        a set of seed genes
        - max_number_of_added_nodes:
        after how many added nodes should the algorithm stop
        - alpha:
```

```
given weight to the sees
- good_links:
set of weighted links in the main network
- outfile:
filename for the output generates by the algorithm,
if not given the program will name it 'first_x_added_nodes.txt'

Returns:
--------
- added_nodes: A list with 4 entries at each element:
* name : name of the node
* k       : degree of the node
* kb   : number of neighbors that are part of the module (at agglomeration)
* p       : connectivity p-value at agglomeration
-
"""

# 1. throwing away the seed genes that are not in the network
all_genes_in_network = set(G_original.nodes())
seed_genes = set(seed_genes)
disease_genes = seed_genes & all_genes_in_network

if len(disease_genes) != len(seed_genes):
print "augDIAMOnD(): ignoring %s of %s seed genes that are not in the network" %(
len(seed_genes - all_genes_in_network), len(seed_genes))




# 2. agglomeration algorithm.
added_nodes = diamond_iteration_of_first_X_nodes(G_original,
                                    disease_genes,
                                    max_number_of_added_nodes,alpha,good_links)
# 3. saving the results
with open(outfile,'w') as fout:
print>>fout,'\t'.join(['#rank','DIAMOnD_node'])
rank = 0
for DIAMOnD_node_info in added_nodes:
rank += 1
DIAMOnD_node = DIAMOnD_node_info[0]
p = float(DIAMOnD_node_info[3])
print>>fout,'\t'.join(map(str,([rank,DIAMOnD_node])))
```

```
        return added_nodes


#
================================================================================
===
#
# "Hey Ho, Let's go!" -- The Ramones (1976)
#
#
================================================================================
===


if __name__ == '__main__':


        # -----------------------------------------------------
        # Checking for input from the command line:
        # -----------------------------------------------------
        #
        # [1] file providing the main network in the form of an edgelist
        #       (tab-separated table, columns 1 & 2 will be used)
        #
        # [2] file providing the secondary network in the form of an edgelist
        #       (tab-separated table, columns 1 & 2 will be used)
        #
        # [3] file with the seed genes (if table contains more than one
        #       column they must be tab-separated; the first column will be
        #       used only)
        #
        # [4] number of desired iterations
        #
        # [5] (optional) weight of links in multiple networks (integer), default value is 1
        # [6] (optional) name for the results file

        #check if input style is correct
        input_list = sys.argv

network_edgelist_file1,network_edgelist_file2,seeds_file,max_number_of_added_nodes,alpha,o
utfile_name= check_input_style(input_list)

        # read the networks and the seed genes:
```

```
        G_original,seed_genes = read_input(network_edgelist_file1,seeds_file)

        good_links=compare_lists(network_edgelist_file1, network_edgelist_file2, seeds_file)

        # run augmented DIAMOnD
        added_nodes = augDIAMOnD(G_original,
                seed_genes,
                max_number_of_added_nodes,alpha,good_links,
                outfile=outfile_name)

        print "\n results have been saved to '%s' \n" %outfile_name
```

B.3 Parse Annotations

```
#! /usr/bin/env python

"""
# ----------------------------------------------------------------------
# encoding: utf-8

# parse_annotations.py
# Kevin Specht
# Last Modified: 2017-04-27

# This code takes a set of GO terms associated with a set of seed genes
#  for a partiular disease and a set of potential disease genes obtained
#  by an algorithm and determines which of them belong to the disease module
# ----------------------------------------------------------------------
"""


# ------------------------------------------------
        # Checking for input from the command line:
        # ------------------------------------------------
        #
        # [1] file with the seed GO terms (if table contains more than one
        #         column they must be tab-separated; the first column will be
        #         used only)
        #
        # [2] file with the potential disease genes from the algorithm
        #         (if table contains more than one column they must be
        #         tab-separated; the first column will be used only)
        #
        # [3] name for the results file
```

```
import sys
import time
import copy
import csv

"""
Gets the list of GO terms associated with seed genes
"""
def get_annotations():
        f = open(sys.argv[1],'r')
        termList=[]
        for line in f:
        termList.append(line.rstrip())
        return termList

"""
Gets the list of potential disease genes obtained from the algorithm
"""
def get_genes():
        f = open(sys.argv[2],'r')
        geneList=[]
        for line in f:
        geneList.append(line.rstrip())
        return geneList

"""
Parses the complete list of gene annotations to determine which of the
  potential disease genes is associated with any of the terms that the
  seed genes are associated with
"""
def parse_annotations(termList, geneList):
        f = open("goa_human.txt",'r')
        list=[] #list of true positives
        for gene in geneList: #loop through every potential gene
        for line in f: #parse every line in gene annotations
        if gene in line: #if the potential gene is on the line
                for term in termList:
                if term in line: #if the gene is associated with one of the seed terms
                if gene not in list: #if the gene is not yet in the true positive list
                        list.append(gene) #add gene to true positive list
        f.seek(0) #return to the beginning of the list of gene annotations
        return list
```

```
"""
Main program
"""
f=open(sys.argv[3],'w') #write to outfile
j=1
while(j<=len(get_genes())): #perform a validation for every iteration of the algorithm
        s=parse_annotations(get_annotations(), get_genes()[:j]) #perform GO test
        f.write(str(len(s))) #record number of true positives for this iteration
        print(str(len(s)))
        f.write("\n")
        j=j+1
print("Check file") #loop until iterations are done
f.close()
```

B.4 Random Selection

```
#! /usr/bin/env python

"""
# ----------------------------------------------------------------------
# encoding: utf-8

# parse_annotations.py
# Kevin Specht
# Last Modified: 2017-04-27

# This code takes a network and a number and outputs a set of that
#  number of random genes
# ----------------------------------------------------------------------
"""


# ------------------------------------------------
        # Checking for input from the command line:
        # ------------------------------------------------
        #
        # [1] file with the seed genes (if table contains more than one
        #         column they must be tab-separated; the first column will be
        #         used only)
        #
        # [2] file providing the network in the form of an edgelist
        #         (tab-separated table, columns 1 & 2 will be used)
        #
```

```
	# [3] number of desired iterations
	#
	# [4] name for the results file

import sys
import time
import copy
import csv
import random

"""
Gets the set of seed genes
"""
def get_seed_genes():
	f = open(sys.argv[1],'r')
	seedList=[]
	for line in f:
	seedList.append(line.rstrip())
	f.close()
	return seedList

"""
Gets the links from the network
"""
def get_links(list):
	f=open(list,'r')
	linkList=[]
	for line in f:
	linkList.append(line.rstrip())
	f.close()
	return linkList

"""
Gets all eligible disease genes in the network (not seed genes)
"""
def get_all_genes(list, seed_genes):
	all_genes=[]
	for line in list:
	genes=line.split()
	if genes[0] not in all_genes:
	if genes[0] not in seed_genes:
		all_genes.append(genes[0])
	if genes[1] not in all_genes:
```

```
        if genes[1] not in seed_genes:
                all_genes.append(genes[1])
        return all_genes

"""
Main program
"""
seedList=get_seed_genes() #get inputs
network=get_links(sys.argv[2])
geneList=get_all_genes(network, seedList)
randomList=[] #list of randomly selected genes
for i in range(int(sys.argv[3])): #loop through specified iterations
        choice=random.choice(geneList) #choose a random gene from the network
        randomList.append(choice) #add gene to list of random genes
        geneList.remove(choice) #remove gene from the network

f=open(sys.argv[4],'w') #write to outfile
for rand in randomList:
        f.write(rand)
        f.write("\n")
f.close()
```