

A Simulation Study of Protein Family Degree  
Distribution

by

Bach Vu Dao  
Class of 2008

A thesis submitted to the  
faculty of Wesleyan University  
in partial fulfillment of the requirements for the  
Degree of Bachelor of Arts  
with Departmental Honors in Computer Science

# Acknowledgments

Writing a thesis has been a tiring yet rewarding experience for me. I would not have been able to complete this thesis without the help both intellectually and spiritually from many people.

Firstly, I would like to express my deep gratitude to my thesis advisor, Professor Danny Krizanc. He has provided a lot of guidance and advice for me during the past two semesters. Although I was interested in the topic since the beginning, I was not very confident that I could pursue it with my limited knowledge of biology. However, Professor Krizanc has given me enough resources and directions to do my research. He has also given me a lot of freedom so I could explore all aspects of my topic.

I also thank Professor Michael Rice and James Lipton for being on my thesis committee. I am very grateful to Professor Norman Danner, my academic advisor, for his suggestions in the beginning when I was looking for an advisor. I have been learning a lot from his classes especially Algorithms which I found very useful when doing my research.

I am greatly indebted to Mr. and Mrs. Freeman for their generosity in the past four years. Without their help, I would not even be able to come to Wesleyan and would never have a chance to pursue Computer Science.

I would like to thank my parents, Tung B. Dao and Anh V. Vu, who have taught me to work hard for whatever goal I choose to pursue. They have showed both care and support for me throughout my life. And to them, I would like to dedicate this thesis.

Last but not least, I would like to thank Nhien Huynh for being a constant source of sympathy and encouragement during the past few months. I wish to thank my best friend, Zhaoxuan Yang, for his help during my undergraduate career. I am very grateful to Phillip Zegelbone, my thesis reader, for his help with my writing. I also want to thank all ST Lab thesis campers and my housemates for helping me with my work and getting me through difficult times.

# Abstract

The focus of this thesis is the degree distribution of protein family network graphs. We propose three models of evolution that generate a protein family. The first one uses Sequence Alignment to quantify protein relationship while the second uses the number of mutations accumulated on proteins. The last model incorporates explicitly preferential attachment. Following many other studies, we consider three operations: duplication, gene death and mutation. The main result that we report from the three models is that although exponential distribution is the best fit of the data, power law distribution fits the data well with the rates of evolution found in many studies.

# Contents

List of Figures	6
List of Tables	8
Chapter 1. Introduction	9
1.1. Protein Evolution	9
1.2. Protein Family	10
1.3. Protein Family Distribution	12
1.4. Overview of the Thesis	12
Chapter 2. Models of Protein Families	15
2.1. Background Information	15
2.2. Fitting Experimental Data	17
2.3. Models of Protein Families	18
2.4. Our Model	23
Chapter 3. The Simulation	31
3.1. Sequence Alignment Approach	31
3.2. Mutation Count Approach	44
3.3. Preferential Attachment Model	56
3.4. Poisson Random Variables	57
3.5. Timing Information	60
Chapter 4. Results	61
4.1. Experimental Results	61
4.2. Discussion	71
Chapter 5. Model Verification	73

CONTENTS	5
5.1. Data Retrieval and Degree Distribution Computation	73
5.2. Results	75
Chapter 6. Conclusion	78
6.1. Implications of Results	78
6.2. Extending the Model	79
Bibliography	81
APPENDIX	83

## List of Figures

2.1 Example of Mutation Types	26
2.2 Example of Duplication and Gene Death	28
3.1 Example of Concatenated String	35
3.2 Representation of the concatenated string in Figure 3.1	36
3.3 Example of calculating number of mutations	50
3.4 Example of searching direction	51
4.1 The graphs and fitted curves for Sequence Alignment Approach with Variable DM ratio	63
4.2 The graphs and fitted curves for Sequence Alignment Approach with Variable DM ratio (cont)	64
4.3 The relationship between DM ratio and R value for Sequence Alignment Approach	65
4.4 The relationship between cutoff and R value for Sequence Alignment Approach	65
4.5 The relationship between DD ratio and R value for Sequence Alignment Approach	66
4.6 The relationship between DM ratio and R value for Mutation Count Approach	67
4.7 The relationship between cutoff and R value for Mutation Count Approach	68
4.8 The relationship between DD ratio and R value for Mutation Count Approach	69
4.9 The relationship between DM ratio and R value for Preferential Attachment Model	70
4.10The relationship between cutoff and R value for Preferential Attachment Model	70
4.11The relationship between DD ratio and R value for Preferential Attachment Model	71
5.1 The graphs and fitted curves for real protein data	76
5.2 The graphs and fitted curves for real protein data (cont)	77

5.3 The relationship between Cutoff and R value for Real Data	77
A.1 The graphs and fitted curves for Sequence Alignment Approach with variable cutoff	83
A.2 The graphs and fitted curves for Sequence Alignment Approach with variable cutoff (cont)	84
A.3 The graphs and fitted curves for Sequence Alignment Approach with variable cutoff (cont)	85
A.4 The graphs and fitted curves for Sequence Alignment Approach with variable DD ratio	85
A.5 The graphs and fitted curves for Sequence Alignment Approach with variable DD ratio (cont)	86
A.6 The graphs and fitted curves for Sequence Alignment Approach with variable DD ratio (cont)	87
B.1 The graphs and fitted curves for Mutation Count Approach with variable DM ratio	87
B.2 The graphs and fitted curves for Mutation Count Approach with variable DM ratio (cont)	88
B.3 The graphs and fitted curves for Mutation Count Approach with variable cutoff	88
B.4 The graphs and fitted curves for Mutation Count Approach with variable cutoff (cont)	89
B.5 The graphs and fitted curves for Mutation Count Approach with variable DD ratio	90
B.6 The graphs and fitted curves for Mutation Count Approach with variable DD ratio (cont)	91
C.1 The graphs and fitted curves for Preferential Attachment Model with variable DM ratio	92
C.2 The graphs and fitted curves for Preferential Attachment Model with variable cutoff	93
C.3 The graphs and fitted curves for Preferential Attachment Model with variable cutoff (cont)	94
C.4 The graphs and fitted curves for Preferential Attachment Model with variable Death Rate	94
C.5 The graphs and fitted curves for Preferential Attachment Model with variable Death Rate	95

## List of Tables

3.1 Summary of Sequence Alignment approach's complexity	42
3.2 Jukes and Cantor Substitution	43
3.3 Summary of Mutation Count approach's complexity	56
3.4 Running time for the three models	60

## CHAPTER 1

# Introduction

The study of evolution has rapidly developed in the last few decades. Many biologists have shifted their study to the finer details of evolution, including protein evolution. This thesis sets out to study one of the questions regarding the **protein family** (which is the result of evolution): What are the family network structures and the underlying random process leading to the observed network properties of protein families? To start with, we define many terms regarding protein evolution, protein families and protein family distribution in the following sections.

### 1.1. Protein Evolution

For many centuries, life's origins have been an intriguing problem for scientists. In his famous 19<sup>th</sup> century work *On the Origin of Species*, Charles Darwin introduced his theory of a common ancestor. According to his theory, **natural selection** largely drives the evolutionary process and acts as the main force guarding against harmful variations and advocating new useful functions. Variations and new features are the result of changes in protein structure. In other words, changes in species' traits are expressions of mutations in protein sequences. Therefore, whether the change is a simple difference of fur color to suit different environments, or introduction of a new organ, the underlying mechanism is the same, accumulation of **protein mutations**, or **protein evolution**.

Since the discovery of DNA structure in the middle of the 20<sup>th</sup> century, molecular biology has finally shed some light on how mutations occur. Although scientists had acknowledged the existence of genetic information carrier like DNA before this time, no one had been certain of the process by which this information passed between generations. They later learned that the double helical structure and complementarities between the bases are some of the forces behind DNA's self-replication. As a protein is encoded by a gene on the DNA, the protein's functions are transferred to its descendants. The genetic information contained in DNA is very crucial to how

the protein functions, implying that DNA replication has to be precise to ensure all functions are kept intact. On the other hand, if this process was perfect, we would be exactly like our ancestors and there would be no room for variation. The great varieties of life forms that exist today prove that there must be defects involved in replication. Scientists found out that due to various factors (such as environmental influences), the replication process might involve various errors. For example, one base on the DNA sequence may be removed. One single mutation may not have a significant impact on the protein but accumulation of such errors makes a change of function a possibility, especially when mutations occur at functionally critical sites. Therefore, self-replication has to be good enough to keep critical traits intact, but the process also has to allow room for mutations [21]. Over time, these differences can result in a new organ or even a new species.

However, because mutations, especially those at critical sites, may disrupt the protein's function, natural selection usually prevents them from propagating across the species. For this reason, there would be very few mutations and not enough to create the vast variation in species today or in other words, mutations alone cannot account for the dramatic evolution of proteins. The explanation lies in two other operations in evolution. The first is **gene duplication**. During the course of evolution, a gene may be replicated producing several clones of the same gene in one genome. Duplication is important because it creates the necessary redundancy for mutations to work on [21]. Natural selection ignores mutations on a gene as long as the protein's critical sites are preserved in some other genes. By having many copies, even if one gene is altered, others still keep the same sequence. Consequently, all functions are preserved so that mutations are allowed and can add up to the result observed in nature. Another operation is **gene death** which occurs when a gene either disappears because of outside factors or becomes inactive and does not get transcribed into a protein. In some cases, the gene is harmful for the species and natural selection may prevent that gene from self-duplicating. With these two operations, we can now explain the high rate of mutations allowed for evolution.

## 1.2. Protein Family

Proteins that originate from the same ancestors are grouped into a protein family. Inside this family, each protein may or may not have a genetic relationship with each other (depending

on how far the two proteins have mutated from their ancestors). The evolutionary process can make later generations of a protein different from the protein's ancestral DNA sequences. In each evolution time step, the sequences of proteins drift farther away from those of their ancestors because mutations accumulate. As the number of mutations increases, it is more likely that the protein acquires new functions and become unrelated to the ancestors. We can use this observation to determine the relation between two proteins. The total number of mutations occurring on both genes represents how many mutations are needed to transform one to the other. If this mutation distance is low, it is very likely that the two proteins are genetically related to each other. However, we can only calculate the number of mutations if we can observe the entire evolution process, which is usually not the case for real life protein data. What we have are the current sequences of the proteins and no information about how they have evolved through time. The best that we can do is to approximate the number of mutations by using **Sequence Alignment**. A Sequence Alignment algorithm computes the edit distance between the two strings, which represents how many string operations are needed to transform one gene to the other. The edit distance is proportional but not equivalent to the number of mutations separating the two genes because of the possibility of multiple hits [18]. Several mutations may occur at one position and change one base to another base. The Sequence Alignment only reports one operation while there might be two or more mutations. For this reason, we need to use a **nucleotide substitution model** to approximate the real number of mutations. For real protein data, Sequence Alignment is usually used to classify proteins into families. Thanks to the development of bioinformatics and computation power, scientists have sequenced many proteins and sorted them into families in online databases. These databases serve as the resources for many researchers in the field and particularly, in this thesis, we make use of them to retrieve real life data for our model. Keep in mind that even if we use substitution models, we still can only get an approximation of the relationship between proteins as pointed out by Fitch et al.: homology (or heredity relationship) is an abstraction, as it is a relationship that can not be determined with total certainty [12].

### 1.3. Protein Family Distribution

Why are we concerned about the relationship between genes in a family? The random nature of evolution operations makes the structure of a protein family very hard to analyze. Thanks to advances in the study of network graphs, we can abstract a complex biology structure such as a protein family into a graph. The vertices of this graph are the members of the family and the edges are the connections between these members [18]. Each connection is the relationship that we discussed in the previous paragraph. Two proteins are connected if their total number of accumulated mutations is lower than a **threshold value**. If we are using Sequence Alignment as in the case of real data, this total number is the sum of the results from a Sequence Alignment and substitution model. From the graph of proteins, we can calculate the node degree for each protein by counting the number of vertices in the graph that it has relationship with.

Again, as the operations that create this family are random, the question we are interested in is whether the resulting graph is a random graph or whether it follows some structural organization. One way to determine this, also the method that we choose in this thesis, is by studying the **degree distribution** of the graph. In the case of a random graph, it follows a Poisson distribution [10]. Otherwise, there should be a statistical distribution other than Poisson that can fit the degree distribution. There have been quite a number of studies on this topic many of which pointed to the existence of a **power law distribution** [5,17,30,32]. Other papers led to other distributions such as the **exponential distribution** [28]. How can there be several distributions for the same type of network? What properties of evolution cause power law behavior? We will explore these questions when describing our model in Chapter Two. The result may have several implications about the network structure. For example, as shown by Koonin et al., power law distribution implies a **scale free network** [18]. These implications may play a role in how we can determine whether a group of proteins forms a family.

### 1.4. Overview of the Thesis

This section provides an introduction to the content in each chapter of the thesis. First, we start by describing various models concerning protein network structure and degree distributions that have been proposed in the literature. The existing models provide a good starting point for speculating on the type of distribution that we might observe and constructing our evolution

simulation. To provide a good overview of the literature and what differences our model can bring about, Chapter Two of the thesis describes our model and others' for comparison.

Our study contains two main parts which are **(1)** the simulation of our model and **(2)** the verification of that process by real data. In Chapter Three, we describe how we implement our model in a computer simulation. As we have mentioned in the earlier sections, there are two methods to determine proteins' relationship: using the actual number of mutations or using an approximated value from Sequence Alignment. Taking this into account, we develop two models: one using mutation counting and one using Sequence Alignment. We will explain why counting the actual number of mutations is feasible in our model. However different these two models may be, they have the same mechanism for simulating evolution. The program starts with a random gene and iterates through a number of evolution time steps in each of which an evolution event can happen by chance until the family reaches a limit size. This process is a simple replica of the real evolution process. The resulting set of proteins is, by definition, a family. From this family, we can perform calculations to compute the degree distribution. As some researchers suggest, there are biases in the evolution operations [16, 19], we propose another model that implements explicitly a network property called **Preferential Attachment**. Chapter Three describes the implementation of all these models and also mentions the various concerns that we face when simulating evolution such as approximating the binomial process for each operations or finding the affected proteins in a mutation and their solutions.

The results of this simulation are presented in Chapter Four. Each result of a simulation run is the degree for each node in the resulting family and regression tests to determine what statistical distribution it follows. We test the results with two distributions: power law and exponential as they appear in most papers in the literature. Another important factor affecting the result is the rate of evolution. As there are usually no definite values for the rates at which each operation occurs, we can only try various values to see which one gives the best fit for each distribution. Furthermore, what matters for the result may not be only each individual operation rate but also the rates' relative values. Hence, we present a number of ratios between rates and how they affect the fitted line of each distribution. We also discuss the significance and implications for the network structure of these two types of distribution.

To confirm the fitted distribution received from the simulation, we test it with real data from well known protein sequence databases. As mentioned before, there is a large amount of data on protein sequences and families online. Nevertheless, it is rarely the case that all the data that we need reside in one database. Thus, we need to combine data from several sources to retrieve the nucleotide sequences of a protein family in order to assess our theory. Unlike the simulation, which is just a hypothetical family, this is real data of current proteins. By applying the method in Chapter Four, we can see which distribution fits the data best. Chapter Five describes how we do this with databases such as Pfam and GenBank [6, 11].

With both the simulation and real data, we come to the conclusion that the protein family structure can be explained by power law distribution. Unlike the real data, our simulation suggests that exponential distribution is always a better fit. Also, we discuss how one might use this information about protein families to devise a method to determine whether a set of proteins is in fact a family. This thesis ends with this discussion together with some directions to extend the proposed model.

## Models of Protein Families

### 2.1. Background Information

**2.1.1. DNA structure.** Before getting into the literature on protein family research, we first show how the DNA structure enables the transfer of genetic information between generations. DNA has a double helical shape with two strands of nucleotide sequences, each of which includes a number of the four **nucleotides: cytosine (C), guanine (G), adenine (A) and thymine (T)**. Computational biology takes advantage of this construction to model DNA as a string created from the four letter alphabet **{A, C, G, T}**. Each evolution operation acting on the DNA becomes a string manipulation which is simple to simulate on computers. One base binds to another because of **complementary pairing**. Specifically, adenine on one strand binds to a thymine on the other one, likewise, a cytosine binds to a guanine. This property drives the DNA **self-replication process**. When the DNA is replicated, the two strands will be separated from each other. Each of these strands will serve as a template to form its opposite strand. Every nucleotide in the strand will bind to its complimentary base supplied by the environment. From two single strands, we now have two pairs of strands each of which forms an exact replica of the original pair. DNA also holds the information (called a **gene**) specifying the **amino acid** sequence for a protein. Every three bases (called a **codon**) in the DNA sequence is translated to a specific amino acid according to a translation rule. The translated amino acid sequence forms the protein governing the functions of organs and other traits. Hence, thanks to the DNA's self-replication process, the proteome can be transferred to later generations.

**2.1.2. Power Law Distribution.** The Power law is a type of probability distribution of the form:

$$(2.1) \quad P(x) \sim \alpha(x^\beta)$$

where  $\alpha$  and  $\beta$  are constants and  $x$  is an integer.

If we take the log of both sides of this function, we have

$$(2.2) \quad \log(P(x)) \sim \log(\alpha) + \beta(\log(x)).$$

Equation 2.2 shows that if we plot the power law distribution on a log-log graph (both  $x$  and  $y$  axis are in logarithmic scale), we will have a straight line. Also, equation 2.1 suggests another property of networks following power law distribution based on the following observation. If we scale  $x$  by a constant  $c$ , we have:

$$(2.3) \quad P(cx) \sim \alpha(cx)^\gamma \sim \alpha(c^\gamma(x^\gamma)).$$

Equation 2.3 indicates that scaling  $x$  by a constant multiplies  $P(x)$  by a constant amount implying that the power law distribution is **scale-free**. A network following the power law distribution is a scale-free network (SF network). Due to a power law degree distribution, a SF network has a lot of low degree nodes but only a few high degree nodes each of which acts as a hub. The dominance of nodes that have only a few edges makes a SF network “robust against accidental failures but vulnerable to coordinated attack” [4]. A random failure would more likely remove a low degree node than a hub so the network’s connectivity is still intact. However, if we take out the hubs, the network would fall apart. An example of such a network structure is the Internet. If we consider the Internet as a network of web pages and the hyper-links between them as the edges, there are a few popular pages and many other less well-known sites, which implies a power law distribution [4]. If the non-important pages are removed, the Internet would not be affected much but a removal of a popular site such as Google would have a huge impact. Many other networks have also been found to be scale free (see [1]).

**2.1.3. Exponential Distribution.** An exponential distribution follows the probability distribution of the form:

$$(2.4) \quad P(x) \sim \alpha(e^{\beta x})$$

where  $\alpha$  and  $\beta$  are constants and  $x$  is an integer.

We consider this distribution in our research because it turns out to be a good fit to our data and in most cases, better than the power law distribution.

**2.1.4. R value.** In this thesis, we need to fit a set of data to one of the two previous distributions. To assess how well a distribution fits the data, we will use the **Pearson's Correlation (R value)**. R values range from 0 to 1 and the closer R is to 1, the better the distribution fits the data. Keep in mind that the R value only tells us how well a distribution fits the data but it does not determine if the distribution is the best fit for the data. There may be some other distributions that may fit the data better. In our research, we assume an R value greater than 0.9 to imply that the distribution is an acceptable fit of the data.

## 2.2. Fitting Experimental Data

The result that we get from the simulation is mapping from a degree  $X$  to the number of nodes with that degree  $d(X)$ . We will not use  $X$  and  $P(X)$  to test again a distribution but we will use the complementary cumulative distribution (cdf)  $P(x \geq X)$ . To convert from the number of nodes with degree  $X$  to the probability of nodes with degree greater than  $X$ , we first need to sort the data in ascending order according to the degree, then the complementary cumulative value of a degree  $X$  is calculated as  $P(X) = Pr(x > X)$ . As we will fit the data to power law distribution, we cannot have value at  $X = 0$ . We have to shift the data by 1, hence,  $Pr(x > X)$  for  $X \geq 0$  is equivalent to  $Pr(x \geq X)$  for  $X \geq 1$ . Equation 2.5 shows how we compute the complementary cumulative distribution.

$$(2.5) \quad P(x \geq X) = \frac{\sum_{i=X}^n d(i)}{\sum_{j=1}^n d(j)}$$

where  $n$  is the number of degrees we have and  $d(i)$  is the number of nodes with degree  $i$ .

There is a simple method to fit power law to real life data by using linear regression on the logarithm value of  $P(x)$  and  $x$ . However, as pointed out by Aaron et al. [7], this method might give a biased result. Instead of implementing a linear regression, we will fit all the simulation

results with a software package Kaleidagraph which provides both power law and exponential curve fitting [27].

### 2.3. Models of Protein Families

With an understanding of genetic information transfer and the statistical distributions that we will encounter, we can now describe the various models of protein family evolution. The literature contains many models concerned with different aspects of the network. Here we only consider those dealing with the **degree distribution**. An important common feature of these studies is the use of graphs in examining the network as we have mentioned in Chapter One. By studying the structure of this graph, one can come to many conclusions about the family. Although their approach is similar, each model leads to a different conclusion. Regarding the resulting distribution, these models can be grouped into two groups: Poisson and Power Law. For the Poisson group, we will consider Erdős and Rényi's work and a supporting paper by Smith et al. These two papers both point to a random network of proteins. The Power Law models are more complicated than the Poisson group. We will start from a simple model by Bebek et al. to more complicated ones by Vazquez et al., Karev et al. and lastly one that is quite similar to our model, Yanai et al.

The most significant model of the Poisson distribution group is the work on random graphs by Erdős and Rényi (**ER model**) [10]. In the beginning of evolution, the ER model starts with  $n$  non-connected nodes. In each step, the model randomly connects these nodes. For every two proteins, there is a probability  $p$  that the model would connect them. Clearly, the probability of a connection between two nodes is independent of other edges and consequently, independent of the nodes' degrees [5]. The result of these evolution steps is a protein network graph whose degree distribution follows a Poisson distribution [18]. Smith et al. [25] supported the ER model with a quite different approach. Instead of studying the evolution process, Smith et al. studied the distribution of the nucleic acid similarities. They compared a Sequence Alignment to a run of heads in coin tossing as follows: "The length of the longest run of matches in the alignment is equal to the length of the longest run of heads in the associated coin tossing sequence". Due to this random coin tossing process, the degree distribution followed a Poisson distribution like the ER model. Since protein relationship is approximated by string difference, the Poisson

distribution also applied to protein relationship. However, as we find out later when discussing the Sequence Alignment algorithm, any particular position on one string can be matched with the same letter, a different letter, or a gap on the other string. Hence, there are three possible options and the coin tossing analogy might not work. In both the ER and Smith et al.'s model, the protein graphs size stayed fixed. There were no proteins added or removed from the graph. As gene duplication or death may occur during evolution, the assumption that the family does not change its size may not be realistic.

In contrast, many other recent papers showed a different result. An example is Bebek et al.'s model which reported a power law distribution [5]. Like the ER model, Bebek's model started the evolution process with a set of protein nodes. These nodes were connected to create a graph (denoted  $G$ ) unlike the ER model in which they were separated. At each evolution time step, the model chooses randomly a member  $u$  in the vertex set  $V(G)$  and duplicates it. The replica of this node (denoted  $t$ ) is added to the graph without any edges attached to it. Then, an edge is created with a probability  $p$  to connect  $t$  with each neighbor of the original node.  $t$  could also be connected to any nodes that did not belong to the neighbor set of  $u$  with a probability different from  $p$ . Bebek et al. concluded that the degree distribution of the resulting graph indeed followed a power law distribution [5]. Certainly, duplication was reflected in Bebek et al.'s model [5] but their model did not simulate mutations and gene deaths at a detailed level but abstracted these two operations in the edge-creation process. If we perform mutations on the newly duplicated protein, the protein will lose some relationships with the old neighbors and also will create some new relationships with other nodes. As a result, on the graph level, the new node will be connected to some of the old related nodes and some of the other nodes. The edge-creation process generates the desired result of mutations on newly added proteins. However, Bebek et al.'s model only allowed mutations on the new vertices and all existing vertices were still intact. If two vertices in the graph were connected, they would be connected throughout the whole evolution process. In reality, all proteins in the family are subject to mutations and it is not the case that only new vertices are allowed to be mutated. Moreover, another problem with Bebek et al.'s model is the lack of gene death. Once a protein was added to the protein network, it remained in the network during the course of evolution. In some cases, a protein may be able to survive for a long time. However, as the environment changes, many genes may

not be suitable anymore and natural selection would eliminate them. Hence, the assumption that all proteins would stay in the graph forever may not be realistic. Although Bebek et al.’s model does not consider all aspects of evolution, it has gone much deeper into the evolution process than the previous ER model.

A slightly different method was described by Vazquez et al. [30] in which they still only used graph operations but started to consider more operations than just duplication. They came up with the **Duplication - Divergence (DD)** model. The name came from the two main operations used to create the network. A network is created by a new vertex  $i'$  being duplicated from an existing vertex  $i$  ( $i'$  would be connected to all  $i$ ’s neighbors). Initially, these two nodes are exactly similar but then they start to drift apart as the model randomly removes a link  $(i, j)$  or  $(i', j)$  where  $j$  is any node in the graph. The DD model still does not allow mutations to occur on any node but only on nodes affected by duplication. The resulting degree distribution that the DD model produces is the same as Bebek et al.’s model, a power law distribution.

Karev et al. [17] constructed their model with more operations: duplication, gene death and innovation (a gene acquired not from inside the family but, for example, from horizontal gene transfer) [17]. Following this method, the result turned out to be dependent on the relationship between the birth rate and the death rate. The Power law only occurred when duplication and deletion were asymptotically equal up to the second order [17]. Although we do not consider gene innovation to such an extent as Karev et al.’s model (specifically, we do not consider gene transfer), this model showed the resulting distribution may not be dependent on only each individual evolution operation rate but also on the ratio of two rates. For our model, we will also study the relationship between the ratios of evolution rates and the network degree distribution.

There is other related work that did not directly deal with the degree distribution of the graph but studied the distribution of the **family size**. Yanai et al. proposed a simple model that could record the number of mutations accumulated on a protein [32]. According to this model, there are two events: gene duplication and mutation. The number of mutations is recorded and later used to quantify how related two genes are. Specifically, “gene  $i$  belongs to a family of size  $l$ , if  $i$  has  $l - 1$  genes from which it is operated by some number of mutations less than a given threshold  $\epsilon$ ” [32]. Another way we can think of this is that  $i$  is genetically related to  $l - 1$

proteins or  $i$ 's degree is  $l - 1$ . Although the model was quite simple, Yanai et al. pointed out the importance of using the number of mutations instead of just the distance returned by Sequence Alignment to determine evolution relationship. The approach of Yanai et al. implied that the more mutations accumulated on one protein, the more likely that it was not in a family or the more different the protein would be from the rest in the family. Although Yanai et al. did not study the degree distribution, their model showed a method of recording and using the number of mutations to relate proteins. Based on their method, we develop an implementation of our model that determines protein relationship by counting the number of mutations.

An important contribution of these studies is the conclusion that the degree distribution follows a power law distribution, which is different from Erdős and Rényi's work. A question, then, is: What factors make a power law occur? The answer to this question determines the factors that we need to consider in our research. The work of Barabási et al. pointed out the two shortcomings of the ER model [3]. First, the number of vertices in the graph stays constant at a set number, or there is no network growth. The graph starts with  $n$  elements and this size remains the same during the whole course of evolution. In other words, ER model does not take gene duplication into account. Gene duplication does not only exist but it also plays an important role in protein evolution. Ohno argued in his book *"Evolution by gene duplication"* that without duplication, the level of mutations would not reach the level observed to explain today's wide range of species [21]. It is not practical to assume that there are no changes in the network size. Another shortcoming of the ER model is that the probability that two vertices are connected is random and not dependent on the vertices' degree [3]. Barabási et al. showed that one of the explanations for power law behavior was a network property called Preferential Attachment. Many real networks such as the World Wide Web, the network of movie actors, share this common property. Preferential Attachment implies that the chance that any two members in a network are connected depends on both members' degree. For example, a popular web page would be more likely to be linked to by another page, or an actor was more likely to act with a well known celebrity than just a normal person [3]. In Barabási et al.'s model, the probability  $\Pi$  that a new node would be connected to node  $i$  of degree  $k_i$  is:

$$(2.6) \quad \Pi(k_i) = \frac{k_i}{\sum_j k_j}.$$

Equation 2.6 indicates that the probability that an existing node  $i$  gets connected to a new node is positively related to its degree. A high degree node would connect to more new nodes than a low degree node so the high degree nodes will get even higher degree and vice versa. Do we need both growth and Preferential Attachment for power law behavior? If the network only displays growth but not Preferential Attachment, the distribution would become exponential. On the other hand, if there is only Preferential Attachment, the model would still display a power law degree distribution for up to  $N^2$  time steps where  $N$  is the size of the vertex set in the beginning of evolution [3]. Therefore, Barabási et al. came to the conclusion that both characteristics: network growth and Preferential Attachment were necessary for power law behavior.

If Barabási et al.'s model is correct, we must observe these two properties in all the power law models that we have discussed (Bebek et al., Vazquez et al., Karev et al. and Yanai et al.'s model). Growth can clearly be seen in all models as they all implement gene duplication which ensures that the family size does not stay fixed. What we are concerned of is how they implement Preferential Attachment. For Bebek et al.'s model, as a protein  $i$  is duplicated to create a new node  $i^*$ ,  $i^*$  would be connected to all of  $i$ 's neighbor with a given probability. The probability that an existing node is connected to  $i^*$  depends on whether it is connected to  $i$ . A higher degree node would have more chance to be  $i$ 's neighbor than a lower degree node. For this reason,  $i^*$  would be connected to more high degree nodes than low degree nodes so Preferential Attachment exists in this model. Vazquez et al. and Karev et al.'s models took a similar approach as  $i^*$  is also connected to all  $i$ 's neighbors. Yanai et al.'s model is more complicated as the protein relationship is based on the number of mutations. When the model made a clone  $i^*$  of a gene  $i$ ,  $i^*$ 's number of mutations is  $i$ 's number of mutations plus a small amount  $\eta$ . The distance  $\delta(i^*, j)$  from  $i^*$  to a node  $j$  is calculated as:

$$(2.7) \quad \delta(i^*, j) = \delta(i, j) + \eta.$$

$i^*$  is connected to  $j$  if  $\delta(i^*, j) < \epsilon$  where  $\epsilon$  is a threshold value to determine relationship. Equation 2.7 indicates that  $\delta(i^*, j)$  is positively related to  $\delta(i, j)$ .  $i^*$  would be more likely to connect to  $j$  if  $\delta(i, j)$  was lower than  $\epsilon$  or  $i$  and  $j$  were connected. Although, Yanai et al. took a different approach to compute protein relationship, their implementation of Preferential Attachment is similar to Bebek et al.'s. From all the models, we can see that to get a power law degree distribution, we need to simulate both growth and Preferential Attachment.

The question is whether Preferential Attachment exists in protein evolution in real life? We will discuss this later in this chapter. Basically, some papers suggest that evolution operations such as duplication may be biased towards low degree nodes [16, 19]. These papers were the reason for us to develop a different model to explicitly incorporate Preferential Attachment.

However, we need to keep in mind that all of the models mentioned may not be complicated enough to describe real life protein networks. Stumpf et al. [28] suggested that the actual protein network may not follow a power law distribution. Several other distributions were suggested to fit real life protein network [28]. An interesting point that Stumpf et al. pointed out was that the Poisson distribution certainly did not fit the data, suggesting that real life families were definitely not random networks. The authors also showed that the exponential distribution may be a better fit than the power law distribution. As the paper only researched a few protein networks, Stumpf et al. did not conclude that power law distribution was wrong but that power law may not be the definitive distribution that researchers had to consider when examining a protein network. Another distribution may possibly be a better fit.

## 2.4. Our Model

**2.4.1. General Setup.** In our research, we propose a detailed model of protein evolution. All the papers that we examine except for Yanai et al.'s treat evolution operations as graph operations, for example a mutation is implicit in an edge removal. Our model will simulate evolution operations on the actual protein sequences.

The goal of our model is to generate a set of proteins which certainly constitutes a family in order to study the node distributions. We choose to simulate the evolution process from a single original protein. Starting with one protein, we go through a series of evolution time steps in which some or no evolution events can take place. In nature, evolution operations happen due

to both chance and other factors such as the reproductive fitness of the protein [24]. Here, as we cannot take into account these additional factors, chance is the sole guide. Following other models that we have seen, the events that we concentrate on are **mutation**, **duplication** and **gene death**. One difficulty for the simulation is how to determine the rate of evolution. It is usually the case that the rate of evolution operations is not certain. Most of the time, only a range of these rates is known and, again, we have to pick a number in that range. Also, although we are studying protein sequences, we do not compare the amino acid strings, but rather we use the nucleotide structures because mutations do occur on the nucleotide level and the amino acid sequence abstracts some of this information away from us. In general, our model simulates an iterative process in which any evolution event can take place to produce a protein family.

#### 2.4.2. Model of Evolution.

2.4.2.1. *Mutation.* As we have said earlier, rather than implicitly considering mutation in the edge removal process, we explicitly simulate the operation on each nucleotide. In our model, mutations include any changes to a single position on the DNA sequence, such as a substitution or deletion, so it is reasonable that we consider a mutation as a chance event that can alter a particular position. Our view is a limited subset of mutations as mutations can also affect several nucleotides on the sequence. We will consider the three types of mutation: **base substitution**, **insertion** or **deletion**. Although mutation affects the protein at the nucleotide level, what really impacts the protein's functions is the amino acid transcribed from the codons. To examine the effect of a mutation, one has to consider both the type of the mutation and the position acted upon. Based on these factors, a mutation can have no effect (a **neutral mutation**) or a devastating effect (a **forbidden mutation**) that can change protein functions [21]. By dividing mutations into three operations, we can account for the various effects of mutations such as multiple hits in which many mutations can occur on the same position.

The first mutation type that we consider is **base substitution**, which may be the most frequent among all mutation types. During the DNA duplication phase, a substitution happens when a single base may be substituted by another base. For example, an adenine can be replaced by a guanine. Codons are composed of three bases ( $4^3 = 64$  possible combinations) but there are only 20 amino acids, many different sets of bases can encode the same amino acid. A change in the nucleotide sequence does not necessarily change the amino acid sequence. In other words, the

protein can resist some mutations as long as the amino acid remains unchanged. For example, both the triplet *CCA* and *CCG* specify *Proline* so if a mutation changes the *A* in *CCA* to *G*, the amino acid that this sequence encodes remains the same. Thus, the protein is not impacted in this case, so base substitution can be either neutral or forbidden. A neutral effect usually occurs if the substitution acts on the third position in the coding triplet (about 30% of the time) [24]. In contrast, if the mutation happens at the first or second position, it is very likely that the amino acid will change (about 96% of the time) and we have a forbidden mutation. In the worst case, a mutation can terminate a protein sequence early if it happens to convert a normal triplet to a **terminating codon**, which signifies the end of protein [21]. Clearly, in this case, all the information beyond this ending triplet is lost. Therefore, the new protein can either introduce some new traits or disrupt the normal behaviors and gets eradicated by natural selection. As a result, we can detect forbidden mutations by checking the changes in the protein function. Neutral mutation, on the other hand, may not be detectable as there might not be any function changes if the amino acid sequence is conserved. From the standpoint of evolution, forbidden mutations are very important as they are the main driving force of new, beneficial functions. That is not to say neutral mutations are useless, as Ohno pointed out that neutral mutations serve as intermediate steps towards the forbidden counterpart. Back to the discussion of base substitution, in our study, base substitution is modeled as a random change of one base on a random location to another possible base (one of the remaining three bases). We follow the nucleotide substitution model proposed by Jukes and Cantor, in which there are equal chances that one base is transformed to one of the other three bases. For instance, for base *A*, 1/3 of the time it will be converted to *G* or *C* or *T*. A common mutation, base substitution is a frequent source of changes in DNA sequence. The effect of a substitution may not be tragic but an accumulation of this event may lead to a more serious impact.

The other two types of mutations are **base insertion** and its opposite, **base deletion**. Insertion occurs when a random base is added to a random position in the protein. Base deletion is the removal of one base at a random position. When a multiple of three bases is inserted or deleted, the protein sequence will gain or lose one amino acid and all other codons remain intact. The result is more dramatic if only one or two nucleotides are affected. All the

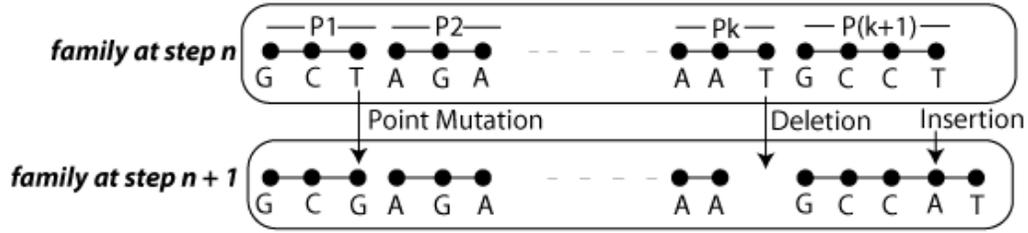


FIGURE 2.1. Example of Mutation Types

codons beyond this point are changed and the sequence will encode a new protein. Therefore, these two types of mutations usually result in forbidden mutations [24].

Like in other evolution events, we need to know the rate of mutation. According to Snustad et al. [26], the mutation rate for eukaryotes is in the range from  $10^{-7}$  to  $10^{-9}$  **detectable mutations per nucleotide pair per generation**. The detectable mutations refer to those that change the amino acid sequence or forbidden mutations. However, as we model mutations on nucleotides, we cannot be sure whether a neutral or forbidden mutation will occur. We need the composite rate (called  $R_{mu}$ ) which should be higher than the rate  $10^{-7}$  reported Snustad et al. [26]. Also, as we divided mutations into three separate operations, we need to divide this  $R_{mu}$  into three parts  $R_{sub}$ ,  $R_{ins}$  and  $R_{del}$  such that  $R_{sub} + R_{ins} + R_{del} = R_{mu}$ . If the rate of insertion is greater than deletion, our gene length can grow very fast in the simulation which is not observed in nature. Gene length tends to stay constant over time. Conversely, if deletion rate is greater than insertion rate, our gene may reach length of zero. To avoid these dramatic changes in gene length, we will use the same value for insertion and deletion rate  $R_{ins} = R_{del}$ . We will pick a value less than  $10^{-7}$  for mutation rate in our research.

**2.4.2.2. Duplication.** As seen in many models, duplication happens when a protein sequence is copied and added back to the family. Duplication, creates a “necessary redundancy” for forbidden mutations. Neutral mutations only account for small changes, larger changes require more drastic mutations. On one hand, forbidden mutations are not tolerated by natural selection, on the other hand, it is needed to explain the various genes in existence today [21]. As a result, duplications serve to create several copies of a gene so that natural selection can allow more

forbidden mutations to occur. Similar to other studies, we simulate this operation by taking a random sequence from the family, duplicating that sequence and then adding the copy to the family.

Unlike the mutation case, we could not find any ranges for the value of duplication rate. A paper by Gao and Innan suggested that the rate is **0.01 to 0.06 per gene per billion years** [15]. This rate is not so helpful for us as we need a measure of duplication in evolution time step units not in years. In the same paper, Gao and Innan stated that the duplication rate is about 28 times the neutral mutation rate which is specified in per base per year. For example, if the neutral mutation rate is  $10^{-6}$  per base pair per generation then the duplication rate is  $10^{-6} \times 28 \sim 1/35714$  per gene per generation.

2.4.2.3. *Gene Death.* Unlike duplications and mutations which are considered in all models, gene death is only considered in a few studies (see [17, 23, 31]). This event happens when a protein is either removed from the family or does not express itself anymore, hence the function that it governs is lost. There can be many reasons for this event. It can be due to environmental factors that cause the disappearance of a gene or maybe because the protein proves harmful to the species and natural selection wipes it out. In fact, during the course of evolution, proteins can accumulate “deleterious substitutions”. Gene death provides the solution for them to get rid of these substitutions and therefore, becomes a major force in evolution [2, 32]. In our model, this event is modeled as a removal of one string in our protein collection.

Like duplication, we could not find any studies stating the definite rate of gene death. As we discussed when describing Karev et al’s model, the death rate should be equal to the duplication rate up to the second order. If the death rate is greater than or equal to the duplication rate, we expect the protein family size will decrease or stay unchanged, which implies that we do not incorporate growth in our model. Therefore, we want the death rate to be less than the duplication. However, if the duplication rate far exceeds the death rate then the size of the family would change radically in each time step, which is not what we observe in nature. In our study, when we need to fix the death rate, we fix it at half the duplication rate. We will also include a study of the ratio between duplication and death rate in our model to show the effect on the family structure.

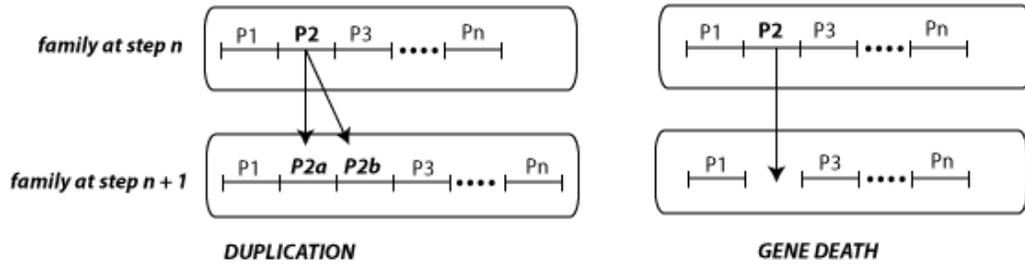


FIGURE 2.2. Example of Duplication and Gene Death

**2.4.3. Determining Protein Relationships.** A major part of our model is determining whether two proteins are related. A traditional approach considers two proteins to be related if their sequences are at least  $X\%$  similar where  $X$  is a tunable parameter of the model. This method requires a way to determine how similar two protein sequences are. If we consider a sequence as a string composed of the four bases, the problem of string similarities can be solved by Sequence Alignment. An alignment of the two sequences finds the optimal way to align the two strings so that we can find how many string edits we should do to transform one to another. As a result, we can quantify the similarity of the two protein sequences.

However, does edit distance really reflect the heredity relationship between a pair of proteins? As suggested in [32], this relationship should be measured by the number of mutation events that separate the two proteins. The number of mutations may not be the same as the other distance because of the possibility of our mutation's nullifying an earlier mutation. For example, a DNA string  $ACCT$  is mutated to be  $ACGT$ . Consider a possible mutation that can occur on the later string, it is possible that the mutation occurs on  $G$  and changes it to  $C$  and now the two strings are actually the same. Hence, it is probable that a mutation can reverse an earlier mutation and the visible distance between two proteins may not reflect accurately how many mutations have occurred. Because of this difference, one should not just simply use the Sequence Alignment distance but attempt to get an estimate of the number of mutations that have occurred. **Nucleotide substitution models** aim exactly to solve this issue. A substitution model takes in the string difference from the Sequence Alignment and returns an approximation of the number of mutations needed to create that much difference. We will

employ the Jukes and Cantor model (JC model) in simulating mutations (see Chapter Three). Modifying the traditional approach, we consider two proteins related if the sequence difference returned by the JC model is at most  $X\%$ . Again, as in the case of evolution rates,  $X$  cannot be determined with 100% certainty but through various trials.

In our model, we can actually count the number of mutations accumulated on a protein. As we have said before, counting mutations may not be feasible because we cannot observe the whole evolution process. For real data, we must use Sequence Alignment and a substitution model. In our case, as we start with one protein in the beginning and iteratively go through several evolution time steps, we are watching the evolution from the start to the end. We can keep a counter of mutations that have occurred and increment it whenever a mutation occurs as seen in Yanai et al.'s model [32]. At the end of evolution, the value of the counter is the total number of mutations that have occurred on the protein. The genetic difference between any two proteins is the sum of their mutations from a common ancestor. Two proteins are related if their genetic difference is lower than a cutoff value. We will implement this mutation counting method as a separate approach to compare the results with those obtained using Sequence Alignment.

**2.4.4. Preferential Attachment.** The model we described above does include network growth as it simulates gene duplications. The other property we need to consider is Preferential Attachment. We can use a similar argument as we did in the case of Bebek et al.'s model. Each new node added to the family is the result of duplicating one of the current members meaning that it must be exactly identical to the original node. The distance between the new node and any existing node  $n$  is equal to the distance between the original node and  $n$ . Similar to Bebek et al.'s model, it is more likely for a high degree node to be related to the original node and hence to the new node. For this reason, we would conclude that our model has already incorporated Preferential Attachment.

The claim of a biased duplication as we presented in Section 2.4.2.2 suggested that we should implement Preferential Attachment more explicitly. Before describing how we do this, we will examine the significance of high and low degree nodes in the family. A high degree node  $i$  means that it has a lot of connections to other proteins in the family.  $i$  would share a common sequence of nucleotides with many other family members which through many rounds of mutations still exists in many proteins. The sequence must play a significant role for the proteins as natural

selection does not eradicate it. As  $i$  contains this important sequence, we can think of  $i$  or any high degree protein as a highly important gene. In contrast, a low degree gene  $j$  means that the gene does not contain many functionally important sequences so  $j$  can be considered as a less important gene. If we want to simulate evolution operations biased towards low degree nodes, we have to explain why evolution operations are more likely to happen on less important nodes.

It is simple to explain why it is biased to towards less important nodes. As a mutation may change a protein's functions (especially if the mutation is forbidden), natural selection will prevent it from occurring on functionally critical sites [21]. On the other hand, for less important genes, a change in the structure does not impact the species much and the mutation can bypass natural selection. As a result, more mutations will occur on less important genes than on important genes. The same reason can be used to explain why gene death might occur more often on less important genes. While it is not certain that a mutation can change the protein functions dramatically as the mutation can be neutral, a gene death certainly leads to an absence of the function specified by the deleted gene. Natural selection would strongly discourage such a harmful operation on an important gene. Mutation and gene death are biased towards low degree proteins due to the effect of natural selection.

For duplication, several papers suggest that duplication occurs more often on low degree nodes. Li et al. reported that preferential duplication occurred in the sparse part of the protein network [19]. A sparse area of a graph is a subgraph that has only a few edges so each element has a low degree. Li et al.'s result indicated that duplication tended to happen more often in the low degree area of the graph. In another paper, He et al. suggested that less important genes had higher duplicability because the duplication of unimportant genes would cause "fewer genetic perturbations" [16]. From these two papers [16, 19], we can simulate a preferential duplication by randomly picking a protein with a degree lower than a randomly generated cutoff degree.

We have shown in this section that there is a method to incorporate Preferential Attachment into our model. We will create a separate model in which every operation is simulated with a bias toward low degree nodes. Chapter Three will describe the actual implementation of this model.

## The Simulation

From the description in the previous chapter, we can have three models: one using Sequence Alignment, one using the number of mutations and one incorporating Preferential Attachment. For each model, we implement a computer simulation program the output of which is a protein family. We then analyze the family's structure by studying its nodes' degree distribution. We will describe the simulation method for each model in the following sections.

### 3.1. Sequence Alignment Approach

**3.1.1. Simulation Description.** For this approach, the program is written as a loop of evolution events that keeps running until the family reaches a certain limit value of members. We can construct a graph from this family and derive the degree distribution. However, as we only need to record the number of nodes for each degree, we can do this while doing the pairwise alignment without the use of a graph (as seen in Listing 3.2). First, we have a Map structure  $M$  mapping a degree to the number of nodes. We will iterate over the list of sequences, and for each sequence  $k$ , we compute the Sequence Alignment distance  $d$  between  $k$  and each of the remaining sequences. If  $d$  is lower than the *cutoff* we specify, we increment  $k$ 's degree by one. After comparing with all other nodes, if  $M$  already contains a key equal to  $k$ 's degree, we increment the value of that key by one. Otherwise, we insert a new pair  $(degree, 1)$  into  $M$ . In the end, the result from this simulation is a Map of degree that we will use in Chapter Four.

Duplication is an event that can randomly occur to any protein in the family. When it occurs on a protein, it makes a copy of that protein and adds the copy to the family. This definition implies a simple method to simulate the event. We can traverse the family and for each protein, we pick zero or one with the probability equal to the duplication rate. However, we can improve the running time by using the **Poisson approximation** described in section 3.4. According to this method, instead of traversing, we can generate a Poisson distribution random number  $\eta$  with the expectation  $n/k$  where  $n$  is the number of strings and  $1/k$  is the duplication rate.  $\eta$  is

---

**Listing 3.1** doSimulation(duplicationRate, mutationRate, deathRate, cutoff)

---

**Precondition:** STRINGLENGTH is a global variable limiting how long a string can be**Precondition:** EXPERIMENTLIMIT is a global variable limiting how many strings there can be a in a family**Return:** a map whose key is the degree and the value is how many nodes with that degree

```

1. list initialized to an empty list
2. p initialized to empty string
3. for  $i = 0$  to  $(STRINGLENGTH - 1)$  do
4.     add a random character from {A,C,G,T} to p
5. end for
6. list.add(p)
7. while  $list.size() < EXPERIMENTLIMIT$  do
8.     doDuplication(list, duplicationRate)
9.     doMutation(list, mutationRate)
10.    doGeneDeath(list, deathRate)
11. end while
12. return constructDistribution(list, cutoff)

```

---



---

**Listing 3.2** constructDistribution(list, cutoff)

---

**Precondition:** list is a list of string from the simulation**Precondition:**  $cutoff < 1$ **Return:** a map whose key is the degree and the value is how many nodes with that degree

```

1. M is an empty map of type  $\langle Integer, Integer \rangle$ 
2. for  $i = 0$  to  $(list.size() - 1)$  do
3.      $degree = 0$ 
4.     for  $j = 0$  to  $(list.size() - 1)$  do
5.          $distance = SequenceAlignment(list[i], list[j])$ 
6.         if  $distance < cutoff$  then
7.              $degree++$ 
8.         end if
9.     end for
10.    if M.containsKey(degree) then
11.        M.put(degree, M.get(degree)+1)
12.    else
13.        M.put(degree, 1)
14.    end if
15. end for
16. return M

```

---

the expected number of duplicated nodes, which means we can pick randomly  $\eta$  proteins in the family, copy each of them and add them to the family. Listing 3.3 shows the implementation of the method and the *pickPoisson* method generates the random number  $\eta$  for us with a given expected value.

Although gene death is the opposite of gene duplication, these two events are simulated in the same manner. They both act on the protein level so we can employ the same method that

---

**Listing 3.3** doDuplication(list, duplicationRate)

---

**Precondition:** duplicationRate is the number of iterations needed for duplication to occur on a protein,  $1/\text{duplicationRate}$  is the rate of duplication

**Precondition:** getRandom(int i) is a method to get a random number from 0 to i-1

**Precondition:** clone is a method that would make a copy of a sequence

1. *expectedDupSequence* = *list.size/duplicationRate*
  2. *actualDupSequence* = *pickPoisson(expectedDupSequence)*
  3. **for** *i* = 0 to (*actualDupSequence* - 1) **do**
  4.     *duplicatedString* = *list[getRandom(list.size())].clone*
  5.     *list.add(duplicatedString)*
  6. **end for**
- 

---

**Listing 3.4** doGeneDeath(list, deathRate)

---

**Precondition:** deathRate is the number of iterations needed for a gene death to occur,  $1/\text{deathRate}$  is the rate of gene death

**Precondition:** getRandom(int i) is a method to get a random number from 0 to i-1

1. *expectedDeadSequence* = *list.size/deathRate*
  2. *actualDeadSequence* = *pickPoisson(expectedDeadSequence)*
  3. **for** *i* = 0 to (*actualDeadSequence* - 1) **do**
  4.     *list.remove(getRandom(list.size()))*
  5. **end for**
- 

we used when simulating duplication. First, we pick  $n/k$  Poisson distributed random proteins in the family where  $1/k$  is now the rate of gene death. For each of these proteins, we will remove it from the protein family.

Unlike duplication and death, mutations do not occur on the protein level but on the nucleotide level. Rather than affecting a protein, a mutation can affect any nucleotide on any protein in the family. One can think of a protein family as a big string created by concatenating all sequences one after another and mutations can occur on any character in this combined string. Like the duplication case, we can traverse all the positions in the combined string and check whether a mutation occurs or not. We can improve the time complexity by just picking a Poisson distributed random number of nucleotides affected as we did in duplication. The performance gain by using the improved method is significant as we are examining strings with length on the order of 1000 and the number of strings is 500. Hence, we would have 500000 positions to consider if we use the traversing method. After knowing how many mutation positions there are in the concatenated strings (let this number be  $x$ ), we will pick randomly  $x$  positions on the combined string and apply the mutation. If we are doing a point mutation, the character at the picked position will be replaced by one of the characters different from the current character.

For insertion, a random character from the alphabet {A,C,G,T} will be added at the chosen site. Lastly, for deletion, the selected character will be removed from the string. However, the concept of a concatenated string only helps visualizing the process, it is not practical in implementation because we would have to keep a huge concatenated string whenever we need to do mutations. Besides, the combination string does not have any information to link back to each individual string, so a change in the combined string is not reflected in the members of the family. To solve this problem, we devise a data structure that lets us do mutations in a set of proteins (see Section 3.1.2). This data structure enables us to create methods that can perform point mutations, deletions and insertions at specified places in the combined string and link that change to the corresponded protein.

As we have discussed in Chapter Two, we use one cumulative rate for all types of mutation. As the portions for deletion and insertion are equal, we pick 0.25 for both of them and 0.5 for point mutation (as point mutation is the most popular type) to simplify the model.

---

**Listing 3.5** doMutation(list, mutationRate)

---

**Precondition:** mutationRate is the number of iterations needed for a mutation to occur on one nucleotide

```

1. expectedTotalMutation = list.getTotalLength/mutationRate
2. pointMutation = pickPoisson(expectedTotalMutation/2)
3. insertion = pickPoisson(expectedTotalMutation/4)
4. deletion = pickPoisson(expectedTotalMutation/4)
5. for i = 0 to (pointMutation - 1) do
6.     changeAt(pickRandom(list.getTotalLength))
7. end for
8. for j = 0 to (insertion - 1) do
9.     insertAt(pickRandom(list.getTotalLength))
10. end for
11. for k = 0 to (deletion - 1) do
12.     deleteAt(pickRandom(list.getTotalLength))
13. end for

```

---

### 3.1.2. A Protein Family Data Structure.

3.1.2.1. *Description.* As discussed in the last section, we need a data structure that can represent the concatenated string. The data structure that we need should enable the following operations:

- (1) Addition: Add a new element to the structure
- (2) Removal: Remove a current element

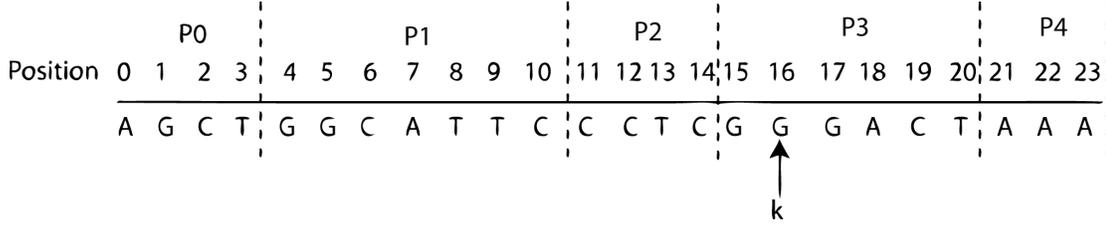


FIGURE 3.1. Example of Concatenated String with  $k = 16$ ,  $m = 3$  and  $i=1$

- (3) Insertion: Insert a character to a position on the concatenated string that this structure represents
- (4) Deletion: Delete a character in the concatenated string
- (5) Mutation: Change a character in the concatenated string

Mathematically, we can formalize the problem for this structure as follows.

Let  $P$  be the set of protein sequences in the family  $P = \{p_0, p_2, p_3, \dots, p_{n-1}\}$ .

Let  $s_i$  be the length of the string  $p_i$  and  $S$  be the string resulted from concatenating all strings in the family  $S = p_0 p_2 \dots p_{n-1}$ .

The data structure needs to represent  $S$  so that given a position  $k : 0 \leq k < \sum_{i=0}^{n-1} s_i$ , the data structure can return the index  $m$  and an integer  $i$  such that:

$$(3.8) \quad \sum_{j=0}^{m-1} s_j < k < \left( \sum_{j=0}^{m-1} s_j \right) + s_m$$

$$(3.9) \quad i = k - \sum_{j=0}^{m-1} s_j$$

In other words,  $m$  is the index of the string containing position  $k$  and  $i$  is the position on the  $m^{\text{th}}$  sequence corresponding to  $k$ . The problem is much easier if each sequence's length is equal to each other ( $s_1 = s_2 = \dots = s_{n-1}$ ). In this case,  $m = \lceil k/s_1 \rceil$  and  $i = k \% s_1$ . However, because of base insertions and deletions, the length of the sequences may not be the same. An important observation from the Equations 3.8 and 3.9 is that  $m$  and  $i$  are computed from the total length of strings to the left of the  $m^{\text{th}}$  string. This problem is similar to a range search so

the structure that we came up with is similar to a 1-dimensional search tree. This **tree-based** data structure satisfying the following conditions:

- 1.1 Each internal node only contains the sum of the leaves on its left subtree.
- 1.2 Each leaf node contains one of the sequence and its length and the total number of leaves is equal to the size of the family.
- 1.3 Each internal node always has two children.
- 1.4 The structure is filled in such a way that every level of the tree except for the last one is fully filled.

These conditions determine how we create the data structure. Condition 1.3 and 1.4 make this data structure similar to a binary heap in implementation. We can implement each node as an element of an array without using any pointers to its children. A node at position  $j$  in the array has two children at  $(2j + 1)$  and  $(2j + 2)$  and its parent is at  $(\lceil j/2 \rceil - 1)$ . In addition, Condition 1.2 makes sure that all sequences are stored in the leaf of this structure or in the last  $n$  positions of the array where  $n$  is the number of sequences. This feature is useful for duplication event as we can pick a random string in the last  $n$  elements of the array. All evolution operations need to preserve these conditions.

The two basic classes for this data structure are **ProteinNode** and **ProteinTree**. **ProteinNode** contains a string and an integer value. **ProteinTree** contains an array which is the representation of the tree. In all algorithm specifications, we refer to the string and value of the **ProteinNode** by **ProteinNode.string** and **ProteinNode.value** and to the array as **ProteinTree.array**.

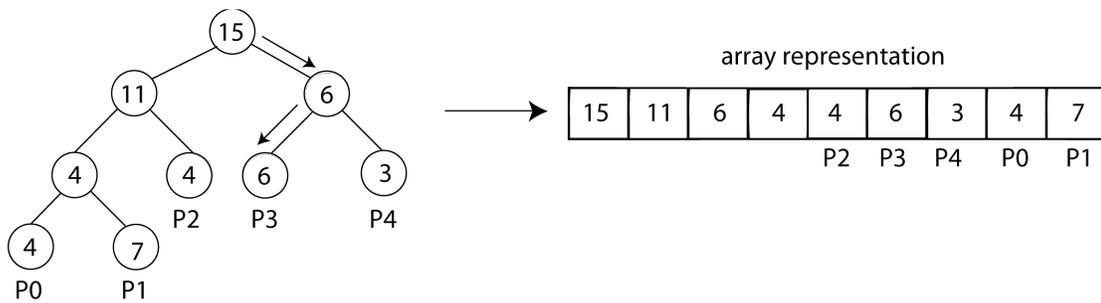


FIGURE 3.2. Representation of the concatenated string in Figure 3.1 and its array representation

3.1.2.2. *Addition.* Each time a new node is added, we can add it to the  $n^{\text{th}}$  position of the array to make sure that the tree is still nearly completely filled (Condition 1.4) so its parent is  $(\lceil n/2 \rceil - 1)$ . However, if we just add this new sequence to the array like this, one of the strings is still stored in an internal node  $(\lceil n/2 \rceil - 1)$  so we violate Condition 1.2 stating that all strings have to be stored in leaf nodes. For this reason, we will duplicate the node  $(\lceil n/2 \rceil - 1)$ , add it as the left child of itself and add the new node as the right child. The string in the node  $(\lceil n/2 \rceil - 1)$  will be moved to its left child. As we create two children for the node, Condition 1.3 is satisfied. For Condition 1.1, we need to update  $(\lceil n/2 \rceil - 1)$ 's parent. We will traverse up the tree, every time we see the current node as a left child, we increment its parent's value by the length of the added string. Note that Listing 3.3 invokes the *add* method with a string while the implementation of *add* (Listing 3.6) requires a *proteinNode*. This mismatch can be solved by creating a *ProteinNode* from the string and use Listing 3.6 method to add that new node. An important point to conclude from the discussion of the method is that the resulting tree satisfies all the conditions.

---

**Listing 3.6** add(s)

---

**Precondition:** s is the new *ProteinNode* to add

**Return:** s is added to the protein list such that all conditions are preserved

```

1. if ProteinTree.array.size = 0 then
2.   ProteinTree.array.add(s)
3. else
4.   nextAvailableIndex = ProteinTree.array.size
5.   parentIndex =  $\lceil \text{nextAvailableIndex}/2 \rceil - 1$ 
6.   ProteinTree.array.add(array[parentIndex])
7.   array[parentIndex].string = NULL {parentIndex is no longer leaf so value is NULL}
8.   ProteinTree.array.add(s)
9.   updateParent(parentIndex, s.length)
10. end if

```

---

3.1.2.3. *Removal.* As we have to keep the “nearly complete binary tree” shape as in Condition 1.4, we cannot just remove the element from the array. First, if the node to remove is the last node  $((n - 1)^{\text{th}}$  node) then to keep the shape, we also have to remove its sibling  $((n - 2)^{\text{th}}$  node). We still want to store the information in this sibling node somewhere. We can do this by putting the sibling node's string and value into its parent  $(\lceil (n - 1)/2 \rceil - 1)$  and then removing the sibling. The  $(\lceil (n - 1)/2 \rceil - 1)$  node becomes a leaf so it should contain the string value. Because the deletion may change the node value, we have to update all the parents' values

---

**Listing 3.7** updateParent(start, k)

---

**Precondition:** start is the index of the index where we start the updating process**Precondition:** k is the length to be added**Return:** Every node in the path from the start to the root that has the start node in the left subtree will be increased by k

```

1. t = start
2. parent =  $\lceil t/2 \rceil - 1$ 
3. while parent  $\geq 0$  do
4.     ProteinTree.array[parent].value = ProteinTree.array[parent].value + k
5.     t = parent;
6.     if (t%2) = 1 then
7.         parent =  $\lceil t/2 \rceil - 1$  {t is left child of parent if it is not divisible by 2}
8.     end if
9. end while

```

---

with the method in Listing 3.7. The case that the removed node is not the last node is more complicated. We do not delete the node at the removal position but we replace it by the last node and update all the parent nodes. The shape of the tree does not change until this time so all conditions still hold. The last node cannot appear twice in the structure, so we have to remove it, which is similar to the first case (we have to remove the last node and its sibling). Notice that the removal process does not simply mean removing an element from the array but it also involves a rearrangement in the array.

---

**Listing 3.8** remove(i)

---

**Precondition:** i is the index of the sequence to be removed**Return:**  $i^{th}$  element value after the call is the last element value of the array before the call or it is removed

```

1. removeLength = ProteinTree.array[i].value
2. parent =  $\lceil (i)/2 \rceil - 1$ 
3. if i = (ProteinTree.array.size - 1) then
4.     ProteinTree.array.remove(i)
5.     ProteinTree.array[parent].string = ProteinTree.array[i-1].string
6.     ProteinTree.array.remove(i-1)
7.     updateParent(parent, 0-removeLength);
8. else
9.     ProteinTree.array[i] = ProteinTree.array[ProteinTree.array.size - 1]
10.    updateParent(i, ProteinTree.array[ProteinTree.array.size - 1].value - removeLength)
11.    remove(ProteinTree.array.size - 1);
12. end if

```

---

3.1.2.4. *Insertion.* For this operation, we are given a position  $k$  on the concatenated string and need to find the  $m$  index of the string containing the position as in Equation 3.8. The method that we use here is similar to a binary search. As each node stores the sum of the

leaves in its left subtree, if the index given ( $k$  in Equation 3.8) is greater than or equal to the value of the node we are looking at then we should search in its right subtree for a value  $k = k - \text{array}[\text{currentIndex}]$ . Otherwise, we search in the left subtree for  $k$ . We do this until we reach a leaf node. The index of that leaf node is the value of  $m$  that we are looking for. This provides a solution for finding which ProteinNode affected by an insertion. Then, we can insert a random letter into that position (one among the four letters  $\{A,C,G,T\}$ ).

---

**Listing 3.9** insertAt( $k$ , start)

---

**Precondition:**  $k$  is an index in the concatenated string to insert

**Precondition:** start is the index where we start searching, for recursive call. Generally we assign start to 0 to search from the root node

**Return:** A random letter from the alphabet  $\{A,C,G,T\}$  will be added to the  $i^{\text{th}}$  position of the  $m^{\text{th}}$  string as in Equation 3.8 and 3.9

```

1. if start is leafNode then
2.     if  $k < \text{ProteinTree.array}[\text{start}].\text{value}$  then
3.          $c$  is a random character chosen from  $\{A,C,G,T\}$ 
4.          $\text{ProteinTree.array}[\text{start}].\text{string.insert}(\text{index}, c)$  {insert character  $c$  to a string at a
           location}
5.          $\text{ProteinTree.array}[\text{start}].\text{value}++$ 
6.     end if
7. else
8.     if  $k \geq \text{ProteinTree.array}[\text{start}].\text{value}$  then
9.          $\text{insertAt}(k - \text{ProteinTree.array}[\text{start}].\text{value}, (\text{start}+1)*2)$ 
10.    else
11.         $\text{ProteinTree.array}[\text{startIndex}].\text{value}++$ 
12.         $\text{insertAt}(k, (\text{start}+1)*2-1)$ 
13.    end if
14. end if

```

---

3.1.2.5. *Deletion.* A deletion works almost exactly the same way as an insertion except for a special case when the last character of a sequence is deleted. In this case, a deletion becomes a sequence removal. Otherwise, we apply the same method as in insertion to find the location of the letter to delete and remove it from the string.

3.1.2.6. *Point Mutation.* The point mutation method is similar to the insertion method. Instead of inserting a random character to the position, point mutation changes the character at that position to a different character in the alphabet  $\{A,C,G,T\}$ .

3.1.2.7. *Complexity.* For *add* (Listing 3.6), the worst case scenario is a non-empty tree. We have to make a duplicate of a current node and add both nodes to the structure. If we consider the array as an array list then we can get an amortized constant running time for these two

---

**Listing 3.10** deleteAt(k, start)

---

**Precondition:** k is an index in the concatenated string to delete

**Precondition:** start is the index where we start searching, for recursive call. Generally we assign start to 0 to search from the root node

**Return:** The  $i^{th}$  position of the  $m^{th}$  string as in Equation 3.8 and 3.9 will be removed

```

1. if start is leafNode then
2.     if k < ProteinTree.array[start].value then
3.         if Protein.array[start].value = 1 then
4.             remove(start)
5.         else
6.             ProteinTree.array[start].string.delete(index) {delete character at a location in
7.                 the string}
8.             ProteinTree.array[start].value = ProteinTree.array[start].value - 1
9.             updateParent(startIndex, -1)
10.        end if
11.    end if
12. else
13.     if k ≥ ProteinTree.array[start].value then
14.         deleteAt(k - ProteinTree.array[start].value, (start+1)*2)
15.     else
16.         deleteAt(k, (start+1)*2-1)
17.     end if
18. end if

```

---



---

**Listing 3.11** changeAt(k, start)

---

**Precondition:** k is an index in the concatenated string to change

**Precondition:** start is the index where we start searching, for recursive call. Generally we assign start to 0 to search from the root node

**Return:** The  $i^{th}$  position of the  $m^{th}$  string as in Equation 3.8 and 3.9 will be changed to a new character in the alphabet {A,C,G,T}.

```

1. if start is leafNode then
2.     if k < ProteinTree.array[start].value then
3.         ch = ProteinTree.array[start].string.charAt(index)
4.         c is a random character chosen from {A, C, G, T} \ {ch}
5.         ProteinTree.array[start].string.setChar(index, c) {set character at a location to c}
6.     end if
7. else
8.     if k ≥ ProteinTree.array[start].value then
9.         changeAt(k - ProteinTree.array[start].value, (start+1)*2)
10.    else
11.        changeAt(k, (start+1)*2-1)
12.    end if
13. end if

```

---

additions. Then we have to consider the complexity of *updateParent* (Listing 3.7). As we are going up the tree till we reach the root and this is a binary tree, we have to go through  $\lg n$  steps or the depth of the tree. In each step, we are only doing a constant time operation to add

a number to the node so we do constant time operation. In total, we have  $O(\lg n)$  complexity for *updateParent*. The total running time for the worst case scenario for *add* is  $O(\lg n)$ . The best case for *add* is when we have an empty tree, in which case we only add an element to an empty array, which takes constant time.

The *remove* method takes around the same time as addition. In the worst case, we have to do the rearrangement and deletion of the last element. The rearrangement consists of a constant time operation to copy a value from the last node and an  $O(\lg n)$  operation for *updateParent*. Deleting the last element involves deleting itself and its sibling, and updating the parents. We are deleting the last element of the array so it takes constant time. Hence, the worst case would take  $O(\lg n)$  time. The best case for this algorithm is removing the last element, which as we have seen takes  $O(1)$  time.

For the other three operations, it is more complicated because the complexity is now dependent on the string operations. As we pick the same portion for deletion and insertion (see Section 3.1.1), the length of the string will be in the order of the start string's length which we denote by  $s$ . For the point mutation method, it recursively calls itself on either the left subtree or the right subtree until it reaches a leaf. Each time it calls itself, the number of nodes decreases by a half just like a binary search. So it takes  $O(\lg n)$  time to do this search. At the leaf, the algorithm changes the letter at the  $k^{\text{th}}$  position to a new random letter. Changing a random letter in a string would take  $O(1)$  time. The whole method therefore takes  $O(\lg n)$  time.

The analysis is different for *insertAt* method because we have to do some work updating the nodes when we make the recursive call. In the base case, the algorithm does a character insertion to the string so that takes  $O(s)$  time. Hence we have the following recurrence:

$$(3.10) \quad T(1) = O(s)$$

$$(3.11) \quad T(n) \leq T(n/2) + c.$$

As a result, the running time for *insertAt* is  $O(\lg(n)) + O(s)$ .

The *deleteAt* method would have the same analysis as *changeAt* if it does not delete a string from the array. If the *deleteAt* method happens to invoke the *remove* method, it would take

$O(lgn)$  time when it reaches the leaf. Otherwise, it just deletes a character which takes  $O(s)$  time and it has to call *updateParent* which takes  $O(lgn)$  time. In total, the time complexity for *deleteAt* function in this case is  $O(lgn) + O(s)$  time. As a result, the worst case running time is  $O(lgn) + O(s)$ .

TABLE 3.1. Summary of Sequence Alignment approach’s complexity

Method	Complexity
Add	$O(lgn)$
Remove	$O(lgn)$
ChangeAt	$O(lgn)$
InsertAt	$O(lgn) + O(s)$
DeleteAt	$O(lgn) + O(s)$

**3.1.3. Sequence Alignment.** As indicated in the description of the experiment, after we get the protein family, we conduct a pairwise alignment for each pair of proteins. Although Salemi points out: “amino acid Sequence Alignments are easier to carry out and less ambiguous than nucleotide alignments” [24], we argue that nucleotide is preferable for our research. As mutations work on the nucleotide level, it is better to compare the nucleotide sequences to see how mutations have changed the proteins. Ultimately, since we use sequence similarity to determine relationship, using amino acid sequences would give us some error. Many sets of codons may encode one amino acid so even if two amino acids are similar, it is not certain that their nucleotide sequences are the same. For these reasons, we will use nucleotide sequences for comparisons.

In this research, we use **Needleman-Wunsch algorithm (NW)** to perform an alignment between two strings. We need to specify the cost of a gap (a difference resulting from insertions or deletions) and a mismatch (a difference resulting from mutations). For simplicity, we choose 1.0 for both of these costs. NW is a **dynamic programming algorithm** that explores all possible ways to convert one string to another and reports the path with the lowest cost. We use one for both of the costs, which makes the total cost the most probable number of string mutations, which is also the proxy that we use to determine relationship. We do not implement NW algorithm ourselves but instead we use **Neobio**, an open source Java library of bioinformatics algorithms [8].

When it comes to calculate how related two sequences are, the correct quantity to use is the number of mutations separating them. We have been saying that this number can be approximated by the score returned from the alignment. However, one problem with this approach is the effect of multiple hits in which several mutations can occur at one position of a string. It is possible that Sequence Alignment reports one mutation when in fact more than one mutation occurred. A **nucleotide substitution model** can fill this gap between the actual number of mutations and the number reported by Sequence Alignment. The simplest possible example of such models is **Jukes and Cantor model (JC)** [24]. This model assumes an equal rate to change from one base to another resulting in the substitution rate as shown in Table 3.2. From this setup, the model can calculate the number of mutation events that have occurred by Equation 3.12:

TABLE 3.2. Jukes and Cantor Substitution

	A	C	G	T
A	-3/4	1/4	1/4	1/4
C	1/4	-3/4	1/4	1/4
G	1/4	1/4	-3/4	1/4
T	1/4	1/4	1/4	-3/4

$$(3.12) \quad d = -\frac{3}{4} \ln \left( 1 - \frac{4}{3} p \right)$$

where  $d$  is the number of actual mutations and  $p$  is the number observed by Sequence Alignment [24].

Our *changeAt* method changes one base to a random base of the three possible bases to conform to the JC model. Equation 3.12 converts the distance returned from NW to the JC corrected distance before comparing with the cutoff value. Specifically, the method *SequenceAlignment* in Listing 3.2 computes the JC corrected distance.

**3.1.4. Complexity.** Before finishing the description of this Sequence Alignment approach, we want to make some remarks about the complexity of this algorithm. The algorithm can be broken into two parts which are the simulation and the Sequence Alignment part. For the simulation part, which corresponds to Listing 3.1 except for the call to `constructDistribution`, it

is an iterative program that keeps executing evolution events until the family reaches an upper limit size. Each of these evolution events are reduced to a data structure operation which is at most  $(O(\lg n) + O(s))$  time. As we discussed before, the duplication rate should be higher than the death rate and most of the time, we use a duplication rate that is twice the death rate. Therefore, we expect a near exponential growth for the family size. Our simulation runs until the family size reaches a limit number so the number of time steps should be  $O(n)$  where  $n$  is the final size of the family. Hence, the simulation part runs in  $O(n(\lg n + s))$  time. The Sequence Alignment part depends on the time complexity of the Needleman-Wunsch algorithm. The running time for the algorithm is  $O(s^2)$ . Then we have to iterate over all each pair of proteins in the family making the total running time for this part  $O(n^2) \times O(s^2)$ . Hence, the Sequence Alignment part dominates the simulation in running time. If we want to make the algorithm run faster, we need find a better way to find the relationship between two strings. This improvement is implemented in the next approach that we take, the Mutation Count approach.

### 3.2. Mutation Count Approach

**3.2.1. Simulation Description.** This approach is similar to the Sequence Alignment except in how we determine the relationship between two sequences. The previous approach uses the JC model to approximate the real number of mutations from the Sequence Alignment score. An alternate method is to keep track of the number of mutations occurring for each protein. Two proteins' distance is measured as the sum of their mutations from a common ancestor. In our case, as we do the simulation, we can record the mutations on a sequence. This number is recorded in a data structure during the evolution process (see Section 3.2.2).

Following the same path as the previous method, the algorithm includes two parts: the simulation and the distribution construction. The simulation is mostly the same as in Sequence Alignment approach. It is an iterative loop that executes the various evolution events. Each time that a mutation event occurs (a point mutation, deletion or insertion), the mutations counter on the sequence has to increase. The data structure storing the sequence will take care of this mutation recording. Listing 3.1 can still be used as the main program for this approach but all the evolution events are implemented differently. For the distribution construction part, we can

optimize by finding the degree for each sequence without having to iterate through each pair of strings. We will explore this idea in the description of the data structure.

**3.2.2. Evolution Data Structures.** For this Mutation Count approach, we need two data structures. First, as we still need to perform the simulation, we need the ProteinTree structure (in section 3.1.2). However, we no longer use Sequence Alignment implying that we do not need to know how the mutations might change the strings. Therefore, we only store the length of the string in the ProteinTree's node, not the string itself. Each of the operations that we described in the previous chapter now no longer have to alter the string; it just needs to update the length. For example, a base deletion is now reduced to a decrease in the sequence's length by one. Besides, we develop a tree structure (referred to as **EvolutionTree**) in which each node  $n$  stores the number of mutations that have happened since its creation.  $n$  does not store the sequence but instead, it stores a reference to a node  $k$  in the ProteinTree. Only the leaves can have these references. In other words, like the ProteinTree, we only store the sequence's information in the leaves. Conversely,  $k$  also stores a reference to  $n$  as defined in Listing 3.13. We implement the EvolutionTree as a binary tree so every internal node of the tree has two children. Each node in the tree is defined as in Listing 3.12. The last two boolean fields (isAlive and isVisited) are meant for constructing the node distribution and simulating evolution operations. The data structure must support the following operations:

- (1) Duplication: duplicate a current element.
- (2) Removal: remove a current element.
- (3) Mutation: change a position on the concatenated string.
- (4) Deletion: delete a position on the concatenated string.
- (5) Insertion: insert at a position on the concatenated string.
- (6) Calculate the degree: calculate how many elements an element is connected to.

The data structure also needs to satisfy the following conditions:

- 2.1 Each internal node contains the number of mutations occurring on its referenced ProteinNode from the time it was created to the time that its children are added.
- 2.2 Each internal node always has two children.

2.3 Each leaf node's Mutation Count is the number of mutations that has occurred on its referenced ProteinNode since the time the leaf was created

2.4 Only the leaves can contain a reference to a ProteinNode

These conditions ensure that our method for constructing the node distribution works properly so they must be satisfied throughout the entire evolution process.

---

**Listing 3.12** class EvolutionNode

---

1. numMutation
  2. parent
  3. leftChild
  4. rightChild
  5. protein {An element in the family tree}
  6. isAlive
  7. isVisited
- 

---

**Listing 3.13** class ProteinNode

---

1. evolutionNode {An element in the evolution tree}
  2. value {The length of the sequence}
- 

3.2.2.1. *Duplication.* Similar to duplication in ProteinTree, we first have to calculate how many ProteinNodes to duplicate by using Poisson random number generator. For each node to duplicate  $k$ , we create a copy  $k'$  of it and we also have a reference to a node  $m$  in the EvolutionTree. We define the duplication of  $m$  as  $m$  generating two replicas,  $m_1$  and  $m_2$ , of itself. We add these replicas as  $m$ 's children;  $m_1$  is the left child and  $m_2$  is the right child. We can think of these two replicas as one being the old node and one being the newly duplicated node. As  $m$  is no longer a leaf, it cannot store the reference to  $k$  anymore.  $m_1$  and  $m_2$  are leaves so  $m_1$  stores a reference to  $k$  and  $m_2$  stores a reference to  $k'$ . As no mutations have occurred on  $m_1$  and  $m_2$ , their Mutation Count are zero.

This implementation has to ensure that all conditions are conserved. As we create two children for  $m$ , Condition 2.3 is satisfied. Before the duplication,  $m$  contains the number of mutations since the time it was created. After the duplication, its Mutation Count does not change so it is storing the number of mutations from its creation to the time its children are added. Our implementation satisfies Condition 2.1. The third condition is trivial. The two new nodes are just created and there are no mutations on their referenced ProteinNode yet,

its Mutation Count is zero. As we have set  $m$  reference to NULL and set  $m_1, m_2$  to correct ProteinNode, the last condition is also satisfied. As a result, duplication reserves all conditions.

---

**Listing 3.14** doDuplication(duplicationRate)
 

---

**Precondition:** duplicationRate is the number of iterations needed for a duplication to occur on a protein,  $1/\text{duplicationRate}$  is the rate of duplication

**Precondition:** ProteinTree.size returns its number of sequences

1.  $expectedDupSequence = ProteinTree.size/duplicationRate$
  2.  $actualDupSequence = pickPoisson(expectedDupSequence)$
  3. **for**  $i = 0$  to  $(actualDupSequence - 1)$  **do**
  4.      $protein = ProteinTree[getRandom(ProteinTree.getNumNodes())]$
  5.      $duplicate(protein)$
  6. **end for**
- 

---

**Listing 3.15** duplicate(protein)
 

---

**Precondition:** protein is the proteinNode to duplicate

1.  $dupProtein = protein.clone()$
  2.  $evolNode = protein.evolutionNode$
  3. \*\*\*\*\*
  4.  $leftDupNode$  is initiated as a new EvolutionNode
  5.  $leftDupNode.numMutation = 0$
  6.  $leftDupNode.parent = evolNode$
  7.  $evolNode.leftChild = leftDupNode$
  8.  $leftDupNode.isAlive = true$
  9.  $leftDupNode.protein = protein$
  10.  $protein.evolutionNode = leftDupNode$
  11. \*\*\*\*\*
  12.  $rightDupNode$  is initiated as a new EvolutionNode
  13.  $rightDupNode.numMutation = 0$
  14.  $rightDupNode.parent = evolNode$
  15.  $evolNode.rightChild = rightDupNode$
  16.  $rightDupNode.isAlive = true$
  17.  $rightDupNode.protein = dupProtein$
  18.  $dupProtein.evolutionNode = rightDupNode$
  19.  $evolNode.protein = NULL$
  20. \*\*\*\*\*
  21.  $familyTree.add(dupProtein)$
- 

3.2.2.2. *Gene Death.* This operation is quite simple to implement. Suppose that we have to remove a ProteinNode  $k$ .  $k$  points to an EvolutionNode  $m$ . As the protein has to be removed, we set  $m$ 's isAlive to false and its reference to ProteinNode to be NULL. At this time,  $k$  can be removed from the ProteinTree.  $m$  is not really removed from the structure, but its link to the ProteinTree is deleted. As all operations are done on the ProteinTree,  $m$  will not be touched

from now on so  $m$  stores the number of mutations that have ever occurred on it since its creation so Condition 2.3 is satisfied. As Gene Death does not alter the structure of the EvolutionTree, all remaining conditions still hold.

---

**Listing 3.16** doGeneDeath(deadRate)
 

---

**Precondition:** deadRate is the number of iterations needed for a gene death to occur on a protein,  $1/\text{deadRate}$  is the rate of dead

1.  $\text{expectedDeadSequence} = \text{familyTree.size}/\text{deadRate}$
  2.  $\text{actualDeadSequence} = \text{pickPoisson}(\text{expectedDeadSequence})$
  3. **for**  $i = 0$  to  $(\text{actualDeadSequence} - 1)$  **do**
  4.      $\text{index} = \text{getRandom}(\text{ProteinTree.getNumNodes}())$
  5.      $\text{remove}(\text{index})$
  6. **end for**
- 

---

**Listing 3.17** remove(index)
 

---

**Precondition:** index the index of the node to remove

1.  $\text{evolNode} = \text{ProteinTree}[\text{index}].\text{evolutionNode}$
  2.  $\text{evolNode.isAlive} = \text{false}$
  3.  $\text{evolNode.protein} = \text{NULL}$
  4.  $\text{ProteinTree.remove}(\text{index})$
- 

3.2.2.3. *Mutation.* Similar to the family tree, mutation is an umbrella term including all types of mutations: point mutation, insertion, deletion. We still use the same division rule: 0.25 for both insertion and deletion, 0.5 for point mutation.

---

**Listing 3.18** doMutation(mutationRate)
 

---

**Precondition:** mutationRate is the number of iterations needed for a point mutation to occur on one nucleotide

**Precondition:** ProteinTree.getTotalLength returns the total length of the concatenated string

1.  $\text{expectedTotalMutation} = \text{ProteinTree.getTotalLength}/\text{mutationRate}$
  2.  $\text{pointMutation} = \text{pickPoisson}(\text{expectedTotalMutation}/2)$
  3.  $\text{insertion} = \text{pickPoisson}(\text{expectedTotalMutation}/4)$
  4.  $\text{deletion} = \text{pickPoisson}(\text{expectedTotalMutation}/4)$
  5. **for**  $i = 0$  to  $(\text{pointMutation} - 1)$  **do**
  6.      $\text{doPointMutation}(\text{pickRandom}(\text{ProteinTree.getTotalLength}()))$
  7. **end for**
  8. **for**  $j = 0$  to  $(\text{insertion} - 1)$  **do**
  9.      $\text{doInsertion}(\text{pickRandom}(\text{ProteinTree.getTotalLength}()))$
  10. **end for**
  11. **for**  $k = 0$  to  $(\text{deletion} - 1)$  **do**
  12.      $\text{doDeletion}(\text{pickRandom}(\text{ProteinTree.getTotalLength}()))$
  13. **end for**
-

3.2.2.4. *Insertion.* Given a nucleotide position, this operation finds which protein is affected. This affected protein points to a `EvolutionNode` in the `EvolutionTree` that needs to be updated. The number of mutations on this `EvolutionNode` will increase by one. At this time, we can invoke insertion in the `ProteinTree`.

---

**Listing 3.19** `doInsertion(index)`

---

**Precondition:** `index` is the index on the concatenated string to add a letter

1. `proteinNode = ProteinTree.findNodeOf(index, 0)`
  2. `evolNode = proteinNode.evolutionNode`
  3. `evolNode.numMutation ++`
  4. `Proteintree.insertAt(index, 0)`
- 

3.2.2.5. *Deletion.* For this operation, we reduce the length of the `ProteinNode` and invoke the `ProteinTree`'s `deleteAt` method. As it is still a type of mutation, the Mutation Count in the affected `EvolutionNode` is incremented by one.

---

**Listing 3.20** `doDeletion(index)`

---

**Precondition:** `index` is the index on the concatenated string to delete

1. `proteinNode = ProteinTree.findNodeOf(index, 0)`
  2. `evolNode = proteinNode.evolutionNode`
  3. `evolNode.numMutation ++`
  4. `ProteinTree.deleteAt(index, 0)`
- 

3.2.2.6. *Point Mutation.* This operation is the simplest among all operations. The length of the string does not change so there is no need to change the `ProteinNode`. We do have to find out which `EvolutionNode` is referenced from the `ProteinNode`. Then, we add one to the number of mutations of that `EvolutionNode`.

---

**Listing 3.21** `doPointMutation(index)`

---

**Precondition:** `index` is the index on the concatenated string to mutate

1. `proteinNode = ProteinTree.findNodeOf(index, 0)`
  2. `evolNode = proteinNode.evolutionNode`
  3. `evolNode.numMutation ++`
- 

**3.2.3. Constructing the Degree Distribution.** Consider a node  $a$  in the tree whose children are leaves  $b$  and  $c$ . As the only way that a node is added to the family is by duplication, a duplication must have occurred on  $a$  and created the two nodes  $b$  and  $c$ . In the beginning,  $b$  and  $c$  started as replicas of  $a$  so they were exactly the same as each other. The number of

mutations on  $b$  or  $c$  indicates how many mutations have occurred since this beginning (Condition 2.3). Hence, the sum of  $b$ 's mutations count and  $c$ 's count is the number of mutations separating them. Generalizing this to two nodes  $b$  and  $c$  that have a common ancestor  $a$ , the sum of all children on the path from  $a$  to  $b$  is the number of mutations that transform  $a$  to  $b$ . If we combine this sum with the similar sum on the path from  $a$  to  $c$ , we have the total number of mutations separating  $b$  and  $c$  (or **mutation distance**).

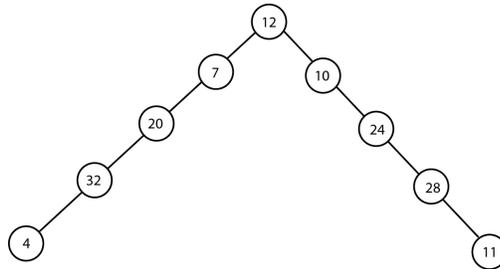


FIGURE 3.3. Example of calculating number of mutations, the distance between the 2 leaves is 136

If we follow the same mechanism as in the Sequence Alignment approach, we can iterate over all pairs  $(m, n)$  of leaves in the tree, find its nearest common ancestor and compute the mutation distance between  $m$  and  $n$ . If that distance is less than a cutoff, then  $m$  and  $n$  are related. The number of  $m$ 's related leaves is its degree. However, we can make some improvements in calculating a leaf's degree by using a depth first search. To better explain this method, we consider the EvolutionTree now as a graph where any node  $m$  in the graph has three possible neighbors: its parent, its left and right child. This graph's root is at a leaf  $k$  which we want to compute the degree. Let  $f(m, d, dir)$  be the function that returns the number of leaves related to  $k$  in the unvisited subgraph rooted at  $m$ , which is separated from  $k$  by  $d$  mutations. An unvisited subgraph is a subgraph whose root has not been visited by the search method before. The  $dir$  variable indicates the direction of the search, whether we are searching down or up the tree (1 for up the tree and 0 for down the tree). With these definitions, we would have the following recursive relation

$$(3.13) \quad f(m, d, dir) = f(m.parent, d + m_{mu}, 1) + f(m.leftChild, d', 0) + f(m.rightChild, d', 0)$$

if  $m$  is not visited where  $m_{mu}$  is the number of mutations in  $m$  and  $d'$  is an integer dependent on  $dir$ .

Base Cases:

$$(3.14) \quad f(m, d, dir) = 0 \text{ if } m \text{ is visited}$$

$$(3.15) \quad f(m, d, dir) = 0 \text{ if } d > \text{cutoff value}$$

$$(3.16) \quad f(m, d, dir) = \begin{cases} 1 & \text{if } m \text{ is a leaf and } m \text{ is alive and } d < \text{cutoff} \\ 0 & \text{if } m \text{ is a leaf and } m \text{ is not alive} \end{cases}$$

Equation 3.15 provides an improvement in running time over the method of comparing every pair of leaves. It states that anytime we see a node  $m$  whose distance from  $k$  is greater than the cutoff, then the number of related nodes in the unvisited subgraph is zero. The reason comes from the observation that the distance between any node  $n$  in the unvisited subgraph of  $m$  and  $k$  is the sum of the distance from  $k$  to  $m$  and from  $m$  to  $n$ . If the distance from  $k$  to  $m$  is already greater than the cutoff, then certainly this sum will be greater than the cutoff. Therefore, there are no nodes in  $m$ 's unvisited subgraph which can be related to  $k$ . From this equation, we do not have to search for every leaves in the tree but we just have to search until we find a node whose distance from  $k$  is greater than the cutoff.

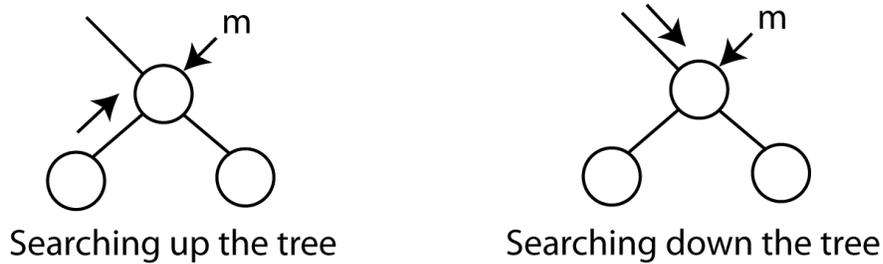


FIGURE 3.4. Example of searching direction

For the value  $d'$  in the Equation 3.13, its value depends on the variable  $dir$ . If we are at  $m$  and start searching its parent, the  $dir$  in the parent's call will be one meaning searching up

the tree. Conversely, if we are searching  $m$ 's children then  $dir$  is zero or down in the children's call. The value of  $dir$ , therefore, depends on the relative position between the previous searched node and  $m$ . If the previous node is  $m$ 's parent then  $dir = 0$  otherwise  $dir = 1$ . What is the importance of this direction of search? Consider the two cases in Figure 3.4. In the case of searching up the tree,  $k$  is in the left subtree of  $m$  (assume that  $m$  is not visited). So the distance between  $k$  and  $m$ 's right child is the sum of distance between  $k$  and  $m$  and the number of mutation on  $m$ 's right child. In the other case of searching down the tree, the distance between  $k$  and  $m$ 's children is the sum of  $k$ 's distance to  $m$  and the sum of number of mutations on  $m$  and  $m$ 's right child. Generalizing these two cases, we have the formulas for  $d'$ :

$$(3.17) \quad d' = \begin{cases} d + m.child.numMutation & \text{if } dir = 1 \\ d + m.numMutation + m.child.numMutation & \text{if } dir = 0 \end{cases}$$

where  $m.child$  can refer to either `leftChild` or `rightChild` depending on whether we finding the distance for right child or left child.

Using all equations from 3.13 to 3.17 we have the algorithm to compute the number of related nodes for  $k$  in the unvisited subgraph of  $m$  as specified in Listing 3.22.

Now, we can use the method `getRelatedNodes` to calculate the degree of a leaf  $k$  by starting the search from its parent  $m$ . The initial call has to calculate the distance from  $k$  to  $m$  which is the number of mutation in  $k$ . We also have to set up an array of `EvolutionNode` for `visitedNodes`. As we can see in Listing 3.22, every time we call the method on a node  $n$ , we set  $n$ 's `isVisited` field to true. As we may need to find the degree for more than one leaf, we have to set  $n$  back to unvisited state. This is the reason for keeping an array of visited nodes. In the end of  $k$ 's degree calculation, we just iterate over all nodes in `visitedNodes` and set them to unvisited.

To construct the degree distribution of the family, we need to calculate the degree for all the leaves in the evolution tree. Because of Condition 2.4, the `ProteinTree` stores the references to all leaves in its last  $n$  elements where  $n$  is the number of proteins in the family. Instead of looping over all leaves in the evolution tree, we will iterate through the last  $n$  elements of `ProteinTree`.

---

**Listing 3.22** getRelatedNodes(start, distance, visitedNodes, isGoingUp)

---

**Precondition:** start is the currently searched node

**Precondition:** distance is the distance that we have followed so far in the search

**Precondition:** visitedNodes is an array of nodes that have been visited

**Precondition:** isGoingUp is a boolean indicating whether the search is going up or down the tree

**Return:** the number of unvisited nodes that is related to the original node of the search

```

1. result = 0
2. start.isVisited = true
3. visitedNode.add(start)
4. if distance > cutoff then
5.     return 0
6. end if
7. if start is a leaf AND start.isAlive then
8.     if distance + start.numMutation ≤ CUTOFF then
9.         return 1
10.    else
11.        return 0
12.    end if
13. else
14.    if start.parent != NULL AND !start.parent.isVisited then
15.        newDistance = start.numMutation + distance
16.        result += getRelatedNodes(start.parent, newDistance, visitedNodes, true)
17.    end if
18.    if start.leftChild != NULL AND !start.leftChild.isVisited then
19.        newDistance = distance
20.        if !isGoingUp then
21.            newDistance = start.numMutation + distance
22.        end if
23.        result += getRelatedNodes(start.leftChild, newDistance, visitedNodes, false)
24.    end if
25.    if start.rightChild != NULL AND !start.rightChild.isVisited then
26.        newDistance = distance
27.        if !isGoingUp then
28.            newDistance = start.numMutation + distance
29.        end if
30.        result += getRelatedNodes(start.rightChild, newDistance, visitedNodes, false)
31.    end if
32. end if
33. return result

```

---

**3.2.4. Main Simulation Program.** With all the information regarding the data structure and the degree distribution construction, we can now combine them for the main simulation program as in Listing 3.25.

**3.2.5. Complexity.** The duplication method has two parts: one to duplicate the node in the EvolutionTree, and one to add a new node to the ProteinTree. Because the EvolutionTree

---

**Listing 3.23** getDegree(k)

---

**Precondition:** k is the node to compute the degree**Return:** the degree of k

1. visitedNodes initialized to a new array of EvolutionNode
  2. *k.visited* = true
  3. *visitedNodes.add(k)*
  4. *degree* = 0
  5. **if** *k.parent* != NULL **then**
  6.     *degree* = *getRelatedNodes(k.parent, k.numMutation, visitedNodes, true)*
  7. **end if**
  8. **for** *i* = 0 to *visitedNodes.size* - 1 **do**
  9.     *visitedNodes[i].isVisited* = false
  10. **end for**
  11. **return** *degree*
- 

---

**Listing 3.24** constructDistribution()

---

**Precondition:** ProteinTree.getNumNodes will return the number of sequences in the family**Return:** a Map *M* that maps a degree *d* to the number of sequence with that degree

1. *M* is initialized as a Map of type  $\langle integer, integer \rangle$
  2. **for** *i* = 0 to *ProteinTree.getNumNodes* - 1 **do**
  3.     

= ProteinTree[ProteinTree.size - Protein.getNumNodes + i]
  4.     *degree* = *getDegree(p.EvolutionNode)*
  5.     **if** !*M.containsKey(degree)* **then**
  6.         *M.put(degree, 1)*
  7.     **else**
  8.         *M.put(degree, M.get(degree) + 1)*
  9.     **end if**
  10. **end for**
- 

---

**Listing 3.25** doSimulation(duplicationRate, mutationRate, deathRate, cutoff)

---

**Return:** a map whose key is the degree and the value is how many nodes with that degree

1. ProteinTree initialized to an empty array
  2. rootNode initialized to a new EvolutionNode
  3. *rootNode.numMutation* = 0
  4. rootProtein initialized to a new ProteinNode
  5. *rootProtein.value* = *STRINGLENGTH*
  6. *rootProtein.EvolutionNode* = *rootNode*
  7. *rootNode.protein* = *rootProtein*
  8. *ProteinTree.add(rootProtein)*
  9. **while** *ProteinTree.getNumNodes()* < *EXPERIMENTLIMIT* **do**
  10.     doDuplication(duplicationRate)
  11.     doMutation(list, mutationRate)
  12.     doGeneDeath(list, deathRate)
  13. **end while**
  14. return constructDistribution()
-

is a binary tree, we always have to create two new `EvolutionNode` so the duplication part takes constant time. Adding a new node to the `ProteinTree`, as we discussed before, takes  $O(\lg n)$  time where  $n$  is the number of proteins in the family. In total, duplication time complexity is  $O(\lg n)$ . Similarly, the gene death method includes a constant time part (when we set the affected `EvolutionNode` to dead) and a  $O(\lg n)$  time part (when we remove the node from the `ProteinTree`). Therefore, gene death takes  $O(\lg n)$  time. Unlike the Sequence Alignment approach, mutations methods do not depend on the string operations complexity anymore. They all have a similar approach: finding the affected `ProteinNode` from the given index which is similar to a binary search and takes  $O(\lg n)$  time, invoking the related method in the `ProteinTree` (except for point mutation). The corresponding `ProteinTree` methods now do not manipulate the string so the time complexity is only  $O(\lg n)$ . For the two mutation methods, insertion and deletion, the time complexity is  $O(\lg n) + O(\lg n)$  which is  $O(\lg n)$ . Point mutation does not have to call the `ProteinTree` method, hence it only does a search and constant time work to update the length. The time complexity for it is  $O(\lg n)$ . Lastly, we consider the complexity of finding the degree for a leaf  $k$  in the `EvolutionTree`. In Listing 3.23, the significant part of this method is adding a node to `visitedNodes` array, calling `getRelatedNodes` and reset all the nodes in `visitedNodes`. The size of the `visitedNodes` array is bounded by the number of leaves in the `EvolutionTree` or the number of proteins  $n$ . Therefore, we can initialize `visitedNodes` to an array with  $n$  elements. Hence, addition to this array would never involve copying elements to a new array so the time complexity is  $O(1)$ . Resetting all elements in `visitedNodes` is  $O(n)$  time. What is left is the complexity of `getRelatedNodes`. Although in Listing 3.22, it recursively call itself three times, in a typical case, the number of recursive call is two as one of the branch is already visited. Consider the initial call in `getDegree` for  $k$ , the call is on  $k$ 's parent. When we execute `getRelatedNodes` for  $k$ 's parent we see that one of its two children is  $k$  which is already visited so the number of calls is two. Except for these recursive calls, the method does not do any other significant work resulting in the following recurrence:

$$(3.18) \quad T(n) = 2T(n/2) + c$$

$$(3.19) \quad T(n) = O(1)$$

TABLE 3.3. Summary of Mutation Count approach's complexity

Method	Complexity
Duplication	$O(lgn)$
Gene Death	$O(lgn)$
PointMutation	$O(lgn)$
Insertion	$O(lgn)$
Deletion	$O(lgn)$
Get Degree	$O(n)$

Solving this recurrence, the complexity is  $O(n)$ . Therefore, the complexity for `getDegree` is  $O(n)$ . This complexity is much faster than using Sequence Alignment to determine the degree.

### 3.3. Preferential Attachment Model

The two approaches we presented have not explicitly incorporated Preferential Attachment. In this section, we will present how we implement Preferential Attachment in our simulation. The basic idea is that before we execute any evolution operations, we will calculate the degree of all proteins in the family. When we pick a random node  $k$  to apply the operation, we compare  $k$ 's degree and a random cutoff value. If the degree is greater than the cutoff value then we re-pick another node, otherwise the operation occurs on the selected node. This method requires a frequent calculation of every node's degree implying that it would take a substantially long time if we use Sequence Alignment approach (computing degrees of  $n$   $s$ -length nodes requires  $O(n^2 \times s^2)$  time). In contrast, if we use Mutation Count approach, we get a lower time complexity for computing the degree. Each computation requires us to traverse up the tree from the node  $k$  to compute the degree to a node  $m$  whose distance from  $k$  is greater than the cutoff number of mutations. As the cutoff value is usually low, we do not have to traverse for too many nodes in the tree. Determining the degree can be much faster than in the case of Sequence Alignment. We will also show later the two methods are equivalent regarding the fits to the two distributions (see Section 4.4.2). Because of this improvement in running time, we will incorporate Preferential Attachment into the Mutation Count approach.

We do not have to modify the main program to fit in Preferential Attachment but we just have to change our operation method. For each method, we will simulate it as follows:

- (1) Loop through every protein in the family and find the degree of the protein using Listing 3.23 and record the maximum and minimum degree inside this loop.
- (2) Pick a random number in the range from the minimum to the maximum degree, this is the cutoff value we will use to determine Preferential Attachment.
- (3) Pick a random protein  $p$  to be affected by the operation.
- (4) If  $p$ 's degree is lower than the cutoff, proceed to step 5, otherwise go back to step 3.
- (5) Simulate the operation on the  $p$  as we would in Mutation Count approach.

The selection in steps 3, 4 and 5 ensure that the operation only occurs on a node with degree lower than the random cutoff we pick in step 2. Using these steps, we will have Preferential Attachment in all operations. The updated operations are describe in Listing 3.26 to 3.29.

---

**Listing 3.26** doPreferentialDuplication(duplicationRate)

---

**Precondition:** minDegree and maxDegree are global variables that hold the current maximum and minimum degree of the family

**Precondition:** degreeCutoff is a global variable holding the cutoff value for Preferential Attachment

**Precondition:** duplicationRate is the number of time steps needed for a duplication to occur

**Precondition:** EvolutionNode has a new property degree

1.  $expectedDupSequence = ProteinTree.size / duplicationRate$
  2.  $actualDupSequence = pickPoisson(expectedDupSequence)$
  3. **for**  $i = 0$  to  $(actualDupSequence - 1)$  **do**
  4.      $setupDegreeCutoff()$
  5.      $protein = ProteinTree[getRandom(ProteinTree.getNumNodes())]$
  6.     **while**  $protein.evolutionNode.degree > degreeCutoff$  **do**
  7.          $protein = ProteinTree[getRandom(familyTree.size())]$
  8.     **end while**
  9.      $duplicate(protein)$
  10. **end for**
- 

### 3.4. Poisson Random Variables

We can model all evolution events by iterating through some population (the list of all proteins or the list of all nucleotides) and asking if an event occurs on the current element by generating some random numbers. Although this method is simple, it may take too much time as we always have to go through the whole population. If we consider each time we ask as a trial, the outcome of each trial is either an event occurs if the result number is one or nothing happens if the result number is zero. As for each trial, we pick a new random number, all trials are independent of each other. These two points, by definition, make each trial a

---

**Listing 3.27** setupDegreeCutoff()

---

**Precondition:** minDegree and maxDegree are global variables that hold the current maximum and minimum degree of the family

**Precondition:** degreeCutoff is a global variable holding the cutoff value for Preferential Attachment

**Precondition:** EvolutionNode has a new property degree

**Precondition:** pickRandom(i,j) returns a random number in the range from i to j

```

1. maxDegree = 0
2. minDegree = 0
3. for i = 0 to ProteinTree.getNumNodes - 1 do
4.   degree = getDegree(ProteinTree[i].evolutionNode)
5.   ProteinTree[i].evolutionNode.degree = degree
6.   if degree > maxDegree then
7.     maxDegree = degree
8.   end if
9.   if degree < minDegree then
10.    minDegree = degree
11.  end if
12.  degreeCutoff = pickRandom(minDegree, maxDegree)
13. end for

```

---



---

**Listing 3.28** doPreferentialGeneDeath(deadRate)

---

**Precondition:** deadRate is the number of iterations needed for a gene death to occur on a protein, 1/deadRate is the rate of dead

```

1. expectedDeadSequence = familyTree.size/deadRate
2. actualDeadSequence = pickPoisson(expectedDeadSequence)
3. for i = 0 to (actualDeadSequence - 1) do
4.   setupDegreeCutoff()
5.   index = getRandom(ProteinTree.getNumNodes())
6.   while ProteinTree[index].evolutionNode.degree > degreeCutoff do
7.     index = getRandom(ProteinTree.getNumNodes())
8.   end while
9.   remove(index)
10. end for

```

---

**Bernoulli trial.** Each evolution event simulation is a collection of such trials so it follows a **Binomial distribution**. This distribution has an interesting property: it converges to a **Poisson distribution** as the number of trials increases and probability of success decreases. This approximation is reasonable for more than 20 trials and the success rate for each is less than 0.05 [9]. The rate of evolution events that we use are really low (all of them are below 0.01) and the size of the population are usually high (the number of genes in a family or the number of bases in a genes). Hence, we can apply this method to improve the running time as follows. Suppose there are  $n$  elements in the population and the probability of the evolution event on

---

**Listing 3.29** doPreferentialMutation(mutationRate)

---

**Precondition:** mutationRate is the number of iterations needed for a point mutation to occur on one nucleotide

**Precondition:** ProteinTree.getTotalLength returns the total length of the concatenated string

```

1. expectedTotalMutation = ProteinTree.getTotalLength/mutationRate
2. pointMutation = pickPoisson(expectedTotalMutation/2)
3. insertion = pickPoisson(expectedTotalMutation/4)
4. deletion = pickPoisson(expectedTotalMutation/4)
5. for i = 0 to (pointMutation - 1) do
6.   setupDegreeCutoff()
7.   randomSite = pickRandom(ProteinTree.getTotalLength())
8.   while ProteinTree.findNodeOf(randomSite).evolNode.degree i degreeCutoff do
9.     randomSite = pickRandom(ProteinTree.getTotalLength())
10.  end while
11.  doPointMutation(randomSite)
12. end for
13. for j = 0 to (insertion - 1) do
14.  setupDegreeCutoff()
15.  randomSite = pickRandom(ProteinTree.getTotalLength())
16.  while ProteinTree.findNodeOf(randomSite).evolNode.degree j degreeCutoff do
17.    randomSite = pickRandom(ProteinTree.getTotalLength())
18.  end while
19.  doInsertion(randomSite)
20. end for
21. for k = 0 to (deletion - 1) do
22.  setupDegreeCutoff()
23.  randomSite = pickRandom(ProteinTree.getTotalLength())
24.  while ProteinTree.findNodeOf(randomSite).evolNode.degree k degreeCutoff do
25.    randomSite = pickRandom(ProteinTree.getTotalLength())
26.  end while
27.  doDeletion(randomSite)
28. end for

```

---

each element is  $1/k$ , the expected number of events occurring is  $n/k$ . The actual number of events will follow a Poisson distribution with the expected value  $n/k$  so it can be generated as a Poisson generated random number  $\eta$  with this expectation. Instead of traversing the population, we can pick randomly  $\eta$  elements and apply the event on each of them. We do not implement the Poisson number generation but instead we use a Java package **Flanagan** [13] as the algorithm this package provides has a good running time for high value means.

### 3.5. Timing Information

In this section, we include a comparison of the running of the three models that we have proposed. Their running time is listed in Table 3.4. The listed running time is determine based on a run with the following parameters:

- (1) Family size 500
- (2) Protein length 1000
- (3) Duplication Rate =  $1/160000$
- (4) Death Rate =  $1/320000$
- (5) Mutation Rate =  $1/2000000$
- (6) Cutoff Rate = 0.07

Changing the value of these parameters may affect the running time. For example, allowing more mutations to occurs would increase the running time. Since the protein length that we chose is much bigger than the family size, the time complexity  $O(n^2 \times s^2)$  of Sequence Alignment would be approximately  $O(n^4)$ . The Sequence Alignment approach takes significantly much more time than the other two models. The Sequence Alignment takes around 5000 times the running time of Mutation Count approach. The bulk of the workload for Sequence Alignment approach comes from the Distribution Construction so by improving this part's running time, the other two models get a much better running time. The other two models (the Mutation Count approach and the Preferential Attachment model) are based on the same method to quantify protein relationship. The Preferential Attachment model takes more time as it has to calculate the degree of each node in every operation, we expect it to take longer than the other model without Preferential Attachment. As suggested in Table 3.4, without Preferential Attachment, the Mutation Count approach takes significantly less time. The Preferential Attachment model takes very little time in constructing the degree distribution because we have already built this distribution in the simulation and we do not have to compute the distribution again in the end.

TABLE 3.4. Running time for the three models (time in seconds)

Model	Simulation	Construction	Total
Sequence Alignment Approach	7	34779	34786
Mutation Count	3	4	7
Preferential Attachment	490	1	491

## CHAPTER 4

# Results

### 4.1. Experimental Results

**4.1.1. Parameter Values.** Before getting into the description of the results, we will examine the parameters that we used in the simulation. The parameters can be divided into two groups: static and tunable. Static parameters are those that we do not change between one run and another including the length of each protein (set at 1000), the size of the family (set at 500) and the mutation rate (at  $1/2000000$ ). We fix the first two variables, gene length and family size, because of the Sequence Alignment. If we have too many proteins in the family or too long protein sequences, the Sequence Alignment may take a very long time to accomplish. Setting the size of the family and protein reduces the running time for the Sequence Alignment approach and also makes it easier to find a comparative family in real life because we know the size of the family. Also, the protein length fixed value is not an impractical assumption as the average gene is usually “assumed to be 1000 nucleotide pairs in length” [26]. The problem that we have is determining the rate of duplication and mutation for our simulation. As we have mentioned in Chapter Two, we pick a rate less than Snustad and Simmons’ suggested rate which is in the order of  $10^{-7}$  to  $10^{-9}$  per base pair per generation for gene mutation. As we are only considering a subset of mutations, the rate we use should be much less than  $10^{-7}$ . The value we choose is  $1/2000000$ .

The tunable variables in the simulation are:

- (1) Gene Duplication
- (2) Gene Death Rate
- (3) Cutoff

We can determine duplication rate by a ratio between the duplication and mutation rate. As we have discussed earlier, Gao and Inaan mentioned a ratio of 28 between the duplication and synonymous mutation rate [15]. In our mutation model, we divide mutation into three types:

point mutation, insertion and deletion among which only point mutation can be synonymous. We set point mutation ratio to be 0.5 so the rate of duplication may be:

$$(4.20) \quad 1/2000000 \times 0.5 \times 28 = 7 \times 10^{-6} \sim 1/140000.$$

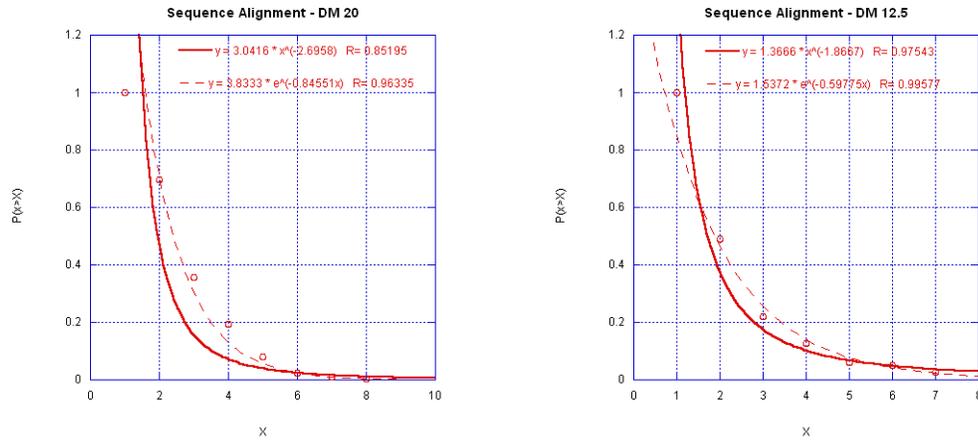
However, the result in Equation 4.20 only serves as an upper bound for the duplication rate as not all point mutations are synonymous. Therefore, we only know that duplication rate should be below 1/140000. We will examine duplication's relation with mutation in each model and observe how our result distribution might change by trying various **Duplication - Mutation (DM)** ratio. The DM ratio in our model is different from the one in Gao and Innan's paper. Our model compute the DM ratio by dividing the duplication rate by the composite mutation rate (not the neutral mutation rate as in Gao and Inaan's study [15]). We can convert our DM ratio to Gao and Innan's by dividing it by 2 since the neutral mutation portion in our model is 0.5. Gao and Inaan's ratio of 28 [15] is translated to a DM ratio of 14. Gene Death Rate is the number of iterations needed for one occurrence of gene death. Again, we will study the effect of **Duplication - Death (DD)** ratio on the data fit. Cutoff is the threshold value of Sequence Alignment score above which the proteins are not related. It is specified as the portion of the string that is different from each other. For example, a Cutoff of 0.1 means two proteins are not related if they are different in more than 10% of their length. The reason these parameters have to be tunable is because we do not know for certain which value is right. By changing the values, we can see how each parameter can affect the shape of the distribution. There may be a range of values in which the distribution is power law but outside that range, the distribution can be totally different. For each of the models, we conduct three studies to examine the relationship between the evolution rate and the R values. These studies are DM ratio and R value, Cutoff and R value and DD ratio and R value.

#### 4.1.2. Sequence Alignment Approach.

4.1.2.1. *DM Ratio and R value.* To study the DM ratio's effect on R values of the fits, we fix mutation rate at 1/2000000 and DD ratio at 2. The result is illustrated in Figure 4.1 and 4.2. The graphs in these two figures are typical of the type of graphs we will get in each studies we conduct. Each graph represent the result we have for one set of parameters and the R values

when fitting to power law and exponential distribution. As for each study we only change the values of one ratio or parameter, from these graphs, we can construct a graph illustrating the change of R values against the change of the examined variable (as in Figure 4.3). In later studies, we will not present the graph of each set of parameters (which can be found in the Appendix) but only the graph with R value against the variable.

FIGURE 4.1. The graphs and fitted curves for Sequence Alignment Approach with Variable DM ratio (the bold line is the power law curve fit and the dotted line is the exponential curve fit)

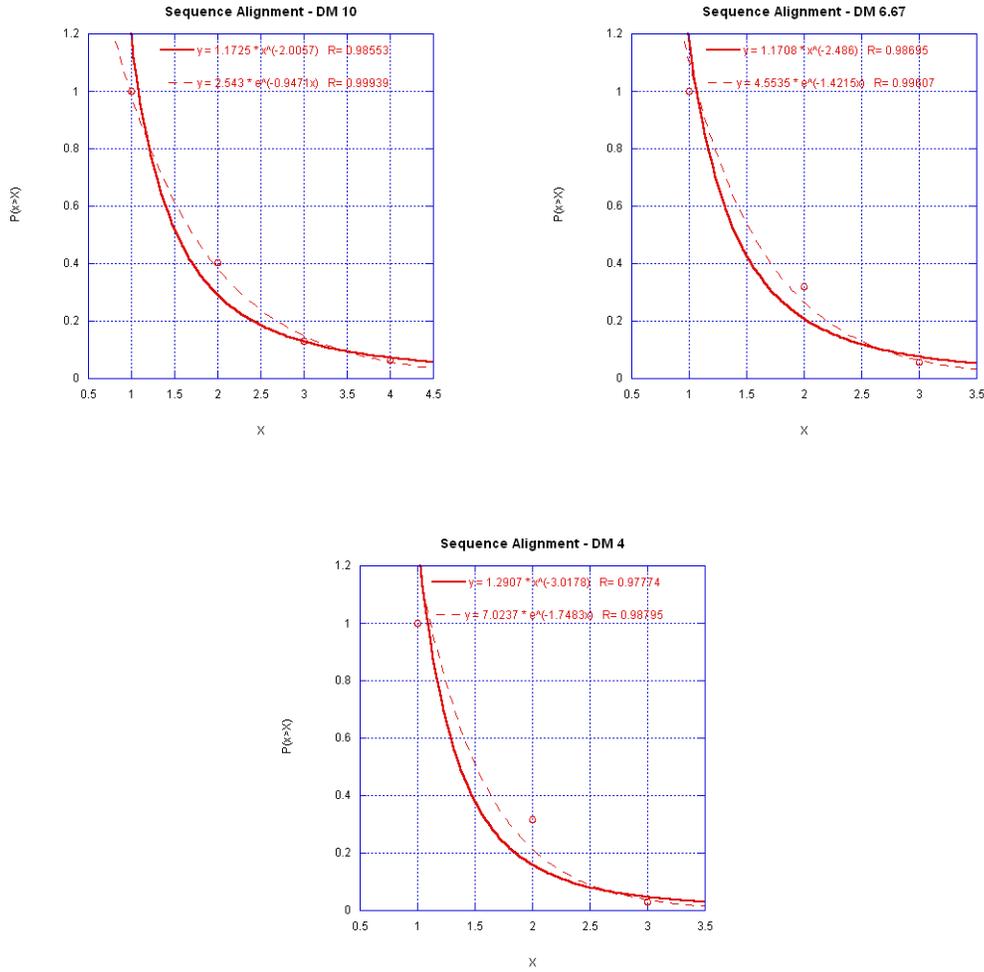


We only give DM ratio larger than 4 because a lower DM gives fewer data points in the Sequence Alignment approach. From the graph in Figure 4.3, we can see that exponential seems to be a good fit with R value always greater than 0.9 while power law's R value is good for DM below 12.5. This range includes the ratio of 14, which consistent with our calculation of the duplication rate in previous section. The trend for both distribution is similar: the R value for both distributions fit decreases as the DM ratio gets bigger or the duplication rate gets higher.

4.1.2.2. *Cutoff Value and R value.* In this section, we set the duplication at  $1/160000$  and death rate to be  $1/320000$ . The cutoff value is given many values to test for the best fit distribution.

The general trend, as suggested in Figure 4.4, is that when we increase the cutoff, the R values for both distribution decrease. Although exponential is not affected much as its R value always stays above 0.9, power law can only be a good fit for cutoff less than 0.1. The cutoff

FIGURE 4.2. The graphs and fitted curves for Sequence Alignment Approach with Variable DM ratio (cont)



has a large impact on the shape of the data as it determines the relationship between proteins. The Sequence Alignment shows that exponential distribution fits the data for a wider range of cutoffs than power law.

4.1.2.3. *DD ratio and R value.* Like the DM ratio, DD ratio may affect the goodness of fit of the distributions. We set duplication rate at  $1/80000$  and cutoff at 20 to see what happens to the R value when the Death Rate changes.

FIGURE 4.3. The relationship between DM ratio and R value for Sequence Alignment Approach

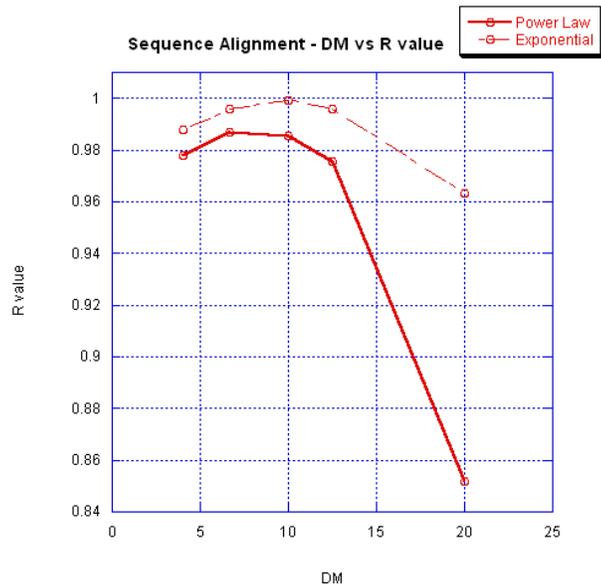


FIGURE 4.4. The relationship between cutoff and R value for Sequence Alignment Approach

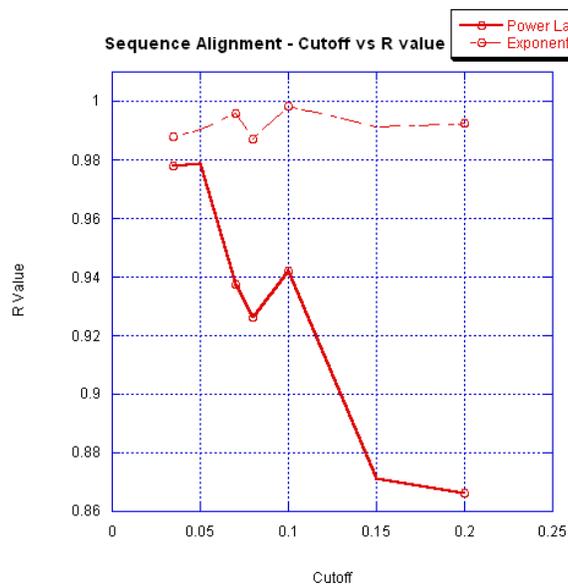
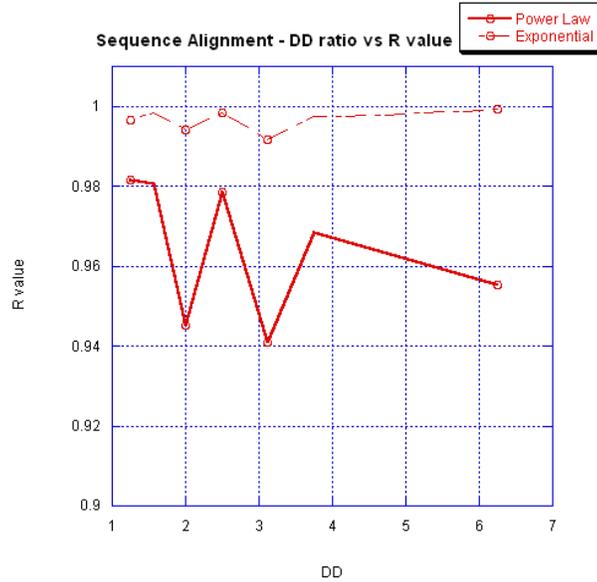


FIGURE 4.5. The relationship between DD ratio and R value for Sequence Alignment Approach



In Figure 4.5, changing the DD ratio does not affect the exponential distribution fit much and the exponential distribution fits the data much better than the power law distribution. However, unlike DM, changing DD in the range we provided does not seem to affect the power law fit also. The R value for power law is going down as DD increases but it stays above 0.9, which indicates that for any death rates in this range, power law can be a good fit for the data.

#### 4.1.3. Mutation Count Approach.

4.1.3.1. *DM Ratio and R value.* To study the relationship between DM ratio and R value, we set the DD ratio constant at 2.0 and the cutoff value at 0.07. We give various values to duplication up to  $1/300000$ . A duplication rate smaller than  $1/300000$  produces very few data points, which make regression unreliable.

From Figure 4.6, we can see that the lower the DM ratio the higher the R value for both the power law and exponential distributions. Exponential distribution is not affected by the DM value as its R value always stays higher than 0.9. Power law is a good fit to the data for DM ratios smaller than 12.5. Although lower ratios show a better R value, ratios below 6.67 also lead to fewer data points (with a ratio of 4, there are only 3 data points). For data that

FIGURE 4.6. The relationship between DM ratio and R value for Mutation Count Approach

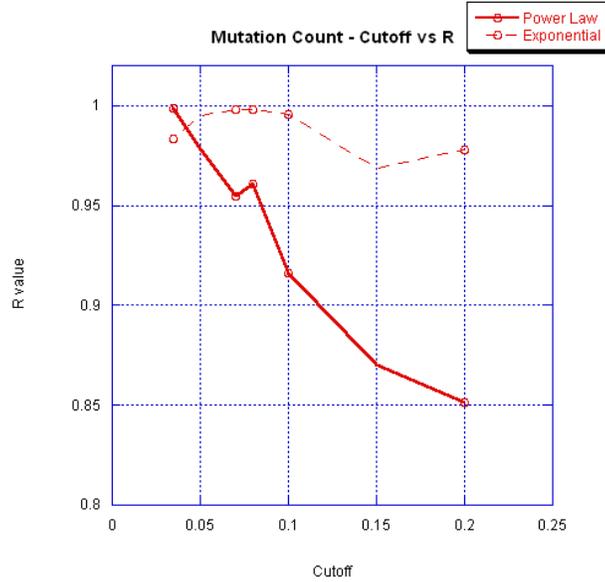


can fit well to exponential and power law distribution, we should use a ratio in the range from 12.5 to 6.67 or duplication rate from  $1/160000$  to  $1/300000$ . This range for duplication confirms our calculation in Equation 4.20 based on Gao and Inaan's study [15]. As exponential is not affected by DM ratio in this range, in later section where we need to fix the duplication rate, we will use the value  $1/160000$  for a better fit of power law distribution.

4.1.3.2. *Cutoff Value and R value.* The cutoff value in Mutation Count approach is not measured as the difference between the sequences but the number of mutations separating the two genes (calculated as a fraction of the gene length). We set the gene death rate at  $1/160000$ , or half the duplication rate.

As we can see in the graphs in Figure B.3 and B.4, the distribution fits power law well for small cutoff values below 0.1. A cutoff below 0.035 would give too few data points and therefore a biased regression. We can use any values from 0.1 to 0.035 for a good power law fit. Although the R values for power law distribution is high in this area, exponential distribution mostly fits the data better than power law. In Figure 4.7, we can see that the Exponential curve is mostly

FIGURE 4.7. The relationship between cutoff and R value for Mutation Count Approach



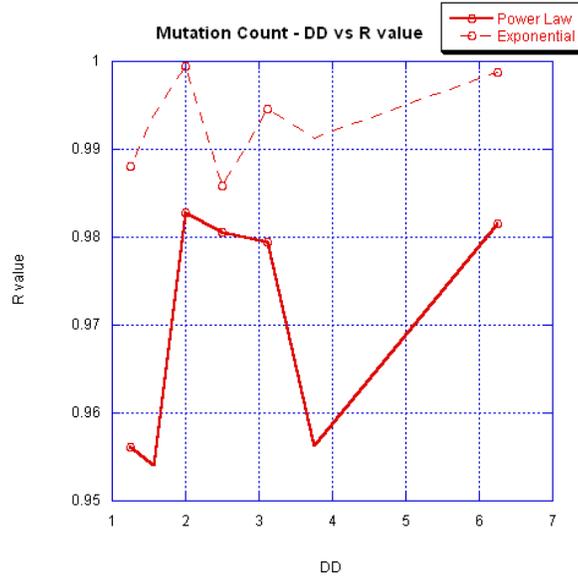
above the power law curve and it is also always above 0.9. Therefore, although power law can be a good fit for the data, the data can also be fitted by exponential distribution.

4.1.3.3. *Death Rate and R value.* To see the effect of death rate on the R value of the fit, we fix the cutoff at 0.07 and try several death rates.

Figure 4.8 shows that both power law and exponential distributions fit the data pretty well. Exponential is significantly better than power law for all values that we considered. As we have seen in previous sections, exponential's R value does not change much when we use different values for the parameters. An interesting point is that DR ratio does not seem to have a big impact on the power law's R value. Power law fit's R value never falls below 0.9 for all death rate values that we considered. In conclusion, in Mutation Count approach, exponential distribution fits the data well for a wide range of parameters while power law distribution depends on the cutoff value and the DM ratio.

#### 4.1.4. Preferential Attachment Model.

FIGURE 4.8. The relationship between DD ratio and R value for Mutation Count Approach



4.1.4.1. *Death Mutation Ratio and R value.* We will study the effect of DM ratio on the R value in Preferential Attachment to see which ratio can bring the best fit for the two distributions. As we have seen before, we will keep the DD ratio constant at 2 and the cutoff at 0.07.

From Figure 4.9, for all values in the range from 6.67 to 20, power law is a good fit but exponential is generally better (there is one instance that the power law's R value is greater than the exponential's). The minimum value that we give DM was 6.67 because a higher value for duplication gives us very few data points. The lower the DM, the better the power law fit but also the fewer the number of data points. An interesting point is that this is approximately the same range of DM ratio that we found in the other two models. Considering the DM ratio mentioned in Gao and Inaan's study [15], this range of value confirms the ratio of 28 between duplication and neutral mutations (which is converted to  $DM = 14$ ).

4.1.4.2. *Cutoff and R value.* We set the duplication rate at  $1/160000$  and death rate at  $1/320000$ . Cutoff is given various values to test its relationship with the R value for each type of distribution.

FIGURE 4.9. The relationship between DM ratio and R value for Preferential Attachment Model

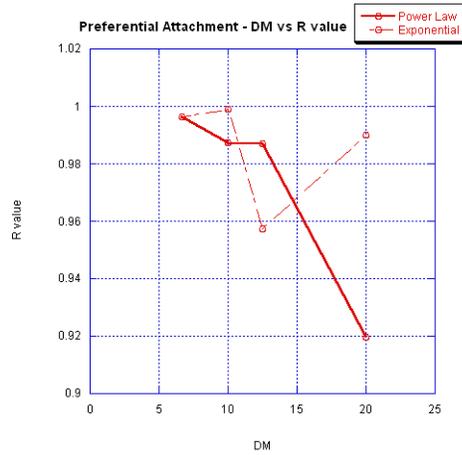
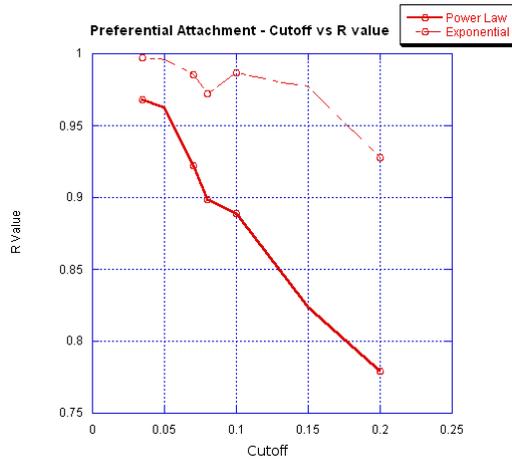


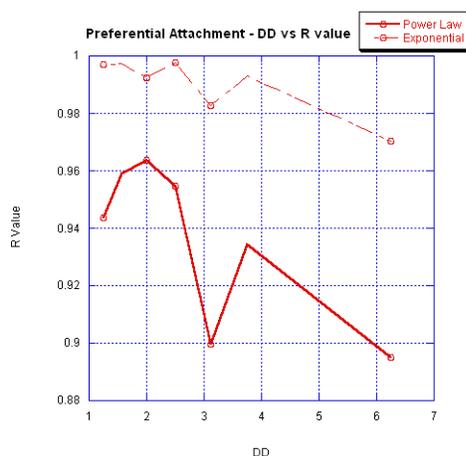
FIGURE 4.10. The relationship between cutoff and R value for Preferential Attachment Model



As Figure 4.10 suggests, exponential is always a good fit to the data and it is significantly better than power law distribution. Power law distribution fits the data well for cutoff values below 0.25. This is the same range of values that we found for the previous models.

4.1.4.3. *DD ratio and R value.* To study the relationship between DD ratio and R value, we fix duplication rate at  $1/80000$  and cutoff at 20.

FIGURE 4.11. The relationship between DD ratio and R value for Preferential Attachment Model



According to Figure 4.11, for a DD ratio less than 6, we the power law distribution is a good fit to the data although it is always worse than exponential for all DD ratios. In general, a higher DD ratio means a lower R value for both distributions.

## 4.2. Discussion

For DM ratio, all three models suggest that the higher the DM ratio the lower the R values for the fits for both distributions. As we fix the mutation rate, a higher DM means a higher duplication rate. If duplication happens more often, it would take less time for the family to reach the limit size so less mutations occur. Therefore, there are more proteins related to each other. In the distribution, we would have more higher degree nodes and less lower degree nodes. The typical shape of power law distribution disappears and the R value decreases. Similarly, we observe the same pattern for cutoff value. A higher cutoff means that there is a higher probability that two proteins are related. There would be less low degree and more high degree nodes which is the opposite of the power law shape. DD ratio does not seem to play an important role. For all DD values that we consider, the power law does fit the data reasonably well.

In the Mutation Count approach, we substitute the JC augmented score by the real number of mutations. If the JC model is correct, the Sequence Alignment approach should give approximately the same result as the Mutation Count one. Although we do not observe exactly the

same graph for both models, the trend of changes for the R value with respect to the changes of the variables are similar. In both approaches, all the variables do not affect the exponential much, we will only discuss the similarity in the power law fit. For DM ratio, the area that gives a good fit for power law behavior is between 12.5 and 20 for both model. Sequence Alignment reports a range of cutoff from 0.03 to 0.08 in which power law fits the data the best. Likewise, Mutation Count gives the same area of cutoff. DD ratio does not seem to affect the power law fit in the two approaches much as R values stay above 0.9 for all values that we consider. This similar behavior suggests that JC model does work very well in approximating the number of mutations in our model. Also, as the two approaches would give the same result and the Mutation Count approach has a much better running time (see Table 3.4), we can use the Mutation Count instead of the Sequence Alignment in simulating evolution. For this reason, we could implement Preferential Attachment model using Mutation Count approach and expect to get the same result as if we had implemented with Sequence Alignment.

An interesting fact that we have found in all of our analysis is that exponential is always a better fit than power law. In many cases, we have a limited number of points so the fitted exponential and power law curves are close to each other. However, in many regions where we have many data points (especially where power law does not fit so well), exponential is a much better choice. An explanation for exponential behavior would be using the limit case suggested in Barabási et al's paper. According to this paper, exponential would occur if a network only exhibits growth not Preferential Attachment. If this paper is correct, in models that have incorporated Preferential Attachment, there should be a higher R value for power law than exponential. However, even when we explicitly account for Preferential Attachment in the third model, exponential distribution still dominates, suggesting that the family network must have some other structure significances other than scale free leading to this phenomenon. Furthermore, as the Mutation Count and Preferential Attachment do not differ significantly in their results, we can conclude that our Mutation Count approach (and equivalently our Sequence Alignment approach) does include Preferential Attachment. The better R value for exponential distribution also supports Stumpf et al.'s idea that power law may not be the best fit for protein network [28].

## Model Verification

### 5.1. Data Retrieval and Degree Distribution Computation

To test our model, we need data on real life protein sequences of a family. The protein family database that we use is the NCBI's (National Center for Biotechnology Information) **Protein Clusters** database [20]. For each protein family, NCBI includes a global Sequence Alignment of all the proteins in the family. However, the alignment was based on amino acid sequence, not nucleotide one as in our model so we cannot use their alignment score for our research. Instead, we use the Protein Clusters to retrieve the nucleotide sequences for all proteins in the family, and conduct pairwise alignment as we did in the Sequence Alignment approach. However, getting the nucleotide information is not a trivial task. Each protein included in the family in Protein Cluster is an ID name in the **Entrez Nucleotide** database (a protein database also maintained by NCBI). We can use this ID to view the protein entry in Entrez which only contains the protein amino acid sequence. We cannot map the amino acid back to the nucleotide sequence because it is not a one to one mapping from a codon to a amino acid. Fortunately, each protein entry in Entrez also includes a **CDS** (Coding Sequence) section. CDS is a link to another web-page which includes the nucleotide sequence that encodes the proteins. Now, we can take this sequence to construct the family graph.

We created a program to automatically retrieve all proteins' nucleotide sequences in a an input family. The program follows the method that we described above. First, the program needs to load the web page that contains the list of proteins in the input family (we will refer to this page as **ProteinList** page). We can construct the URL for ProteinList page based on an observation that its url follows the form:

<http://www.ncbi.nlm.nih.gov/sutils/prkview.cgi?result=align&cluster=X>

where X is the Protein Clusters ID of the family.

ProteinList page includes a list of links to protein Entrez entry web page (referred to as **EntrezEntry** page). Each link follows the following regular expression:

$$\text{http://www.ncbi.nlm.nih.gov/entrez/viewer.fcgi?db=protein\&val=(.*)\$}$$

We can search in the html source code of the ProteinList page for all lines that conforms to the above expression. Each result line is a link to a EntrezEntry page which has a link to a **CDS** page containing the nucleotide sequence. We can store all lines in a list and start extracting the nucleotide sequence for each protein. Applying the same method as we did to get the EntrezEntry page, we will be searching the html source code of the EntrezEntry for a link to CDS page with the following regular expression:

$$(.*)\text{CDS}(.*)\$$$

After we get the CDS page, we now have the nucleotide sequence at the end of the page. The nucleotide sequence section starts with a number 1 signifying the start position (following the expression “<>(.)1(.\*)\$”). The end of the section is not certain as the page does not have a consistent ending signal. As the nucleotide sequence is the last part of the page, we can use the beginning of the page’s ending line as the end of nucleotide sequence. The ending line follows the expression:

$$\text{< http://www.ncbi.nlm.nih.gov/About/disclaimer.html > Disclaimer}$$

However, there may still be text between the end of the sequence and the last line of the page. The text is either empty or non letter. Also, the protein sequence is broken into lines each of which contains 60 bases. In front of each line, there is a number indicating the starting index of the line. We need to get rid of both the ending text and the indexes by removing all characters that do not belong to the set {A,C,G,T} or do not follow the expression [actg]. After the removal, the text from the start position to the end position is the nucleotide sequence of the protein we are considering. We repeat the process for every protein in the list that we have from the ProteinList page. In the end, we will have a list of sequences for all proteins in the

family. We can now apply pairwise alignment on every pair of proteins in the list and get the degree distribution.

From the algorithm description, it is clear that we need a way to load a web-page's html source code as a string so that we can apply regular expression search. For this purpose, we use **HTML Parser** [22], an open source Java library supporting parsing a web page.

## 5.2. Results

To use the algorithm that we described above, we need to supply a protein family ID. We need a family which is similar to the result family of the simulation. The simulated family has 500 proteins so we want a family around that size and we chose to use a 400 protein family. The selected family's ID is **PRK00013, chaperonin GroEL**. This family contains proteins of length around 1500 bases which comparable to the proteins that we have in our simulation.

Unlike the simulation, to verify our model with real data, we only need one tunable parameter, the **cutoff** value for determining relationship. We try the following cutoff values: 0.07, 0.08, 0.09, 0.1, 0.2 and 0.25. The graph of the result data for these cutoff values can be found in Figure 5.1 and 5.2.

The graphs in Figure 5.1 and 5.2 show a quite different trend as we saw in the simulation data. The power law fits the data for cutoffs from 0.07 to 0.1. When the cutoff increases to 0.2 or above, the graph does not display power law behavior anymore. We also used the value 0.07 for all the cutoffs in our models. The behavior of the power law distribution is similar to our models. The greater the cutoff, the smaller the R value. However, the exponential distribution is different. We have seen in all models that exponential distribution consistently fits the data much better than power law but in the real data, exponential is worse than power law for cutoff below 0.2. In conclusion, data from a real protein family suggests that the degree distribution of a protein family does follow power law distribution.

FIGURE 5.1. The graphs and fitted curves for real protein data

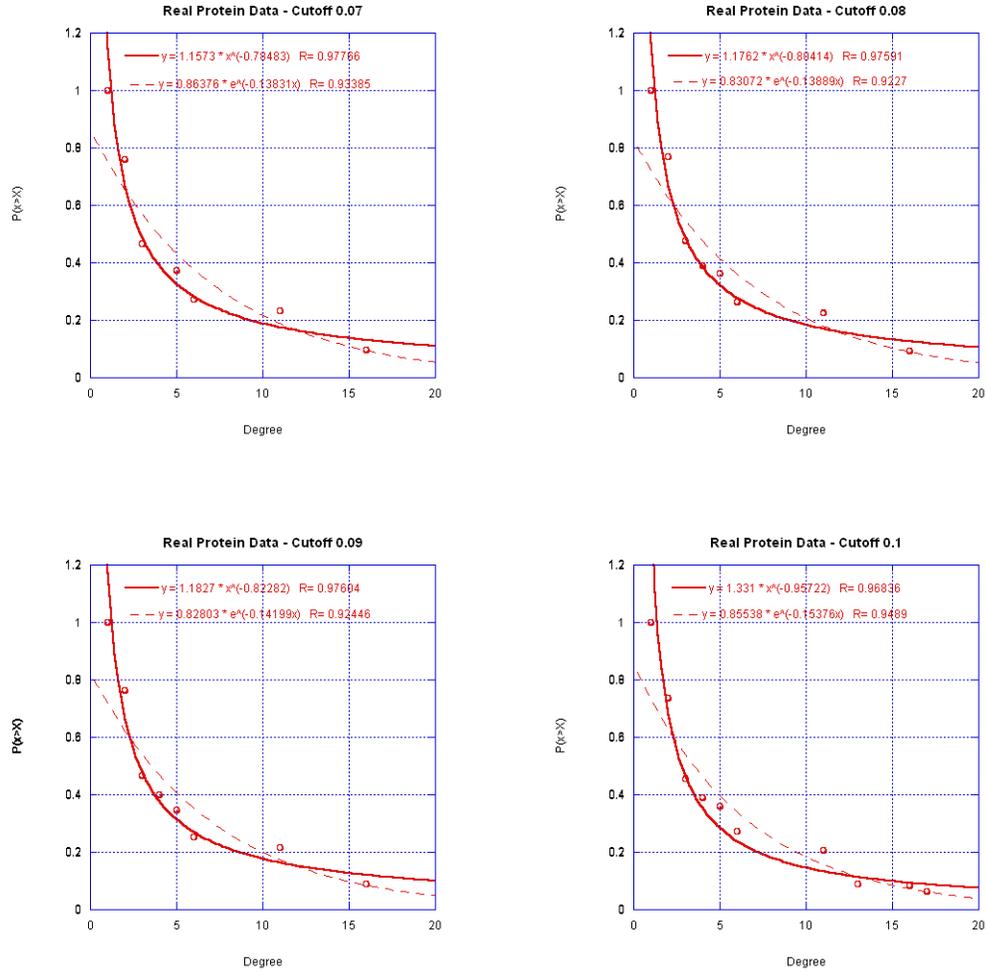


FIGURE 5.2. The graphs and fitted curves for real protein data (cont)

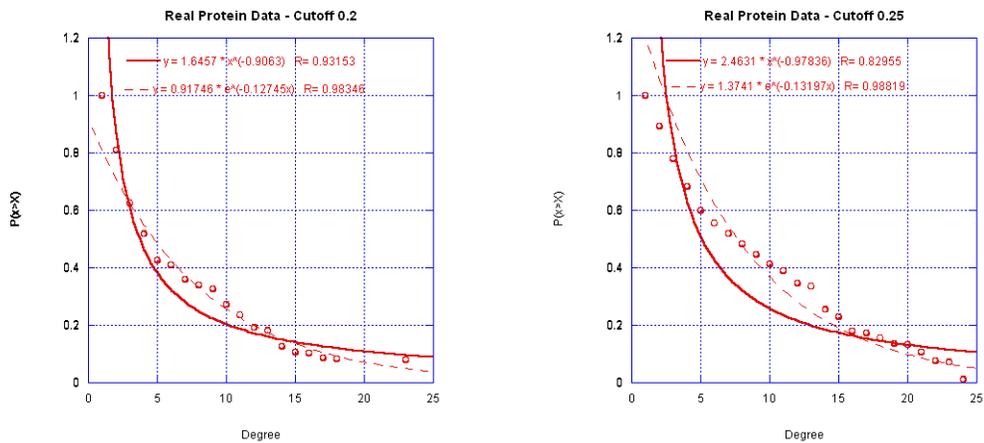
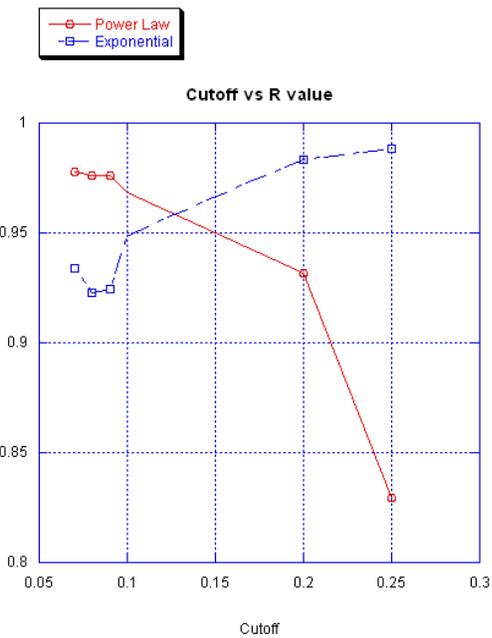


FIGURE 5.3. The relationship between Cutoff and R value for Real Data



## Conclusion

### 6.1. Implications of Results

We saw in the analysis in Chapter Four that the range of values in which power law fits the data well suggested by all models is:

- (1) Duplication Rate  $\sim 1/160000$
- (2) Mutation Rate  $\sim 1/2000000$
- (3) Death Rate  $\sim 1/320000$
- (4) Cutoff  $\sim 0.07$

The value of mutation rate is taken from suggestions in Snustad's book [26], the duplication rate fits the rate between neutral mutations and mutation rate suggested in Li et al.'s study [19]. For the cutoff value, in Chapter Five, we found out that the range of value in which power law occurs is below 10% of the string length or below 100 of a string of size 1000. It can be seen that the range of values we found fits well with other findings or real life data. The claim that the family structure follows power law distribution is plausible for this range. Therefore, the protein family is very likely to be a scale free network. This network property seems to be very useful for the network as it guards the network against random mutations. The chance of a mutation is quite high and if the network is not robust against random changes, it would fall apart easily. A collapse of a protein family would lead to a serious consequence for the species function, which should not be allowed. On the other hand, if some mutations manage to alter the high degree proteins, the family structure is significantly affected, which would lead to a disruption in the protein function. However, with a structure with much more low degree nodes than high degree nodes, such a devastated mutation is not very likely.

Nevertheless, our models do not completely reflect real world evolution. Consider the graph in Figure 5.3. We see a pattern that the power law would dominate for small values of cutoff and the exponential takes over for larger cutoff values. None of the three models we proposed

generated a similar behavior. In all of these models, the exponential is better than the power law for all cutoff values, which suggests that we must have missed something in our models. We will discuss some possible ways to improve our model in the next section.

## 6.2. Extending the Model

Our simple model left out some other possibly important operations in evolution. The first is **horizontal transfer**, a process by which a new gene is imported into a species from another species. This particular operation can explain many phenomena in biology, such as rapid bursts in evolution [29]. We can simulate this events by starting the evolution with multiple original proteins. From each of these starting proteins, we will have a separate family in the end of evolution. At any time step, one gene from one family can be inserted to a different family. Another significant operation is **gene conversion**, in which DNA information from one gene is transferred to another gene [14]. This process renders the receiver gene to be more similar to the source gene so it can fix the differences resulting from many rounds of mutations. Gene conversion can be simulated as a series of point mutations at consecutive locations. Adding these operations would create a more realistic model but it also increases the complexity. We would have more evolution rates to consider. These operations may be the reason for the difference between our graphs in Chapter Five and Four. Also, all mutation types that we consider only affect the protein sequence at one nucleotide site. We may need to expand the model to consider other types of mutations that can affect many positions in one operation.

A comparison between the Preferential Attachment and the Mutation Count approach reveals that these two models are not much different in their result. This may be because of our method of implementing Preferential Attachment. We just chose a uniformly distributed degree cutoff in the range from the minimum to the maximum. We did not find any previous work suggesting how evolution operations were executed biased to low degree nodes in nature. It could be that the cutoff degree should be picked as a Poisson generated variable as we did when simulating the various operations. The fact that Preferential Attachment plays an important role in many real life networks suggests that it may be very important to protein family network as well. The problem is we do not know how to simulate it correctly.

The simulation may be used to devise a method to determine protein family. Given a set of protein sequences, we can construct its degree distribution as we did in the Chapter Five and check what distribution fits the data. If the data does not fit power law distribution, we can definitely reject that the protein group is a family. If it does follow the power law, we cannot conclude anything yet but have to use more conventional methods to determine the relationship. Our simulation serves as a preliminary check to eliminate non-family sets of proteins.

## Bibliography

- [1] Réka Albert and Albert-László Barabási. Statistical mechanics of complex networks. *Rev. Mod. Phys.*, 74(1):47–97, Jan 2002.
- [2] Siv G. E. Andersson and Charles G. Kurland. Reductive evolution of resident genomes. *Trends in Microbiology*, 6(7):263–268, 1998.
- [3] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.
- [4] Albert-Laszlo Barabasi and Eric Bonabeau. Scale-free networks. *Publicationes Mathematicae*, 288:60–69, 2003.
- [5] G. Bebek, P. Berenbrink, C. Cooper, T. Friedetzky, J. Nadeau, and S. C. Sahinalp. The degree distribution of the generalized duplication model. *Theor. Comput. Sci.*, 369(1):239–249, 2006.
- [6] Dennis A. Benson, Ilene Karsch-Mizrachi, David J. Lipman, James Ostell, and David L. Wheeler. GenBank. *Nucleic Acids Research*, 35:D21–25, 2007.
- [7] Aaron Clauset, Cosma R. Shalizi, and M. E. J. Newman. Power-law distributions in empirical data. <http://arxiv.org/abs/0706.1062>, Jun 2007.
- [8] Sergio Anibal de Carvalho Junior. Neobio - bioinformatics algorithms in java. <http://neobio.sourceforge.net/>.
- [9] NIST/SEMATECH e-Handbook of Statistical Methods. 6.3.3.1. counts control charts. <http://www.itl.nist.gov/div898/handbook/pmc/section3/pmc331.htm>, 2006.
- [10] Erdős and Rényi. On random graphs. *Publicationes Mathematicae*, 6:290–297, 1959.
- [11] R.D. Finn, J. Mistry, B. Schuster-Bockler, S. Griffiths-Jones, V. Hollich, T. Lassmann, S. Moxon, M. Marshall, A. Khanna, R. Durbin, S.R. Eddy, E.L. Sonnhammer, and A. Bateman. Pfam: clans, web tools and services. *Nucleic Acids Researchs*, 34:D247–251, 2006.
- [12] Walter M. Fitch. Homology: a personal view on some of the problems. *Trends in Genetics*, 16(5):227–231, 2000.
- [13] M.T. Flanagan. Michael Thomas Flanagan’s java scientific library. <http://www.ee.ucl.ac.uk/mflanaga/java/>, 2007.
- [14] Davd Freifelder. *Molecular biology*. Jones and Bartlett Publishers, Inc, 1987.
- [15] Li-zhi Gao and Hideki Innan. Very low gene duplication rate in the yeast genome. *Science*, 306(5700):1367–1370, 2004.

- [16] Xionglei He and Jianzhi Zhang. Higher duplicability of less important genes in yeast genomes. *Mol Biol Evol*, 23(1):144–151, 2006.
- [17] Georgy Karev, Yuri Wolf, Andrey Rzhetsky, Faina Berezovskaya, and Eugene Koonin. Birth and death of protein domains: A simple model of evolution explains power law behavior. *BMC Evolutionary Biology*, 2(1):18, 2002.
- [18] Eugene V. Koonin, G. Karev, and Y. Wolf. *Power laws, scale-free networks and genome biology (molecular biology intelligence unit)*. Springer, March 2006.
- [19] Li Li, Yingwu Huang, Xuefeng Xia, and Zhirong Sun. Preferential duplication in the sparse part of yeast protein interaction network. *Mol Biol Evol*, 23(12):2467–2473, 2006.
- [20] NCBI. Protein clusters home. <http://www.ncbi.nlm.nih.gov/sites/entrez?db=proteinclusters>.
- [21] Susumu Ohno. *Evolution by gene duplication*. Springer-Verlag, 1970.
- [22] HTML Parser. Html parser. <http://htmlparser.sourceforge.net/>.
- [23] Ryszard Rudnicki, Jerzy Tiuryn, and Damian Wójtcowicz. A model for the evolution of paralog families in genomes. *Journal of Mathematical Biology*, 53(5):759–770, 2006.
- [24] M. Salemi and Vandamme A. M. (ed). *The phylogenetic handbook. A practical approach to DNA and protein phylogeny*. Cambridge University Press, UK, 2003.
- [25] Temple F. Smith, Michael S. Waterman, and Christian Burks. The statistical distribution of nucleic acid similarities. *Nucl. Acids Res.*, 13(2):645–656, 1985.
- [26] D. Snustad and M. Simmons. *Principles of genetics*. Wiley & Son, 2003.
- [27] Synergy Software. Kaleidagraph - scientific graphing, curve fitting, data analysis software. <http://www.synergy.com/>.
- [28] Michael P. H. Stumpf and Piers J. Ingram. Probability models for degree distributions of protein interaction networks. *Europhysics Letters*, 71:152, 2005.
- [29] Michael Syvanen. Cross-species gene transfer; implications for a new theory of evolution. *Journal of Theoretical Biology*, 112:333–343, 1985.
- [30] A. Vázquez, A. Flammini, A. Maritan, and A. Vespignani. Modeling of protein interaction networks. *Complexus*, 1(1):38–44, 2003.
- [31] Damian Wójtcowicz and Jerzy Tiuryn. On genome evolution with innovation. *Mathematical Foundations of Computer Science 2006*, pages 801–811, 2006.
- [32] Itai Yanai, Carlos J. Camacho, and Charles DeLisi. Predictions of gene family distributions in microbial genomes: evolution by gene duplication and modification. *Phys. Rev. Lett.*, 85(12):2641–2644, Sep 2000.

# APPENDIX

This appendix contains examples of graphs resulting from simulation with a variety of parameter settings. In each graph, the bold line is power law distribution fitting while the dotted line is the exponential distribution fitting.

## A. Sequence Alignment

FIGURE A.1. The graphs and fitted curves for Sequence Alignment Approach with variable cutoff

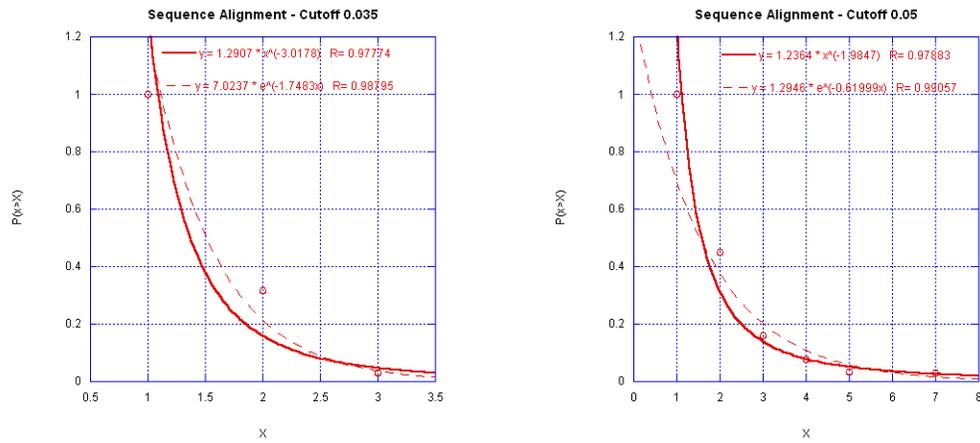


FIGURE A.2. The graphs and fitted curves for Sequence Alignment Approach with variable cutoff (cont)

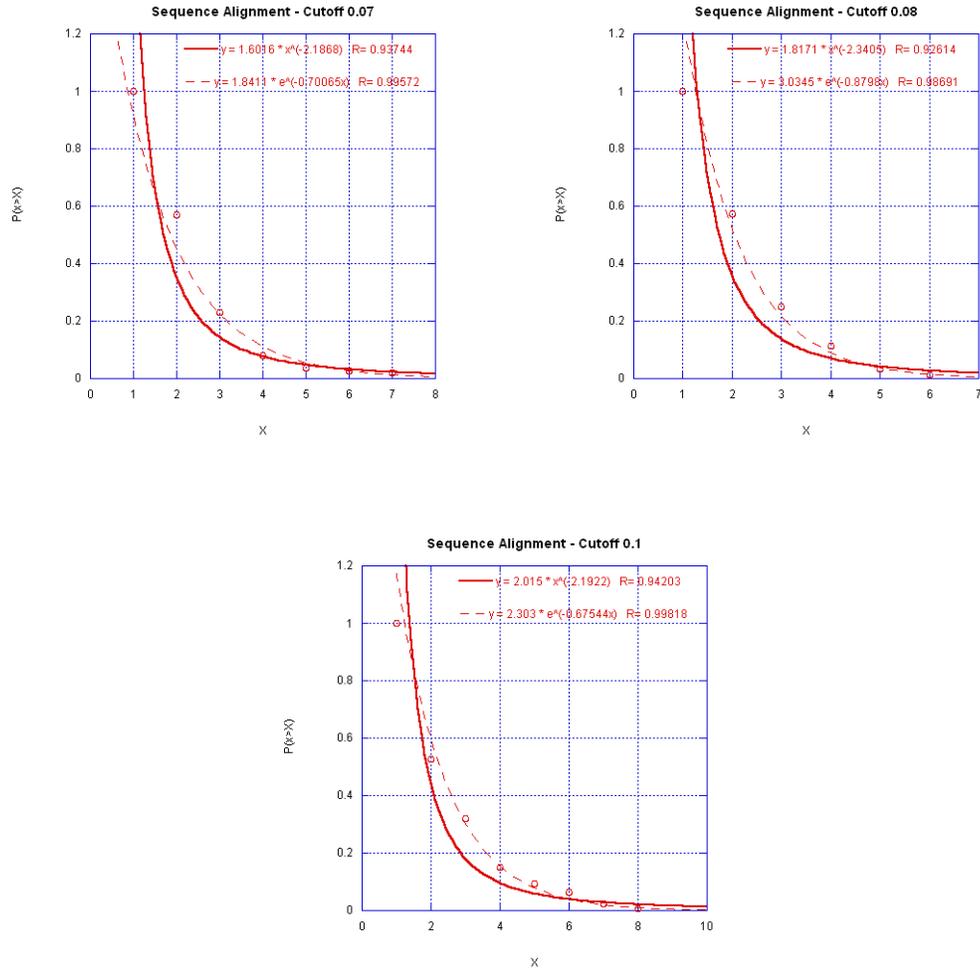


FIGURE A.3. The graphs and fitted curves for Sequence Alignment Approach with variable cutoff (cont)

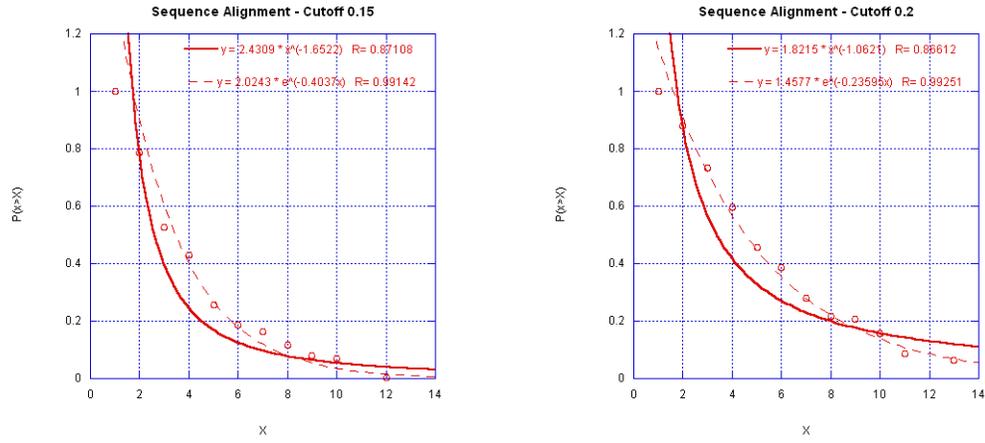


FIGURE A.4. The graphs and fitted curves for Sequence Alignment Approach with variable DD ratio

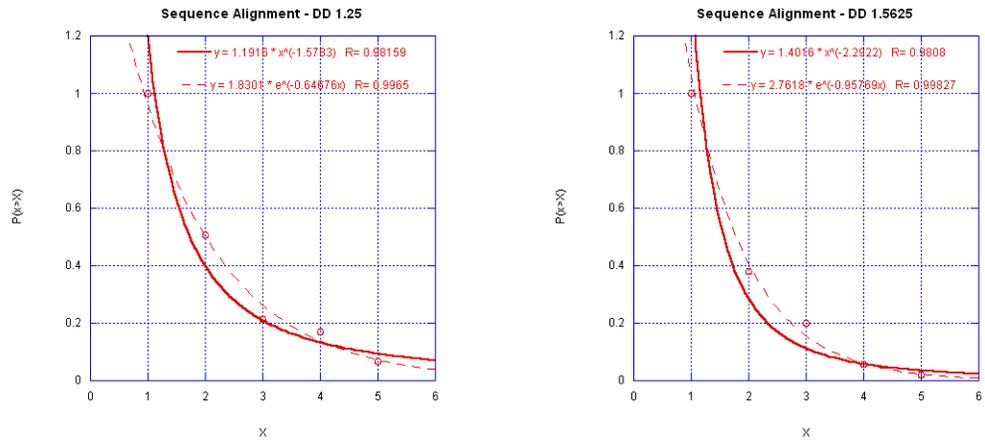


FIGURE A.5. The graphs and fitted curves for Sequence Alignment Approach with variable DD ratio (cont)

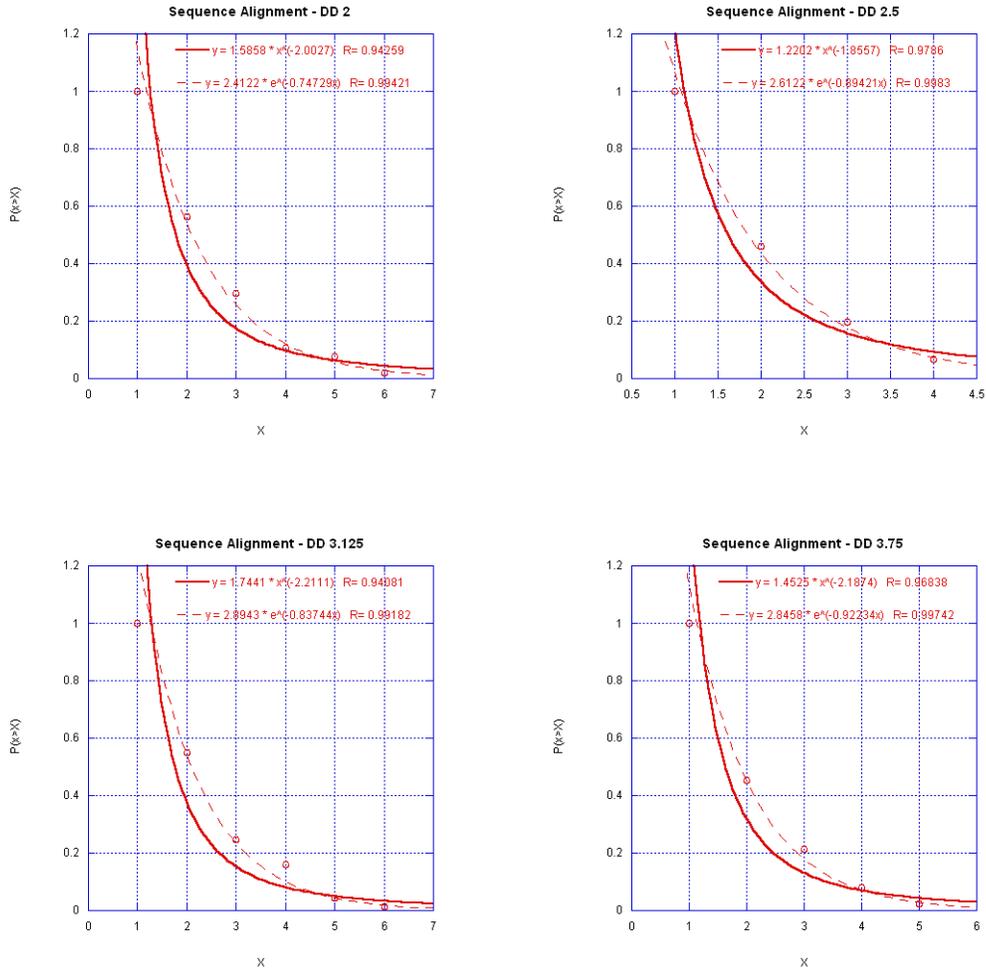
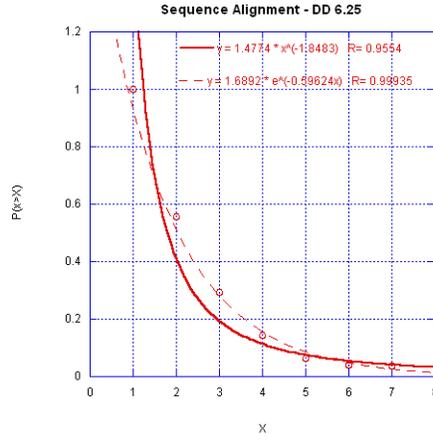


FIGURE A.6. The graphs and fitted curves for Sequence Alignment Approach with variable DD ratio (cont)



**B.Mutation Count**

FIGURE B.1. The graphs and fitted curves for Mutation Count Approach with variable DM ratio

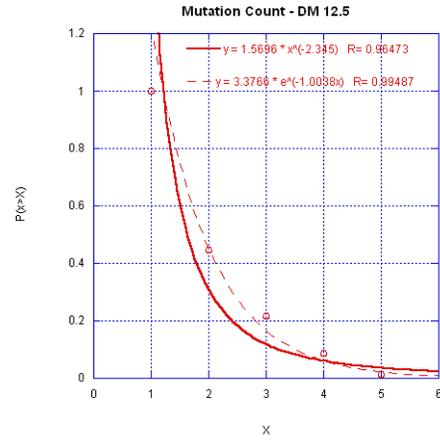
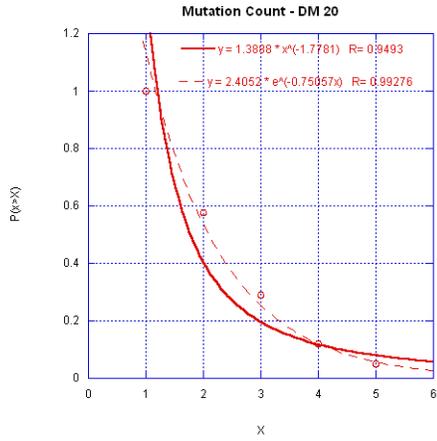


FIGURE B.2. The graphs and fitted curves for Mutation Count Approach with variable DM ratio (cont)

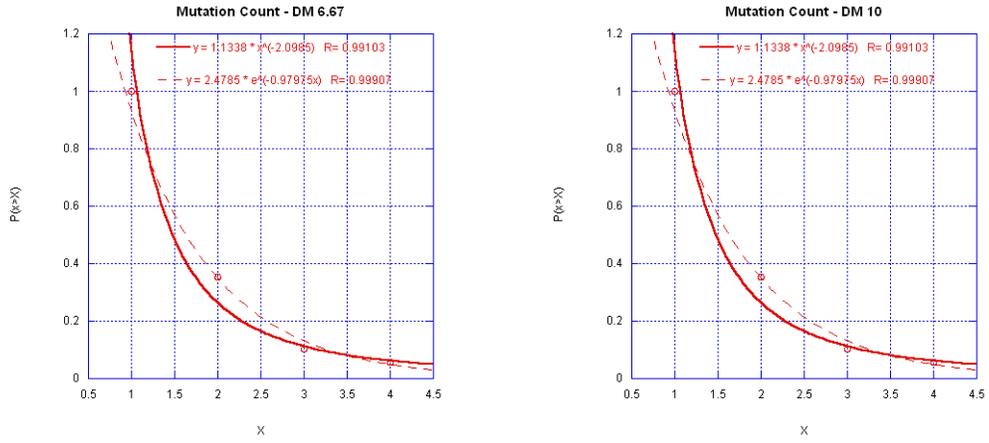


FIGURE B.3. The graphs and fitted curves for Mutation Count Approach with variable cutoff

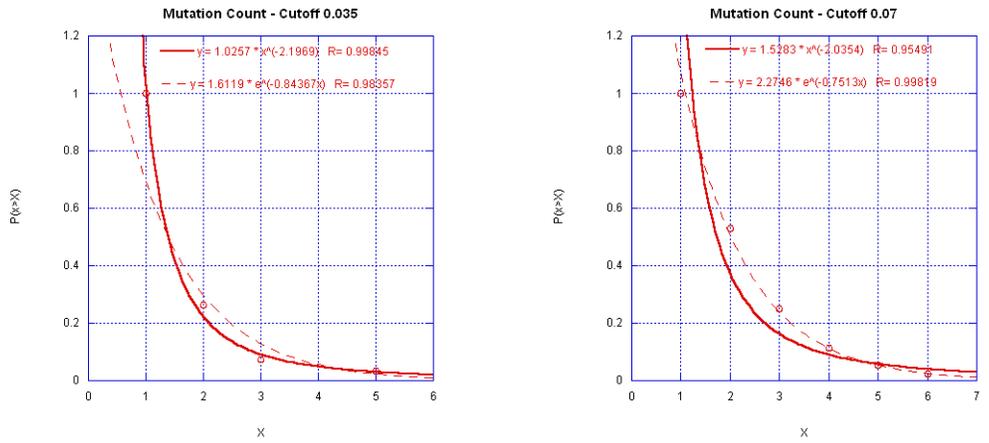


FIGURE B.4. The graphs and fitted curves for Mutation Count Approach with variable cutoff (cont)

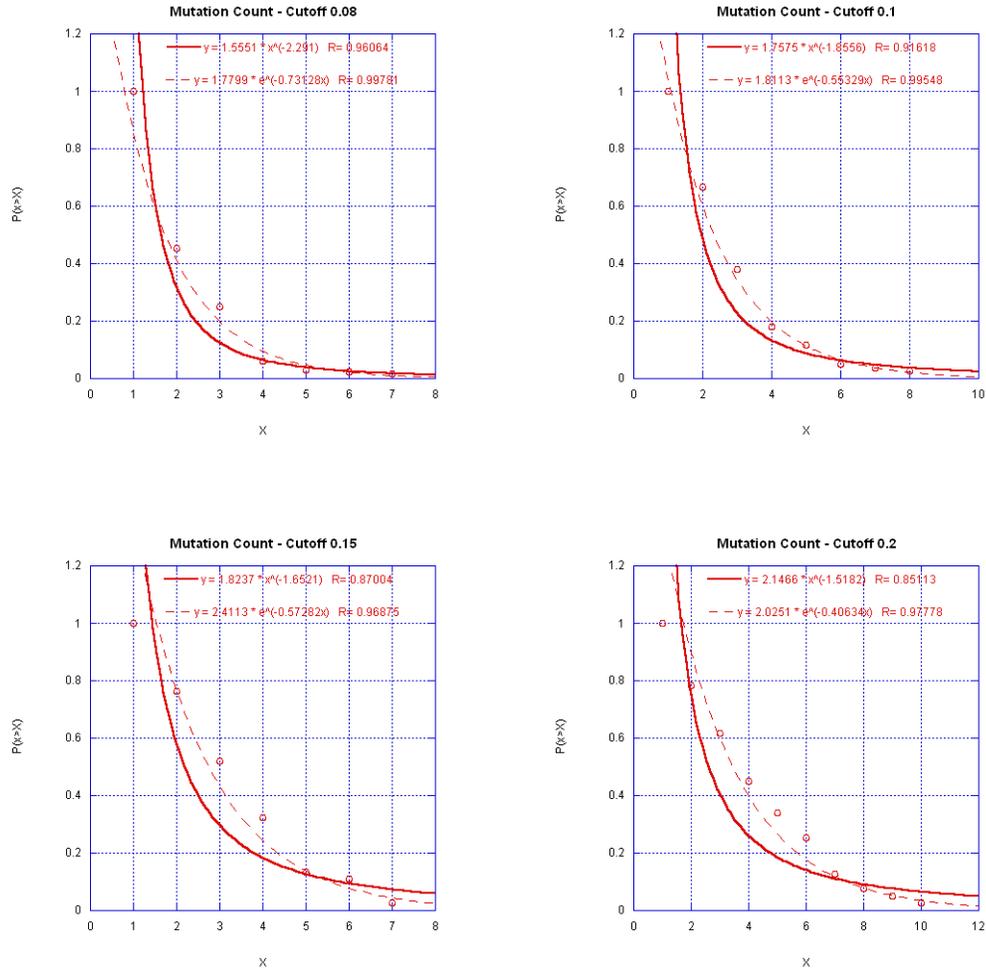


FIGURE B.5. The graphs and fitted curves for Mutation Count Approach with variable DD ratio

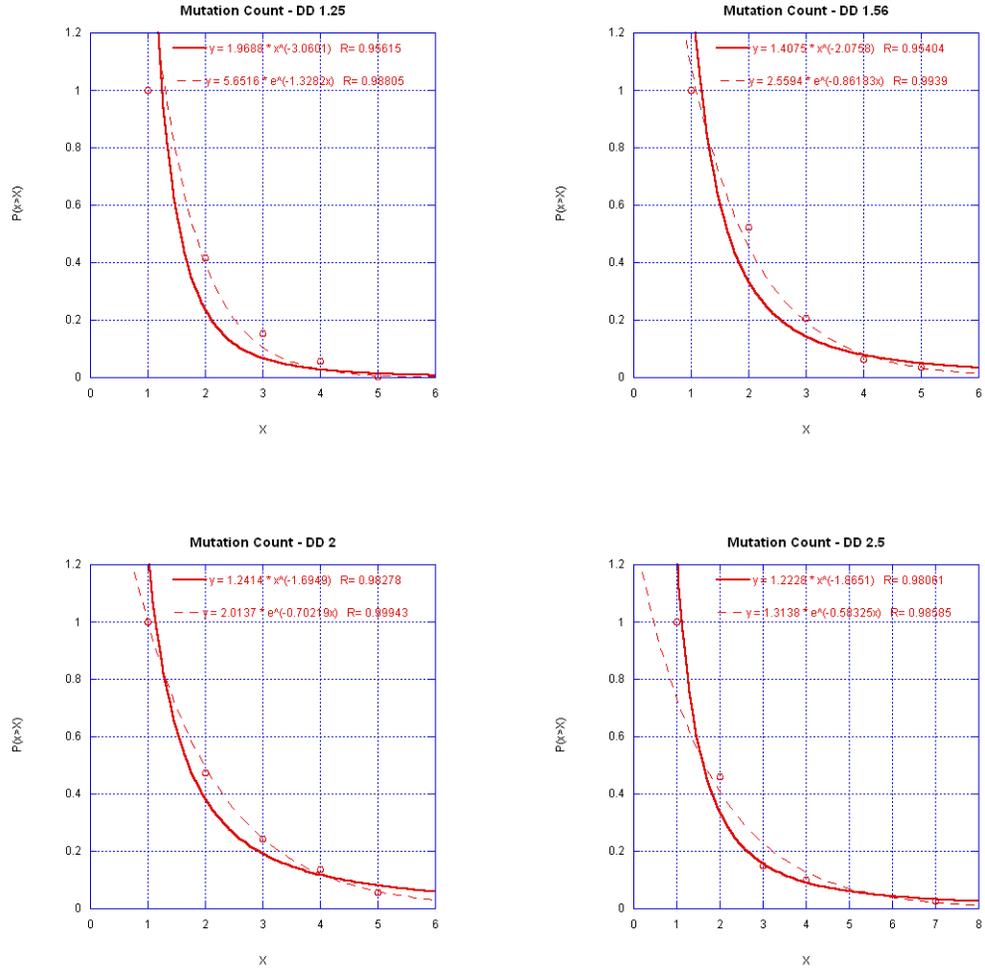
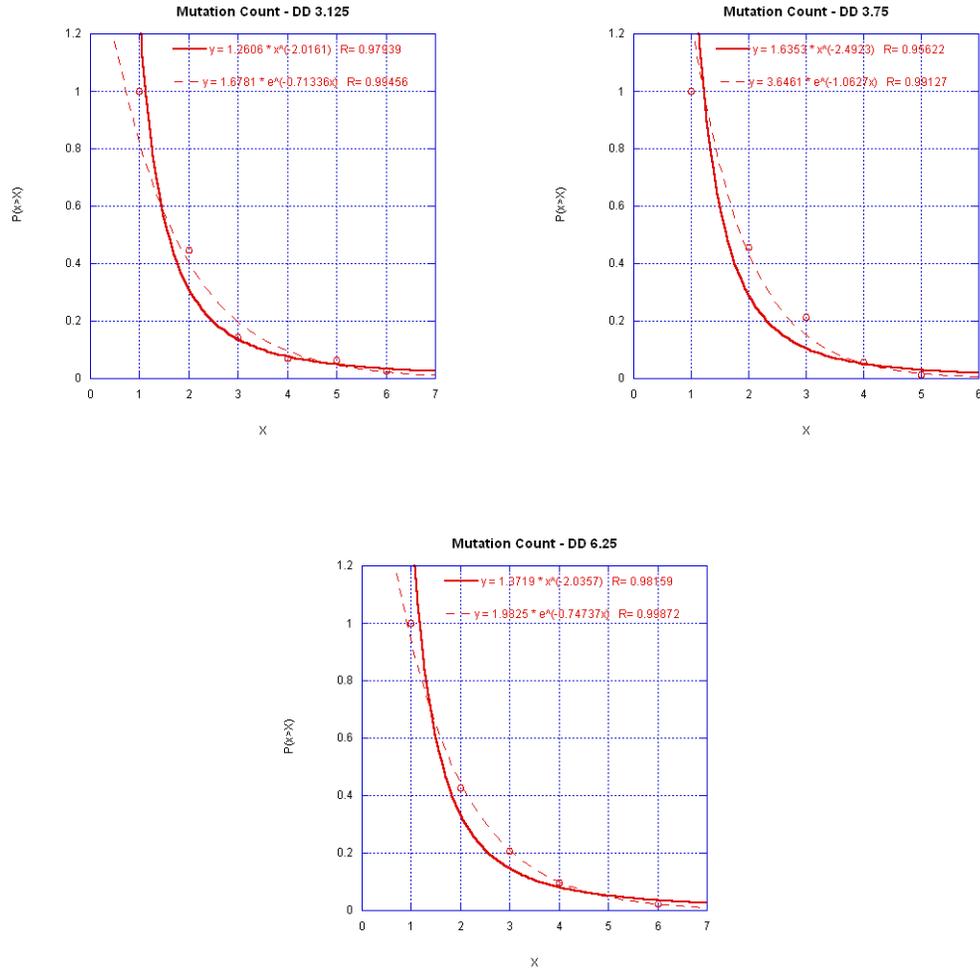


FIGURE B.6. The graphs and fitted curves for Mutation Count Approach with variable DD ratio (cont)



C. Preferential Attachment

FIGURE C.1. The graphs and fitted curves for Preferential Attachment Model with variable DM ratio

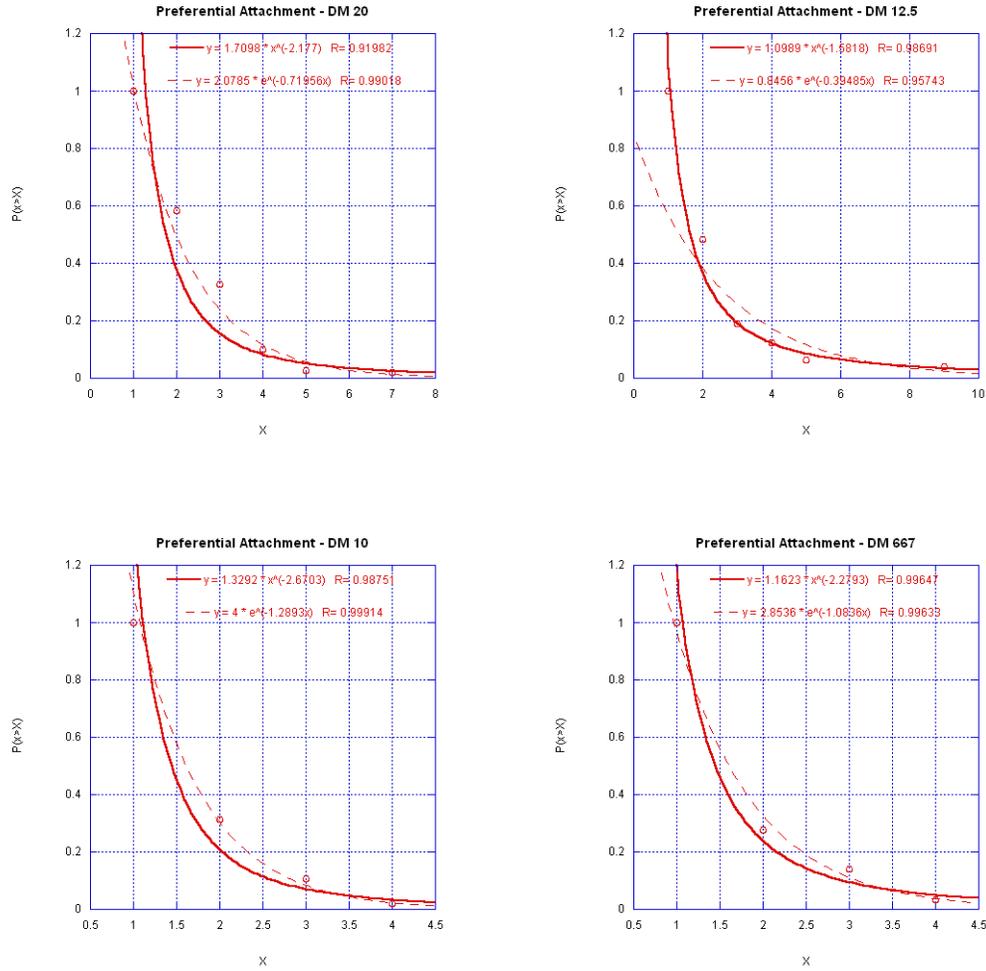


FIGURE C.2. The graphs and fitted curves for Preferential Attachment Model with variable cutoff

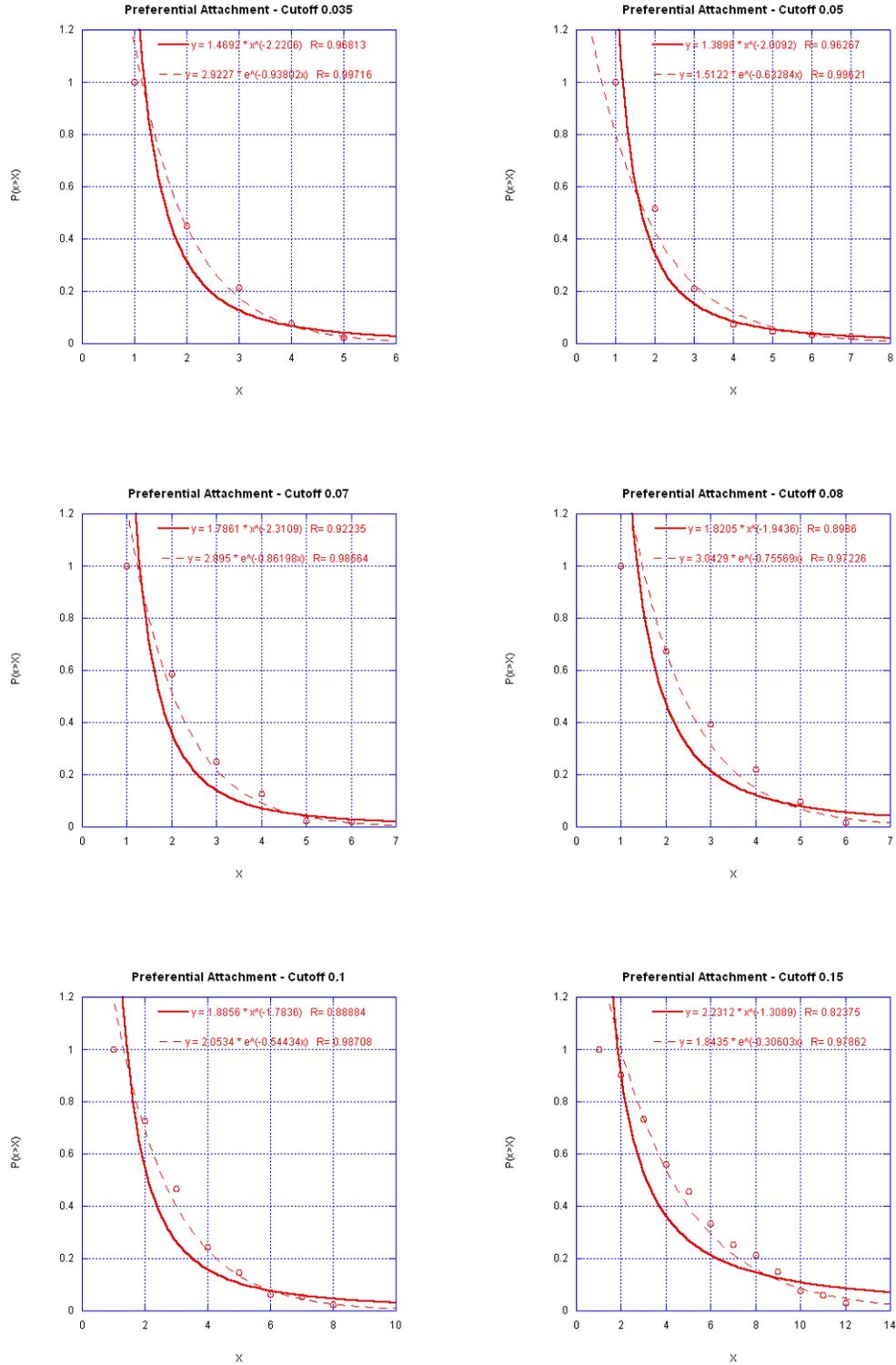


FIGURE C.3. The graphs and fitted curves for Preferential Attachment Model with variable cutoff (cont)

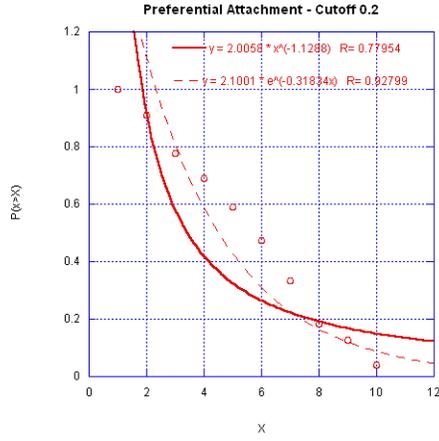


FIGURE C.4. The graphs and fitted curves for Preferential Attachment Model with variable Death Rate

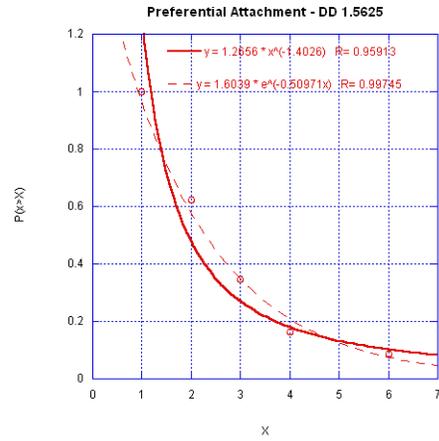
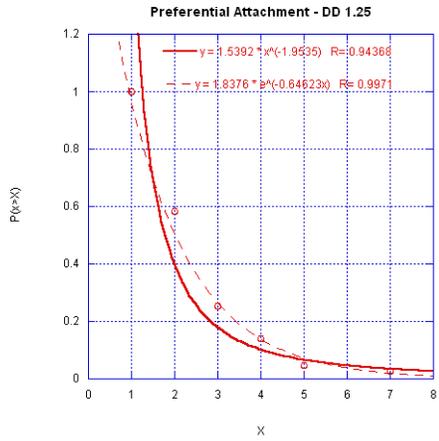


FIGURE C.5. The graphs and fitted curves for Preferential Attachment Model with variable Death Rate

