LEARNING GENETIC REGULATORY NETWORK CONNECTIVITY FROM TIME SERIES DATA

by

Nathan Barker

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

December 2007

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a dissertation submitted by

Nathan Barker

This dissertation has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

	Chair: Chris J. Myers
	James P. Keener
	Robert R. Kessler
	Ellen Riloff
	Gil Shamir

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the dissertation of Nathan Barker in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date

Chris J. Myers
Chair, Supervisory Committee

Approved for the Major Department

Martin Berzins
Chair/Dean

Approved for the Graduate Council

David S. Chapman
Dean of The Graduate School

ABSTRACT

Recent experimental advances facilitate the collection of time series data that indicate which genes in a cell are expressed. This information can be used to understand the genetic regulatory network that generates the data. Typically, Bayesian analysis approaches are applied which neglect the time series nature of the experimental data, have difficulty in determining the direction of causality, and do not perform well on networks with tight feedback.

To address these problems, this dissertation presents an improved method, called the `GeneNet` algorithm, to learn genetic regulatory network connectivity which exploits the time series nature of experimental data to allow for better causal predictions on networks with tight feedback. More specifically, the `GeneNet` algorithm provides several contributions to the area of genetic network discovery. It finds networks with cyclic or tight feedback behavior often missed by other methods as it performs a more local analysis of the data. It provides the researcher with the ability to see the interactions between genes in a genetic network. It guides experimental design by providing feedback to the researcher as to which parts of the network are the most unclear. It is encased in an infrastructure that allows for rapid genetic network model creation and evaluation.

The `GeneNet` algorithm first encodes the data into levels. Next, it determines an initial set of influence vectors for each species based upon the probability of the species' expression increasing. From this set of influence vectors, it determines if any influence vectors should be merged, representing a combined effect. Finally, influence vectors are competed against each other to obtain the best influence vector. The result is a directed graph representation of the genetic network's repression and activation connections. Results are reported for several synthetic networks showing significant improvements in both recall and runtime while performing nearly as well or better in precision over a dynamic Bayesian approach.

To my loving family, who mean more to me than they will ever know.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

NOTATION AND SYMBOLS

$\rightarrow$	Graphical representation of activation	20
$\dashv$	Graphical representation of repression	20
S	A vector containing the species in a genetic network	30
E	A time series data experiment	30
$\|E\|$	The total number of data points within E	30
e	The experiment number	30
τ	A time point in an experiment	30
ν	A vector over S which includes the state of each species	30
$\nu(s)$	Indicates the value of species s for that vector	30
H	A mutational high state	30
L	A mutational low state	30
$-$	Indicates that the state of the species should be measured	30
$SUCC$	The set of successor time points	30
$PRED$	The set of predecessor time points	30
n	The number of bins used to discritize the data	31
θ	The numerical level assignments	31
$\theta_0(s)$	The lowest numerical level for species s	31
$\theta_n(s)$	The highest numerical level for species s	31
Φ	Every bin assignment for each species in S	31
$\Phi_i(s)$	Bin i for species s	31
$\Phi_0(s)$	The lowest bin assignment for species s	31
$\Phi_{n-1}(s)$	The highest bin assignment for species s	31
$\Phi_*(s)$	Any bin assignment to species s	32
b	An assignment of a single bin to each $s \in S$	31
$b(s)$	Inidcates the bin assigned to species s	32
$b \cup b'$	The union of two partial bin assignments	32
$b(s)++$	Increments the bin assignment of species s to the next highest bin	32
$b(s)--$	Decrements the bin assignment of species s to the next lowest bin	32
B	A set of bin assignments	32
CD	Compressed data	37
CD_s	The compressed data for species s	37
$CD_s(b)$	The increasing and occur information at bin assignment b	37
PCD	Compressed and projected data	38
$\mathcal{I}$	The influence vector function	41
IV	Shorthand for influence vector	41
i	An instance of an influence vector	41
$\mathcal{I}(c)$	Returns the vector of influences on species c	41

CHAPTER 1

INTRODUCTION

There is nothing more difficult to take in hand, more perilous to conduct, or more uncertain in its success, than ... the introduction ...
–Niccolo Machiavelli

Aristotle, born in 384 BC, is credited as being the first biologist [35]. There have been significant scientific advances since then. We are now able to monitor cellular activities with a variety of techniques [18, 8]. These techniques offer the potential of providing vast information about the interactions between proteins. The entire human genome has recently been sequenced and with it the ability to study these genes in greater detail than ever before [104]. As of yet, little is known about the interactions between individual genes in most genetic networks. This dissertation tries to help use experimental data to explain the activation and repression patterns that occur between genes found in living organisms.

1.1 Learning Methodology

Figure 1.1 shows our methodology. It has been adapted and extended from [21] to allow added functionality and illustrate items important to our methods. Many of the tasks in the figure are generally performed by hand, and we have developed techniques to help automate many of the steps in this process.

This section describes a few of the most important steps in Figure 1.1. Typically, biologists, when studying a particular genetic regulatory system, know some of the behavior that they expect from the system but they would like some deeper knowledge about how the system functions to either better model or better understand the behavior of their system. To obtain this knowledge, they perform some experiments upon their system to collect data. As shown in Figure 1.1, background knowledge coupled with the experimental data can be used to learn or construct a high level model of the regulatory system of interest. This model can then be simulated and predictions can be made about

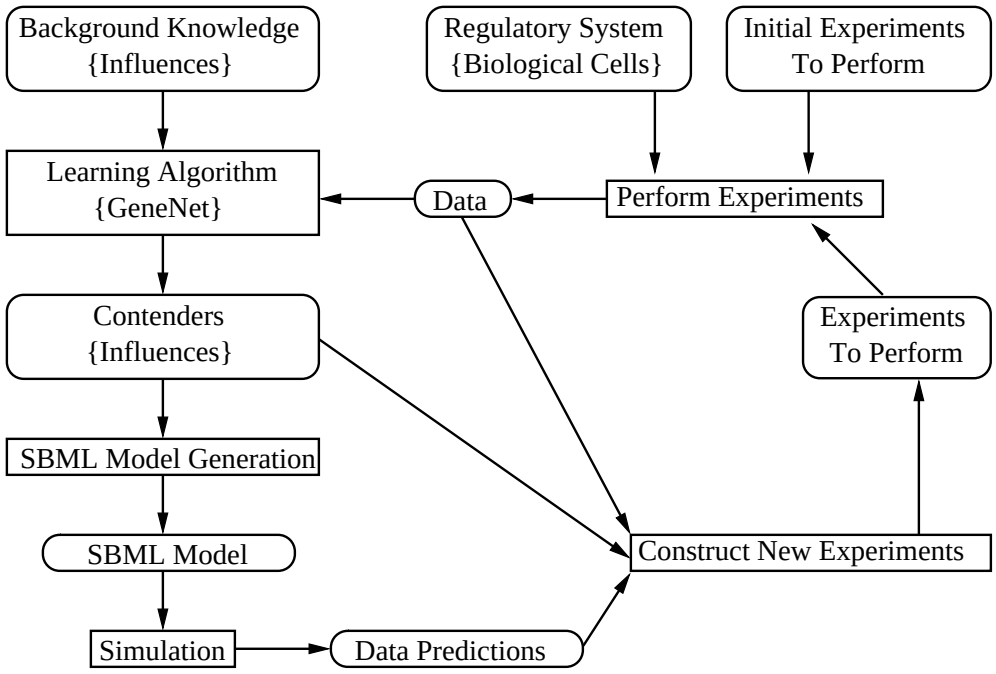


Figure 1.1. Analysis methodology in the study of genetic regulatory systems. The oval boxes represent data, knowledge, or biological cells, while the square boxes represent functionality, and the arrows represent data flow (adapted from De Jong [21]).

the system. These predictions are compared with actual experimental data to see how close the computer model is to the actual biological system. If the experimental data and the predictions are not the same, new experiments can be suggested that might help clarify uncertain areas of the model and the process can be repeated.

One of the goals of our work is to automate many of the steps in Figure 1.1. Our work allows biologists to input their initial background knowledge and experimental data into our tool. Our tool constructs models from the data provided. It presents biologists with the model that it constructs. It also suggests possible experiments that could improve the model. A further advantage to having a model of the system is that a biologist is also able to simulate the model under different conditions to evaluate how the model could behave in different environments or respond to different stimuli from those that can easily be obtained from experiments. The model also improves observability into the inner workings of the system of interest.

1.2 Previous Learning Methods

There are many different and varied ways in which to use data collected from genetic networks to try and understand the relationships between species in the networks. These methods vary as to the types of information used as well as to the depth of the interactions they try and understand or model. This section describes some of these approaches in more detail. Werhli *et al.* compare several of these methods directly [112].

1.2.1 Clustering Approaches

Clustering is a method of grouping genes together that have similar expression patterns. If two genes' expression levels rise and fall at approximately the same times, they are grouped into a common cluster. There are various approaches to clustering. One approach is to select the number of clusters a system should have, and let an algorithm decide where to place the dividing lines. A more advanced approach is to allow an algorithm to decide the number of clusters, and where the dividing lines should be drawn.

Eisen *et al.* describe how clustering can be used for analysis of gene expression data [24]. More specifically, they use *hierarchical clustering* for obtaining information about genes. They claim that this type of relationship is more intuitive to biologists who would use their method, as it is closely related to sequence and phylogenetic analysis of genomes. Another reason for their choice of hierarchical clustering is that they can graphically represent the results obtained from applying their method. An example of the results of a clustering analysis is shown in Figure 1.2.

Kundaje *et al.* use a function based on statistical splines to determine if two genes should be grouped into a cluster [61]. After they have grouped the genes into clusters they then compute causal relationships between the clusters by using a joint probability distribution and variable time lag to infer how independent the cluster distributions are from each other. An advantage of their method is that they do not break the expression levels of the clusters into discrete levels, but rather use continuous quantities when calculating the causal relationships. They also use time series data rather than just correlational data. They applied their methods on time series data obtained from the yeast cell cycle and obtained several clusters and interactions between those clusters. They decided to give meaning to their clusters in the framework of cell cycle regulation. They know from the literature which genes and therefore clusters are involved in which phases of the cell cycle. They explain how some of their cluster interactions support a

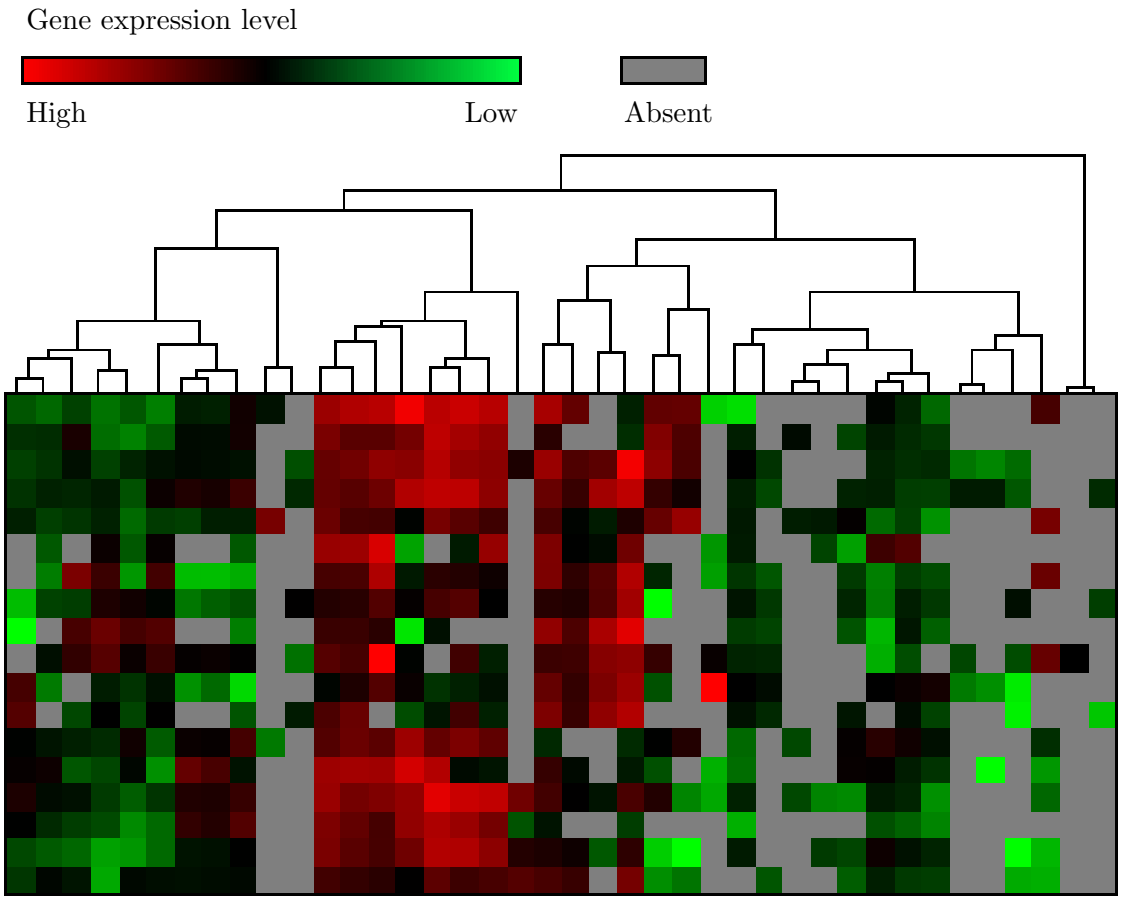


Figure 1.2. Hierarchical clustering of time series microarray data. This analysis was generated using NCBI's Gene Expression Omnibus.

model found in Simon *et al.* of the transcriptional control of genes for each phase of the cell cycle [100].

Guthke *et al.* reduce the number of species in one system from 18,432 to 6 using clustering [38]. They then select one species from each cluster to represent the group. Finally they use this reduced system to try and understand the relationships between the selected species.

Hastie *et al.* extend traditional clustering methods [43]. One such extension allows genes to belong to more than one cluster. Their method may be either supervised or unsupervised. Their clustering approach searches for clusters showing both high variation across sample as well as correlation across genes. They call their approach 'gene shaving'

as they select small clusters of genes that vary as much as possible across the samples.

A drawback of clustering is that only coarse grain information between sets of genes is obtained. This means that no information is found between individual genes. Another drawback is that for some clustering algorithms, the user must make a guess as to how many clusters there should be. Also, it is very difficult to judge the correctness or meaningfully interpret the results of cluster analysis as clustering simply groups genes that seem similar in some way.

1.2.2 Regression Techniques

Regression techniques are a statistical approach to understanding the relationships between dependent and independent variables. Regression usually involves solving, or best fitting, a system of equations that govern or model the behavior of the dependent variable. There have been several applications of this method. Yeung *et al.* break the problem of learning genetic regulatory networks into two parts [116]. They first use singular value decomposition to construct a set of feasible solutions. After a set of solutions has been computed, they use robust regression techniques to select the sparsest network from the set. They assume that the data they use for singular value decomposition comes from perturbing the system and then collecting data as the system relaxes back to its normal state.

Rogers and Girolami use a Bayesian algorithm designed to provide sparse connectivity which uses regression to construct networks [95]. They assume that the data they use comes from first having the system at *steady state* then performing knockouts and observing the results. Steady state experiments are those in which the system is assumed to have settled into a state where there are only minor fluctuations in molecule counts per species. They claim that by using a regression algorithm the data does not first need to be discretized but the continuous nature of the data is preserved. Also, their Bayesian approach breaks the learning task down into learning the influencing species for each species so that no discrete topology search is required as is usual for standard Bayesian approaches.

1.2.3 Mutual Information

Mutual information is a statistical approach that measures how dependent two variables are upon each other. It is defined by taking the sum of the entropy, or average of

the log probabilities of its states, of the two variables and subtracting the entropy of the conditional mutual information. Entropy can be either discrete or continuous [11].

Basso *et al.* use mutual information to identify statistically significant connections between species [11]. They then eliminate indirect relationships in which two genes are co-regulated through one or more intermediaries by applying the 'data-processing inequality'. They have applied their method to reverse engineer the regulatory networks in human B-cells.

1.2.4 Functional Approaches

Functional approaches look for functions from genes to genes that would explain the behavior that occurs in the gene expression data. In its basic form, two different time slices where a gene has changed from a low value to a high one are used. The other gene expression levels are then used to determine which level changed, and it is assumed that this caused the change in the gene's expression level. These functional approaches often use a Boolean (*i.e.,* on / off) model of a gene's expression level. Figure 1.3 shows a simple functional representation of a genetic network.

Liang *et al.* developed an algorithm called REVEAL that tries to reverse engineer the genetic network from gene expression profiles [68]. The REVEAL algorithm constructs Boolean functions for each gene in the system by analyzing the mutual information between input and output states to infer the connections between species in a network. It determines the set of parent genes that influence each child gene by looking through the state transitions for the smallest input function of the parents to explain the child gene's behavior. If the entire state transition table is known, an exact function of each gene based on the parent genes can be determined [68].

Current State		Next State	
Gene A	Gene B	Gene A	Gene B
0	0	0	1
0	1	1	1
1	0	0	0
1	1	1	0

Figure 1.3. A simple functional representation of a genetic network where gene B activates gene A and gene A represses gene B.

Akutsu *et al.* later extended the work of Liang *et al.* [2, 68]. They give a mathematical proof for the work of Liang *et al.* as well as developing a similar, but equally powerful algorithm for the same problem. They use an exhaustive search in order to prove an upper and lower bound on the time that their algorithm would take to determine the genes that affect a given gene. They bound their problem by stating no gene can be controlled by more than three other genes. This limits the amount of searching they must do. They state that the time and memory space of their algorithm may be worse than REVEAL, but because of the simplicity, it is easier to develop a proof for its correctness.

Ideker *et al.* describe methods for inferring a Boolean genetic network through mutational experiments [51]. For each gene, they look in the table of expression level data for rows where the expression level varies. They then apply a minimum set covering algorithm to determine the smallest set of genes that would explain the observed differences. This set of genes then forms part of the gene's Boolean function. The entire graph is made when the minimum set for all genes has been found. The input data used by Ideker *et al.* differs from those of Liang *et al.* and Akutsu *et al.* in a major way. While Liang *et al.* and Akutsu *et al.* take the state transition table of a network as input, Ideker *et al.* take data obtained from mutations. Their data are obtained by first setting the cell into a state, and then waiting until the cell enters a steady state some time later. They then assume that any changes in the steady state must have arisen from the mutational changes, as many biological systems do not have a single steady state. The decision circuit for the bacteriophage λ for example has two. Their assumptions may not be valid for all genetic networks. After the steady state data are obtained, their algorithm then searches the data for the conditions where a child's gene expression differs. It then determines which parent genes also differ in these situations and looks for the smallest parent set that explains this change.

Lähdesmäki *et al.* apply two solutions to problems used in the machine learning domain to discover the interactions between genes [65]. The two problems are the Consistency and Best-Fit Extension problems. Consistency refers to the problem of creating or finding functions that can perfectly classify the behavior of a system. The Best-Fit problem is much like the Consistency problem, except that instead of trying to find a function that can perfectly classify the behavior of the system, it tries to find a function that makes the fewest mistakes possible in its classification. The solution to either one of these problems can also be seen as a solution to the problem of discovering

a genetic network. In particular, the output of the discovered function would describe the child gene's on / off behavior in response to the potential parents (*i.e.,* the inputs to the function). They claim this is faster than the method used in Akutsu *et al.* [2]. They also try and return all possible networks that explain the data as they try and compensate for noisy and sparse expression data. They have evaluated their method on the data set found in Spellman *et al.* [101] and claim to agree with most of the known regulatory behavior. However they state that the probability of finding a good predictor in Spellman *et al.* [101] by chance is fairly high.

There have been other interesting functional approaches and insights. McAdams and Shapiro give insight into large circuit analysis and how Boolean models can analyze large genetic networks [76]. Albert and Othmer show how a Boolean model of the segment polarity genes in *Drosophila melanogaster* can be analyzed to correctly reproduce the genes expression patterns [3]. They also show how the Boolean model can account for mutations in the network.

One of the drawbacks of functional or Boolean methods is that they often do not allow for multiple levels of gene expression. An example of this would be a gene that at very high concentrations interferes with cellular activities. It would need at least three levels to be correctly interpreted. The first level would be off, the second would be normal concentration, and the third would be extremely high. This three level abstraction of the gene could not be done with normal Boolean models. There is some work that allows multivalued models. Thieffry and Thomas, for example, extend regular Boolean models to show how a multivalued Boolean model of the bacteriophage λ can be analyzed [105]. This technique helps to combat the effects of having too few levels of expression.

1.2.5 System-Theoretic Approaches

Where most other approaches take a bottom up approach at learning genetic regulatory connectivity (*i.e.,* at the species to species level), system theoretic approaches take a top down approach. These approaches focus on high level network modules and try to learn the connection between these modules and ignore the interaction that occurs inside modules. This assumes some basic knowledge about the modules and the ways in which modules interact. For example in modeling interaction between cells, the intercellular molecules can be ignored and the intracellular molecules are used for communicating intermediates. Cluster analysis is one potential way in which to identify top level modules.

Kholodenko *et al.* describe how they use systematic perturbation to try and learn the

connections between the modules [57]. They place a system into steady state and then use inhibitory, activation, and changes in external signals to perturb every module while measuring the global responses to the changes. They apply their method to the MAPK cascade network as well as to an example four species network.

Andrec *et al.* use the algorithm and network assumptions in Kholodenko *et al.* and explain the theoretical effects of experimental uncertainty from differing noise levels and the strength of the interactions on the learned network [5, 57]. They describe a geometric tool that can validate the known connections between species and even validate the accuracy of this type of influence. They show that more data increases the accuracy of their inferred network and that added noise decreases their accuracy. They use an example of a MAPK cascade where there are a few high level modules and a few communicating intermediates.

Pilpel *et al.* use genomic expression data to try and understand the combinatorial nature of eukaryotic transcription [88]. They study the interaction between gene modules and focus on combinatorial effects. They provide their learned motif map of *Saccharomyces cerevisiae*. Segal *et al.* also study *Saccharomyces cerevisiae* and try to learn both the network motifs and their regulators by using genomic expression data [98].

1.2.6 Partial Correlations

Partial correlations are a statistical technique used to evaluate how correlated two variables are when their relationship with at least one more variable is excluded. Using an example from genetic regulatory networks, assume that species A activates species B. If the partial correlation is taken with respect to an unrelated species C, then species A and species B should have a strong correlation. If, however, species C activates both species A and B, the partial correlation between species A and B when species C is excluded should be very low. Partial correlations are often expressed as *Graphical Gausian Models*.

Rice *et al.* use conditional correlations between genes from steady state data [93]. They perturb each species in turn to a value of zero and then track the value of each other species in order to infer the genetic network connectivity. They perform 10 experiments per perturbation, along with 10 experiments where no species is perturbed. This gives a total of $10 * (N + 1)$ experiments that need to be performed for their method to give a reasonably low error rate. They study how the effects of noise, network topology, size, sparseness, and dynamic parameters influence the ability of their method to correctly infer

the genetic network. They apply their method to the transcriptional control network of *Escherichia coli* (E. coli).

Willie *et al.* use a modified graphical Gaussian modeling approach to improve upon the fact that there can be little absolute pairwise correlation but high absolute partial correlation between the expression level of species in a genetic network [114]. They perform partial correlation by selecting two species and take a partial correlation with respect to every other species in the network separately. If the expression level of no other species in the network can explain the correlation they include an edge in the graph. They claim that by taking partial correlations with only three species this helps improve the results obtained using small data sets relative to the number of species. They have applied their method to the isoprenoid gene network in *Arabidopsis thaliana*.

Schafer and Strimmer suggest several problems with using standard graphical Gaussian models [97]. The first one is that as the number of species is large compared to the amount of experimental data, partial correlations cannot be reliably computed because the sample covariance and correlation matrices are not positively defined. They also suggest that the linear dependencies between species leads to the problem of multicollinearity. They have developed three new small sample point estimates to use when calculating the partial correlations between species. They have applied their method to a breast cancer data set.

1.2.7 Bayesian Approaches

Bayesian methods are an approach that relies mainly on the probabilities of genes' expression levels being correlated. This can be done by considering each gene as a variable in a joint *probability distribution function* (PDF). If knowing information about gene A tells you information about gene B, the genes are said to be correlated, or one gene is a (potential) parent of the other. If information from the knowledge of gene A's expression level does not give any information for gene B's expression level, the genes are said to be unrelated. In other words, if two genes are always high or low together, then they are correlated. If one is high and the other low, they are inversely correlated. This information helps to determine the probability of two genes being related.

Many different Bayesian network graphs can imply the same set of dependencies. Equivalence classes are used to imply that two different graphs have the same undirected graph structure. It is impossible to distinguish between equivalent graphs when examining observations from a distribution. This means that given a statistical dependence between

the activities of proteins X and Y, this method cannot determine whether X activates Y or whether Y activates X [85]. For a more detailed description of Bayesian Networks see, for example, Heckerman [44]. For more information on how they are used to model and learn genetic networks see, for example, Pe'er [85].

Friedman *et al.* use Bayesian approaches to learn genetic networks [27, 26]. As it is often difficult to determine the direction of interaction with such a model, they extend their model to causal networks. These are networks where the direction of interaction is important. They then search the state space to find a set of possible models to explain the data. As they are using Bayesian networks, they are unable to represent cycles in the networks. In other words, they are unable to find a network in which two genes repress each other.

Hartemink *et al.* extend the basic Bayesian network model to allow the labeling of edges [42]. These edge labels can either represent a default influence, activation, repression, or an influence that is either positive or negative but whose true relationship is not known. This extra annotation on Bayesian networks helps describe the effects of each parent individually and not as a whole.

Sachs *et al.* have recently applied Bayesian methods to intracellular multiparameter flow cytometry with much success [96]. They collected data on 11 of the proteins and lipids which are active in human T-cell signaling and are able to correctly infer many of the existing causal relationships. They do, however, get the direction of one relationship reversed. They explain how interventions are usually necessary for them to discover the direction of influence between proteins in the network but show an example where they correctly infer the correct direction of an unperturbed protein. Figure 1.4 shows an example of what the data might look like using flow cytometry with two particular genes in the phage λ decision circuit. Notice the correlation of the genes. These data may be used in a Bayesian analysis to determine if these genes are related.

Unfortunately, Bayesian approaches have several limitations. First, Bayesian approaches often have difficulty determining which gene is the parent and which is the child. A drawback of this method is that it relies on statistical analysis, so it is very sensitive to small amounts of data. That is, statistical anomalies found in small amounts of data, but averaged out with enough data, can drastically change the analysis. Second, these approaches rely primarily on correlational data and only use time to separate the data points. Third, these approaches do not learn networks that are cyclic. This last one

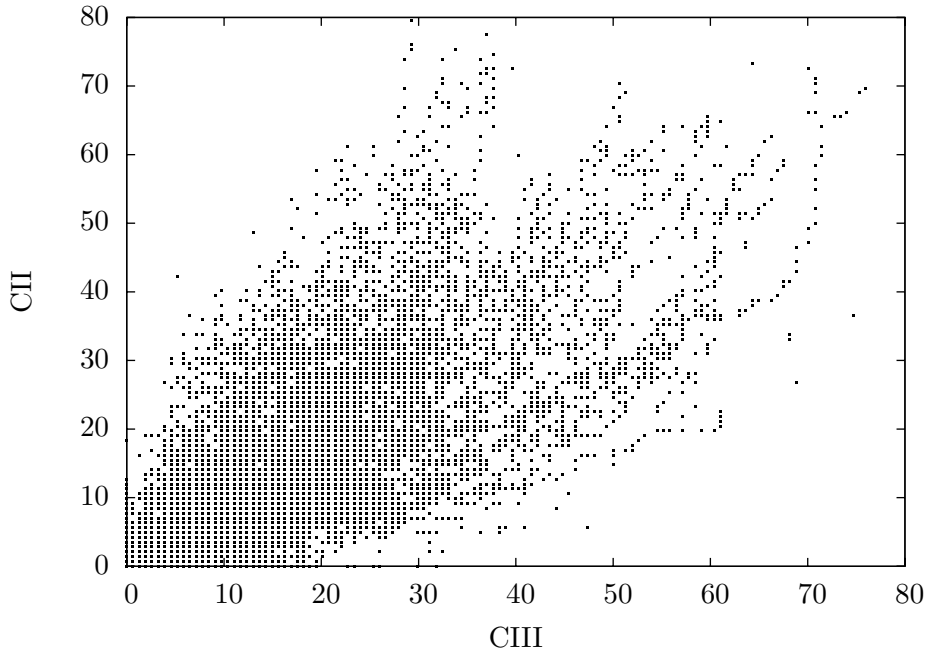


Figure 1.4. Simulated data from the phage λ decision circuit comparing the expression level of CII and CIII.

we consider a major limitation since feedback control is a crucial component of genetic regulatory networks.

To address some of the limitations of Bayesian networks, *Dynamic Bayesian Networks* (DBN) have been introduced which incorporate an aspect of time to allow networks with cycles to be found [82, 81, 27, 26]. Although these networks allow cyclic behavior to be found, they still have the other limitations of Bayesian Networks.

Nachman *et al.* use DBNs to learn genetic regulatory networks [80]. They also incorporate prior biological knowledge in the solution that they learn. Finally, they model the kinetic parameters that govern the transcription rates for gene expression. Thus, they combine both qualitative and quantitative approaches into their approach. They first learn a DBN model of the system. They then try and learn the kinetic parameters and the hidden activity levels of the regulators from the observations. They applied their method to a series of examples related to transcriptional regulations in the yeast cell cycle.

Husmeier test DBNs by trying to realistically model and simulate the networks and

data that are currently available in a simulation study [49]. They conclude that the global network inferred from the data is meaningless, as the data are too small to accurately infer the correct network. They also claim that using data from perturbed systems that are relaxing gives better information, as compared to collecting data from networks at a steady state.

Bernard and Hartemink incorporate multiple types of data when using DBNs to learn genetic regulatory networks [13]. They use both gene expression data as well as transcription factor binding location data. They evaluate their method using a synthetic cell cycle model, where in each of the three cycles a different network is active. They also use their method on real data from a genetic network found in yeast. They state that using two types of data improves their method over using data solely from each type alone.

Beal *et al.* use state-space models, which are a class of DBNs in which the observed measurements depend on some hidden state variables [12]. They state that two genetic network models with equivalent gene-gene interactions but different implementations in terms of hidden variables could have large differences in expression profiles as the hidden variables may play a large role over time. They also state that when learning network structure from actual networks, as opposed to simulated networks, the results of the learned method must be experimentally verified by perturbation experiments to assure that the learned network actually correctly models the real network. These experiments are often difficult to perform in a systematic way.

An example of a Bayesian analysis that is very similar to our work is the work by Yu *et al.* [117]. They use two different Bayesian scoring functions to find the best matching network to their data. They split the learning of genetic networks into two parts. The first part is to find potential parents for a gene. The next is to evaluate the potential parents to determine the most likely candidates. They do this by first generating a DBN to limit the number of parents considered for a gene. The main reason for splitting the approach into two is that the second step is exponential in the number of parents. Next, the algorithm assigns an *influence score* to each potential connection. To find this score, they build a *cumulative distribution function* (CDF) for the child's potential parents. A CDF is basically a set of bins that indicate the number of times the genes are seen in a given configuration or an earlier one. For example, a configuration may be that proteins produced by all genes are at their highest state. If the CDF shifts in the positive

direction as the parent's value increases, the parent is considered an activator. If the CDF shifts in the negative direction, then the parent is considered a repressor. Finally, if the CDF does not shift, within some bounds, then the parent is considered for removal. As there can be many different parents and the data is noisy, each case contributes a vote on the likelihood of activation or repression. To evaluate their method, they generate synthetic data for 10 genetic networks that include 20 genes in which 8 are unconnected. Using this synthetic data, their method is shown to be fairly successful at recovering the initial networks given sufficient data. They still do, however, occasionally have problems determining the direction of influence. Also, only two of their networks have any feedback (*i.e.,* cycles), and these cycles are several genes long.

Some limitations of the method in Yu *et al.* are that it begins with an expensive search for a best fit network. As the state space is exponential in the number of genes, this can be very expensive. The method in Yu *et al.* also neglects the time series nature of the data when it computes a CDF. However, the information between time points is very helpful in determining the genetic network especially in the case where there is tight feedback in the network. In other words, not only the context of when a species is high or low gives information, but more importantly the context in which it rises.

1.3 Model Simulation

Once a model has been learned, it can be simulated and further evaluated. There are two main approaches to biological model simulation. The first is *classic chemical kinetics* and the second is *stochastic chemical kinetics*. This section describes these two approaches in more detail.

1.3.1 Classical Chemical Kinetics

Many traditional temporal behavior analyses of genetic regulatory networks rely on a set of *ordinary differential equations* based on *classical chemical kinetics* formalism. *Classical chemical kinetics* models the continuous dynamics of coupled biological reactions [42]. There are several assumptions that are made about cellular activity using these models. One assumption is that a system is *well-stirred*, or that every molecule in a system can easily come into contact with every other molecule. Another is that the number of molecules in a cell is high. Another is that molecular concentrations can be viewed as continuous variables. Based on these assumptions, reactions in the cell can be viewed as occurring continuously and deterministically.

Stark *et al.* give an example of how differential equations can be used to model genetic regulatory networks [102]. This includes both linear and nonlinear approaches. They describe how these types of methods can be learned using perturbations to the network. They describe the types and amounts of experiments that should be performed for optimal network reconstruction. They explain how the collection of data usually requires multiple networks, and how the different networks do not necessarily grow at the same rate which would hinder the network reconstruction. Gouzé and Sari describe how they use piecewise linear differential equations to model biological systems that contain switch-like elements [36]. De Jong *et al.* describe a qualitative approach to the simulation of genetic regulatory networks [22].

Many of the assumptions for classical chemical kinetics do not hold when trying to model genetic network reactions. For example, genetic regulatory networks often involve species with small counts, like a single strand of DNA. Also, proteins produced from gene expression are produced in a discrete and not a continuous manner. Gene expression can also have substantial fluctuations in the gene products which often are not produced deterministically. Classical chemical kinetics may not be appropriate in such situations.

1.3.2 Stochastic Chemical Kinetics

To more accurately predict the temporal behavior of genetic regulatory networks, the *stochastic chemical kinetics* formalism can be used. These methods probabilistically predict the dynamics of biochemical systems. They describe the time evolution of a system as a discrete-state jump Markov process governed by the *chemical master equation.* Like classical chemical kinetics, this type of approach assumes that a system is well-stirred, but does not assume that molecule counts are high or that molecular concentrations can be viewed as continuous variables. One example where this approach is better is that often a random number of proteins are produced from an active promoter in short bursts at random time intervals [73, 75].

As it is often infeasible to obtain analytical solutions of chemical master equation models of any realistic system, an exact numerical realization of the system is performed to infer temporal behavior, often via Gillespie's *stochastic simulation algorithm* (SSA). This approach comes with potentially unlimited controlling capabilities and abilities to capture virtually any dynamical properties.

Arkin *et al.* perform a stochastic kinetic analysis on a pathway in the phage λ virus as it infects E. coli cells [7]. They describe how conventional deterministic kinetics cannot

be used to predict the virus' behavior as there are highly erratic time patterns of protein production. They go on to explain how by using stochastic kinetic analysis they can predict the statistical regulatory outcome of the infected cells.

One problem with stochastic simulation algorithms is that the computational requirements can be substantial. Often hundreds to thousands of simulations must be done to accurately predict a system's average behavior. Gibson and Bruck developed a method to simulate these systems that reduces the time complexity to be logarithmic in the number of reactions, without a loss of precision [30]. Kuwahara *et al.* have developed methods to allow for automatic abstraction of these systems which they show can reduce the computational requirements by several orders of magnitude at very little cost of accuracy [62, 63, 64].

1.4 Contributions

The major focus of this dissertation is the `GeneNet` algorithm. This algorithm provides several contributions to the area of genetic network discovery. It finds networks with cyclic or tight feedback behavior. It provides biologists the ability to use time series data to discover genetic network models. It provides the researcher with the ability to see the interactions between genes in a genetic network. It guides experimental design by providing feedback to the researcher as to which parts of the network are the most unclear. It is encased in an infrastructure that allows for rapid genetic network model creation and evaluation.

Most genetic networks have tightly controlled network behavior. A major reason for this is that if genes are only limited in their ability to produce proteins by the cellular resources, other cellular activities could become compromised and the cell could be damaged or destroyed. Other genetic network learning methods, such as Bayesian methods, do not allow cyclic behavior to be discovered or only allow limited long feedback loops. The `GeneNet` algorithm is designed to discover networks with tight regulatory behavior.

Most other approaches to learning genetic networks either discard the time information from time series data, or use other forms of data, such as steady state data, to understand genetic networks. The `GeneNet` algorithm is designed to focus specifically on the timing aspect found in time series data experiments. Using an aspect of time helps improve the quality of our results.

The `GeneNet` algorithm provides the user with a directed graph representation of the genetic network supported by the time series data. This convenient representation allows the researcher to see what genes are connected to which others. This representation can be easily converted into other forms, like a table, if the network is too large to be properly displayed. Allowing the researcher to see the results obtained by the algorithm provides the researcher with the ability to understand the network of interest.

The `GeneNet` algorithm provides information to the researcher as to what areas of the genetic network are the most unclear. This information tells the researcher which experiments would be most beneficial to perform. This saves the researcher both time and money as performing experiments that would yield little additional information to the algorithm are both costly and labor intensive.

In creating the `GeneNet` algorithm, we needed to create a tremendous amount of infrastructure to provide us with easy evaluation of our method. Although this proved very useful to us, it can also be useful to others who wish to either generate genetic network models rapidly or generate data from simple genetic network models. This infrastructure is integrated with a simulator that supports both deterministic and stochastic analysis to assist a researcher to see the protein evolution of each species as well as providing the ability to see what effects may occur with changes to the genetic network.

1.5 Thesis Overview

The rest of this dissertation is organized as follows:

Chapter 2 gives a brief overview of genetic regulatory networks. It also describes several current types of experimental data gathering techniques. Finally, it presents how these data are represented for efficient analysis.

Chapter 3 describes the influence vector notation used to represent the influences between species in a genetic network. It also describes the process of determining how likely an influence vector represents the experimental data as well as optimized implementations of this process. Finally, it presents a method for making the most out of sparse data.

Chapter 4 describes the `GeneNet` algorithm. It describes how the algorithm determines which species should be included in an influence vector to best represent the data. It also describes how influence vectors can be compared against each other to determine the most likely influence vector representation based on the data provided.

Once influence vectors are determined for a genetic network, a model can be created. Chapter 5 describes how a high level description of a genetic network can be translated into a very detailed reaction based model. This model includes many reactions that are abstracted in the influence vector representation.

As with many machine learning techniques, anomalies in the data, especially with sparse data, can lead to low information content. Chapter 6 shows how the `GeneNet` algorithm informs the experimenter which parts of the model are in question based on the data. It also explains how the `GeneNet` algorithm suggests experiments to help improve the model.

Chapter 7 describes how the `GeneNet` algorithm is evaluated. In order to properly evaluate the `GeneNet` algorithm, it must be tested on known networks. As there are few genetic networks where the connectivity is known, synthetic data are generated from a variety of networks for evaluation purposes. This chapter presents results varying a number of the algorithm's parameters to show how the default parameters are selected. As data can be collected in many different ways, this section describes some of the variations used when generating data. Finally, this chapter concludes with a comparison to a DBN approach and presents an analysis of the cases where our method obtains an incorrect answer.

Chapter 8 describes how this work could help biologists to better understand the networks they are interested in. It also describes some of the ways in which our work has further advanced this understanding. It summarizes our work, as well as suggesting further improvements that could be undertaken.

CHAPTER 2

EXPERIMENTAL DATA

Mathematics may be compared to a mill of exquisite workmanship, which grinds your stuff of any degree of fineness; but, nevertheless, what you get out depends upon what you put in; and as the grandest mill in the world will not extract wheat-flour from peascods, so pages of formulæ will not get a definite result out of loose data.

–Thomas Henry Huxley [50]

This chapter gives a brief overview of genetic regulatory networks and describes several experimental techniques used on these networks to collect data. It also gives a formal description of time series experiments as well as the data collected from them. Finally, it describes methods used to discretize and compress this time series data.

2.1 Regulatory Systems Overview

Figure 2.1 shows the molecular interactions between genes *CI* and *CII* in the phage λ decision circuit. The DNA portion of the figure contains the genetic sequences that are a blueprint for producing *proteins* from the genes, represented as black boxes. The DNA also includes regions called *promoters*. Our example includes two promoters, P_R and P_{RE}, represented as bent arrows indicating the direction of transcription. *Transcription* begins when an enzyme called *RNA polymerase* (RNAP) binds to the promoter. It then proceeds along the DNA strand. As the RNAP molecule proceeds along each new nucleotide of the DNA, a complementary monomer of *Ribonucleic acid*, binds to a growing chain, forming a complementary strand called *messenger RNA* (mRNA). After the RNAP molecule reaches a stopping point along the DNA, the next step that occurs is when this mRNA molecule comes into contact with a *ribosome*. The ribosome *translates* the mRNA into a functional peptide or protein that is coded for by the gene. In the example, the mRNA molecule is translated into the CII protein. A gene is said to be *expressed* if it is being actively transcribed, indicated by the presence of mRNA synthesized from that gene.

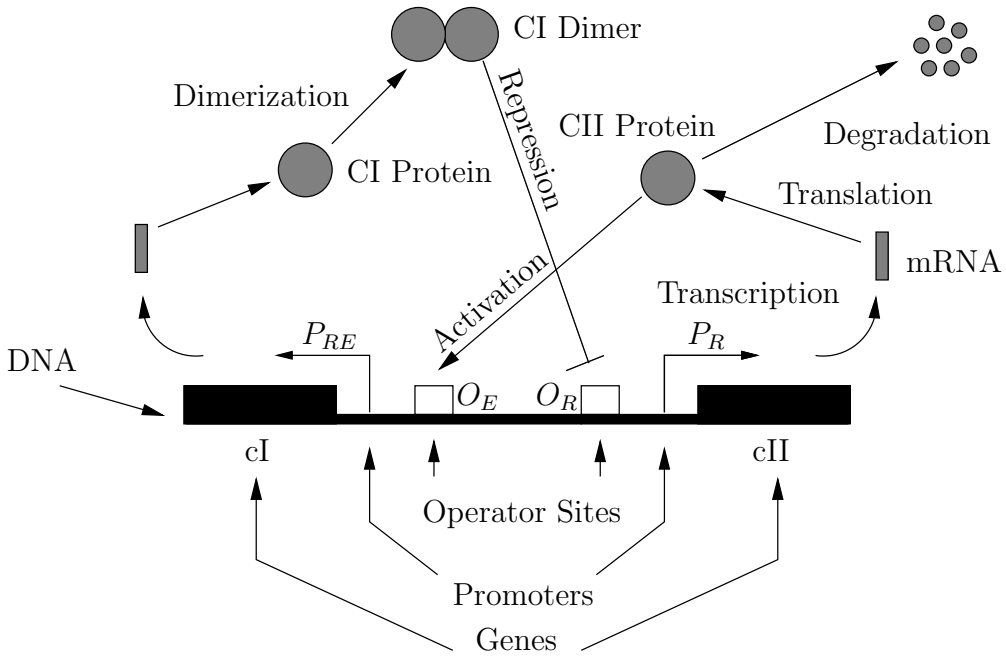


Figure 2.1. A portion of the genetic network for the phage λ decision circuit.

Proteins can bind to operator sites to either *activate* or *repress* transcription. There are two *operator* sites O_E and O_R shown in Figure 2.1. In the figure, activation is represented as a '$\rightarrow$' and repression is represented as a '$\dashv$'. The '$\rightarrow$' is also used to show the flow of molecules. If a gene does not need to be activated, RNAP can freely bind to the promoter and begin transcription. If a gene is activated, RNAP usually binds and begins transcription only after the gene is activated.

In this example, RNAP binds to the P_R promoter. It then transcribes the *cII* gene, producing a molecule of mRNA. This molecule is translated by a ribosome to produce a CII protein. Next, a CII protein can bind to the O_E operator site to activate transcription of the *cI* gene from the P_{RE} promoter. Before it can act as a *transcription factor*, two CI molecules must bind together to form a *dimer*. This dimer molecule can bind to the O_R operator site and prevent RNAP from binding to the P_R promoter. This inhibits, or represses, the transcription of the *cII* gene. Once transcription of the *cII* gene stops, then both the mRNA for the *cII* gene and the CII proteins are no longer being created and degradation eventually destroys these species. Without the CII protein bound to the O_E operator site, the gene *cI* is no longer actively transcribed and degradation eventually

removes most of the CI proteins. Without the CI dimer to bind to the O_R operator site, RNAP can bind to the P_R promoter and the entire process starts again.

Most cellular events occur by chance encounters of molecules. For example, mRNA degradation can occur quickly or slowly depending upon chance encounters with *Ribonuclease* (RNase). The longer the mRNA stays around, the more likely it is to produce many proteins for that particular gene. One strand of mRNA can produce from 0 to more than 10 proteins. It depends on the number of ribosomes it comes into contact with that translate it before it degrades.

As biological interactions rely on chance encounters between chemical species, activation and repression patterns are also not perfect. Even though the CII protein activates gene *cI* in our example network, there is still a chance that gene *cI* can be transcribed without the CII protein being present. This chance is usually referred to as the *basal rate* of production and would typically have a much smaller chance of occurring than would the activation rate when the CII protein is present. The same type of thing can happen with repression. The CI dimer represses gene *cII*, but there is still a chance that even with the CI dimer present in large quantities it may not be bound to the O_R operator site and the *cII* gene can be transcribed. For more information on the phage λ decision circuit, please see [89].

2.2 Experimental Methods

As Figure 1.1 shows, experiments are performed on regulatory networks to gather data. There are currently several different ways in which experiments are performed to gather data on regulatory networks. Some of the most common in use today are *cDNA microarrays*, *fluorescence*, *mass spectrometry*, DNA *sequencing*, and *genome-wide location analysis*. This section describes each of these types of experiments in greater detail.

2.2.1 Microarray Experiments

Microarrays are used to detect gene expression patterns in cells over the entire genome in parallel [119]. Microarrays work by hybridization of labeled mRNA in solutions to cDNA molecules attached at specific locations on a surface, such as glass, plastic, silicon, gold, a gel or membrane or even beads at the end of fibre-optic bundles [70]. Currently glass slides are the most common choice as cDNA can be deposited on this material in very high densities [69]. These slides are often referred to as a *chip*. Figure 2.2 shows what one such experiment looks like upon completion.

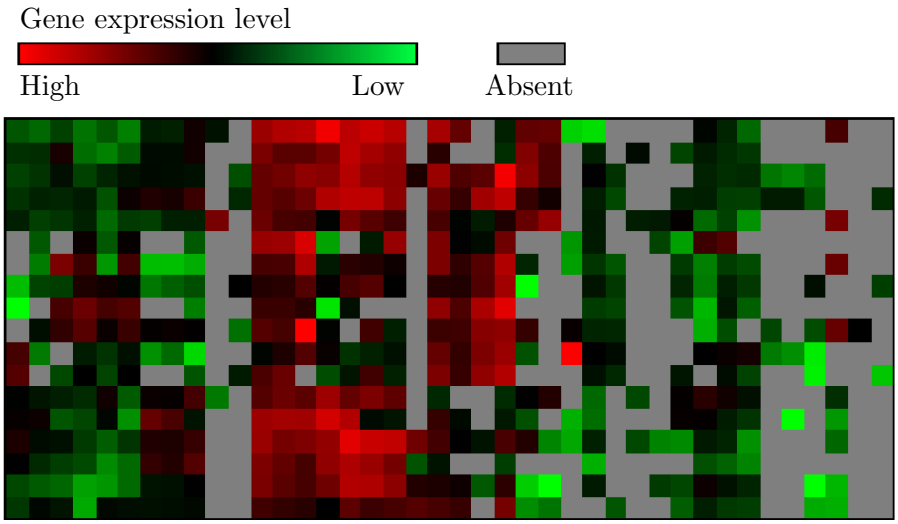


Figure 2.2. A microarray experiment generated using NCBI's Gene Expression Omnibus.

To determine genetic network expression patterns, typically many thousands of cells are exploded, and the contents are put onto the glass slide. The glass slide is then allowed to sit idle for an amount of time (typically overnight) to allow for the mRNA to bind to the cDNA on the glass slide. This slide is then viewed by a microarray scanner, which reveals the amount of matched sequences. This is often reported as some amount of fluorescent intensity as the mRNA was labeled with a fluorescent protein. The scanner then detects the intensity from each part of the slide, and by knowing where the gene markers are, infers the activity of a gene.

There are, however, problems with using microarrays to gather data. One of the first is that the cells must be exploded to obtain any data. This prevents the cells from being tested as they evolve over time. Another problem is that in order to obtain meaningful information from the microarray scanner, the molecules from many cells must be used at the same time. This limits the number of time points that can be measured by a microarray experiment based upon how many cells are in the experiment and how fast the culture grows. There are, however, methods being developed that start with small amounts of mRNA and use amplification, such as *polymerase chain reaction* (PCR), to increase the number of molecules [70]. Another problem is that microarrays generally measure mRNA levels, which do not exactly capture if a gene's products are currently

in the cell and more importantly are involved in any reactions such as activation or repression, but rather only measure if a gene is currently being expressed in the cell. The reason for this is that mRNA potentially degrades more rapidly than the molecules that are produced from it. Microarrays can also vary in sensitivity and specificity, as well as correct assignment and annotation meaning that care must be taken when creating gene probes for microarrays to assure their statistical significance and correct analysis [108]. The measuring of gene expression with microarrays can be prone to error. Huber and Vingron describe one way that this error can be modeled [48].

2.2.2 Fluorescence Experiments

Unlike microarray experiments, fluorescent experiments are able to identify individual gene expression as it occurs [4]. However they do not target the entire genome of a cell as do microarrays. This allows for very fine tuned information on specific gene expression, but no information on what other genes are active or inactive. Fluorescent experiments are also performed without exploding the cell which allows for the ability to see what changes occur to specific gene expression patterns over time. This is a distinct advantage over microarrays.

One of the most commonly used fluorescent proteins is a green fluorescent protein referred to as GFP. Typically a plasmid containing this protein is created and inserted into a cell. The plasmid is constructed such that if the gene of interest is active, the plasmid becomes active as well. This is often done by copying the activating sequence of the gene of interest and placing it directly before the sequence that encodes for GFP on the plasmid. One can therefore see the activity of a gene based on the intensity of green light emitted under fluorescent lighting.

There are a few other things about fluorescents that should be mentioned. The first is that the fluorescent protein that is monitoring a gene's expression level most likely degrades at a different rate than the protein it is trying to monitor. It would be ideal if it degraded at the same rate as the gene it is monitoring, but this is unlikely. In the ideal case, the fluorescent measurement indicates if the protein of interest is present in the cell. However, if the fluorescent protein degrades faster than the protein it is monitoring, the measurement indicates if a gene is being expressed, and not the protein level of that gene. If the fluorescent protein degrades slower than the protein it is monitoring, it indicates if the gene is expressed during the lifetime of the fluorescent protein.

Current fluorescent techniques do not allow for a large number of genes to be measured

at a time. This limitation is currently based on the number of different fluorescent proteins that can currently be produced as well as the ability of a cell to produce and contain extra proteins without interfering with its own functioning. Measuring a small number of genes seems to be a major drawback as there are many thousands of genes in many networks. However, to monitor only a handful of genes is still very useful. The activation and repression patterns from a single gene to other genes could be established with only a few colors and lots of experiments. Each experiment would contain the gene of interest and a few other genes that it may influence. From these data, it could be determined if the gene of interest appears to activate, repress, or not influence the other genes being monitored. Cause and effect relations could then be determined for the gene in question.

Another factor that needs to be taken into consideration when using fluorescent experiments is that advanced image processing methods must be used. This is less of a problem when only a single fluorescent protein is being monitored, as the cell either glows or it does not. However if there are a large number of fluorescent proteins and the experiment is being recorded with a video recorder, then each frame of the video must potentially be analyzed using image processing techniques for information on all of the proteins.

Gene Myers proposes to get the entire 4D gene expression data of a *Drospholia* from conception to some early state of development [79]. Gathering these data would be done as follows. An experiment would consist of a *nucleus dye*, and 8 gene or protein dyes. The reason he suggested using only 8 different dyes is that after 8 different color dyes the image produced would be too cluttered for image processing. Images would then be taken of the *embryo* during the entire maturation phase. The protein dye allows for exact visual conformation of when a protein first begins to be *synthesized*. By dyeing the nucleus, the closest nucleus, and therefore the exact cell that is expressing the gene can be determined. Naturally, many experiments would need to be performed as there are over 600 genes in *Drospholia*. Each gene would probably be included in several different experiments to add to the evidence that it is produced where it is determined to be. After this type of experiment, each protein produced in Drospholia would be mapped to an organ or even specific cell giving great insight into what proteins do for a developing Drospholia.

Fluorescent experiments like those proposed by Gene Myers could gather a lot of data, but there are several other things to consider. *Cloning* would probably have to be done

to help assure that the expression level of the genes over time are close to the same in each experiment although there is no guarantee that this would work. The reason for this is that expression levels in cells are often *stochastic*. Even identical twins do not look exactly alike suggesting that some stochastic behavior does occur even with the exact same genetic data. Even with these limitations, fluorescent techniques are one of the most promising areas of data gathering.

A promising area of fluorescent experimentation is in *multiparameter flow cytometry.* Currently there are 12 colors and 2 scatter parameters that can be measured and used to study cellular behavior [46]. This is achieved by using 3 types of lasers to excite the 12 dyes used in this analysis. This method has been applied to human T-cells [96, 87]. One of the advantages of flow cytometry is that individual cells can be sorted by their fluorescence. Also, this sorting of cells does little damage to the cells, so that subsets of cells with similar expression patterns can be separated and further analyzed. However, as cells are sorted individually, connectivity involving many cells cannot be captured using this method. Multiparameter flow cytometry experiences similar problems with regular fluorescent experiments. The unavailability of more than just a few colors to monitor these cells poses a problem. The 12 or so colors used in multiparameter flow cytometry is nowhere near the approximately 25,000 genes estimated to be in the human genome [86].

2.2.3 Mass Spectrometry Experiments

Mass spectrometry is the identification of proteins by their weight (mass) and electrical charge [1]. It can be used to identify what proteins are in a cell. In order to perform mass spectrometry the proteins in a cell must be separated. *Gel electrophoresis* is one way in which proteins can be physically separated. In gel electrophoresis, proteins are separated by size and electrical charge by being moved though a gel by an electric current. As gel electrophoresis has limitations, there are several other ways in which proteins can be removed from cells and used in mass spectrometry [90].

Mass spectrometry has similar problems to microarray analysis. The proteins must be removed from the cell in order to be analyzed. This means that the cell is no longer able to grow and be tested further. There have also been suggestions that gel electrophoresis is unable to fully capture all proteins in a cell, or proteins of a small number [39]. This means that many cells must be studied at a time, limiting the length of a study by the

number of cells in the system.

2.2.4 Experiments from DNA Sequencing

As described earlier, current techniques allow for the DNA sequencing of entire genomes. Some learning approaches incorporate this DNA sequence information when learning genetic regulatory networks. Bussemaker *et al.* use an approach that detects upstream motifs, or recurring patterns of up to 7 bases found in the DNA sequence information for the genes in a system [19]. They then correlate this information with the gene expression levels to find the strongest matching motifs that bind to specific DNA portions. This finds the most active regulatory proteins in a system. Related motifs can then be grouped, much like clustering, except that they are grouped based on what protein regulates them and not just expression profiles.

2.2.5 Genome-wide Location Analysis Experiments

Genome-wide location analysis is the process of determining where along the DNA strand transcription factors bind. It is sometimes referred to as *chromatin immunoprecipitation* (ChIP) as it is one of the main processes. The entire process is sometimes referred to as *ChIP-to-chip* as microarrays are also used. Some of the steps involved in genome-wide location analysis are as follows: The transcription factors are attached to the DNA strand during normal gene activity. A substance is then injected into the cell to glue or affix these transcription factors to the DNA. The DNA strand is then cut or split around where the transcription factors are bound. Specific antibodies bind to the transcription factors so that they, along with the DNA to which they are bound, can be extracted from the rest of the DNA. The transcription factors are then unbound from the DNA strands and the DNA strands are then purified and amplified. With the binding sites separated from the rest of the DNA, microarrays are then used which have been specifically constructed to allow for the DNA to bind to specific locations to identify the specific part of the DNA to which the transcription factors bind. These data can be analyzed to determine the accuracy of the experimental results after which the binding sites can be identified. The process can start all over again with a different experimental setup or different goals in mind for the analysis like using different antibodies to identify different transcription factors. This process is explained in more detail in [92].

For more information about the molecular bindings and reactions that occur during transcription in eukaryotic cells, please see Lee and Young [67]. Garvie and Wolberger also

provide some insight into the physical process of how specific proteins bind to particular portions of DNA [29]. They describe how particular protein structures and folds are able to recognize and bind to specific portions of DNA.

There have been several applications of this method. Hall *et al.* use genome-wide location analysis on a yeast genome [40]. They discovered a metabolic enzyme that is attached to the DNA. They performed a mutation study and verified that the metabolic enzyme they discovered bound on the DNA did in fact regulate eukaryotic gene expression. Lee *et al.* describe how they use genome-wide location analysis to identify the transcriptional regulatory networks in *Saccharomyces cerevisiae* [66]. They describe several different types of network connections that they are able to identify. Some of these are autoregulation, regulator chains, multicomponent loops, and multi-input modules. Ren and Dynlacht describe how they use genome-wide location analysis on mammalian DNA [92]. They describe that synthesizing a microarray for the many extra bases that occur in mammalian DNA as compared to smaller organisms is not practical as of yet, for example, there are 3 million DNA fragments for the human genome. They, therefore, suggest that only placing the arrays that represent the gene promoters on the microarray would be both practical and still able to capture significant information.

There are several problems in genome-wide location analysis. Several of these are: finding ways in which to cement different transcription factors to the DNA strand, correctly cutting the DNA strand, proper amplification of the DNA strand, and microarray analysis. Robyr *et al.* suggest that some information is lost using generic antibodies to fix the transcription factors to the DNA segment and that very specific antibodies are sometimes needed to more fully cement the transcription factors to the DNA [94].

There are several approaches that combine genome-wide location analysis with other methods. Want *et al.* try to improve the results of this method with sequence analysis to potentially produce better results of where transcription factors bind, or even validate results from either approach [111]. Hartemink, and Bernard and Hartemink use this method in combination with DBNs to learn genetic networks which is described in more detail in Section 1.2.7 [41, 13]. Bar-Joseph *et al.* also use this method to construct genetic regulatory networks [9]. As genome-wide location analysis does not provide any type of regulatory information, they use expression data sets in order to determine activation or repression patterns. They use data from *Saccharomyces cerevisiae* which contains 106 transcription factors. Gao *et al.* also combine this method with expression data

[28]. Boulesteix and Strimmer use this method with gene expression data and by using a partial least squares approach infer the genetic network [16]. They claim that by using both of these together they are able to remove many of the false positives from the genome-wide location analysis and also allow activation and repression activities to be determined. They also claim that microarrays that measure regulatory transcription levels are generally inappropriate proxies for transcription factor activities and suggest a method to help infer the true transcription factor activities.

2.2.6 Wild Type Versus Mutational Experiments

The previous sections describe several different experiments that collect data from cells. However experiments can also be performed where the cells themselves are different. These differences are usually small and known so that differences of behavior can be traced to the differences between cells. These cells are often referred to as *wild type* cells and *mutated* cells.

Wild type experiments are experiments where the genetic regulatory network is allowed to function as it would in real life. For example, if the network of interest is the phage λ decision circuit and its effects on an *E. coli*, a normal virus and normal *E. coli* would be selected and studied. The environment would be carefully selected so as to not introduce anything that could potentially alter or change the way in which the normal genetic network functioned.

Although the wild type experiments are of particular interest, mutational experiments can provide significant insight into the genetic network. Zak *et al.* claim that mutational experiments are more favorable when trying to infer the genetic network [118]. Mutational experiments involve interfering with the genetic networks regular behavior to test specific parts of the network. These experiments usually involve single gene mutations instead of large amounts of mutations.

There are two main types of mutational experiments. One is performed by artificially increasing a gene's products in the cell. This can be performed by putting a population of cells into a medium rich with the products of a gene and allowing the cells to uptake that protein. Guet *et al.* performed something similar to this type of experiment in their designs. They grew the population of cells in mediums rich with ITPG, aTc, or both types of molecules artificially raising these protein levels to high concentrations in some of their experiments [37]. By increasing the amount of these proteins in the cells they studied, they are able to change the steady state of the experiment. It should be noted

that the genes that make the ITPG and aTc proteins are not included in their networks, so these proteins are either present in the medium or entirely absent in the experiments.

Another type of mutational experiment is done by removing or repressing genes so their products do not get produced, or by preventing them from performing their usual function. This can be done by physically removing the DNA sequence that encodes a protein from the DNA of the cell in question. It can also be done by injecting a different species that binds to the first, or applying an outside stimulus. In the phage λ decision circuit, for example, applying fluorescent light breaks the bonds of the CI dimer, thus preventing it from binding to the DNA strand and repressing the *cII* gene.

2.2.7 Steady State Versus Time Series Experiments

Many experiments that are performed are *steady state* experiments. Steady state experiments are experiments where a cell or culture of cells are allowed to develop for a specific amount of time. After time has elapsed, the cells are checked for the amount of proteins present. The cell protein levels are assumed to have settled into some steady state by the end of the experiment and are no longer significantly changing.

Guet *et al.* use steady state experiments to understand how their networks function in living cells. They create plasmids which are inserted into *E. coli* cells. These plasmids contain a gene that encodes for GFP. They then allow the cells to mature for a specified amount of time. After that time, they measure and record the amount of fluorescence coming from the culture of cells [37].

Steady state experiments only give limited information about what is happening in the cells. The reason for this is that they give the final state of the gene products and perhaps the initial state of the gene products if that information is explicitly known, but no intermediate information. Steady state information is helpful in providing correlational information using these two time points, but still provides little information. If the cells truly reach the same steady state each time, then additional steady state experiments would give no further information about the cell which would make doing more experiments useless as they provide no more insight into the inner workings of the cell. If the cells can end up in different steady states, then more steady state experiments can add useful information, but there would have to be many different ending states for the gene products to provide any statistical significance.

Time series experiments overcome some of the limitations of steady state experiments. As stated above, steady state experiments only give information on the genetic network

at the final time. Time series experiments are experiments where the gene expression information is studied as time progresses. This can provide information as to how the cell is able to move into the steady state, or even provide information as to how a cell that does not end in a steady state keeps changing. This adds crucial information about how the genetic regulatory system functions over time. It is for this reason that time series data are better for studying and learning the genetic connections between species.

2.3 Time Series Experiments

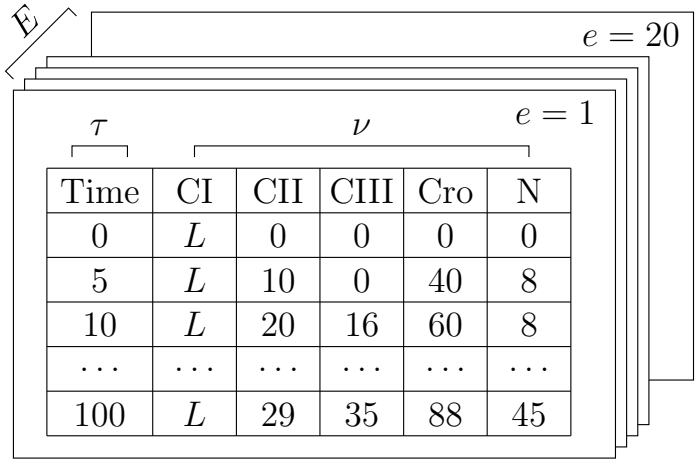
The goal of this work is to determine the influences between species (typically proteins) in a set S by examining a set of experiments, E, which include time series data over these species. Each data point in E, is a 3-tuple $\langle e, \tau, \nu \rangle$ where $e \in \mathbb{N}$ is a natural number representing the experiment number, $\tau \in \mathbb{R}$ is the time at which the species values were measured, and $\nu \in (\mathbb{R} \cup \{L\} \cup \{H\} \cup \{-\})^{|S|}$ is the state of each species $s \in S$. L and H are values that represent that a species is mutated low and high, respectively, in that data point. The symbol '$-$' represents an unknown value. The notation $\nu(s)$ denotes the value of species s for that data point. The notation $|E|$ is used to indicate the total number of data points within all of the experiments in E. The set of experimental data point successor pairs is defined formally as:

$$\begin{aligned} SUCC = \{(\langle e, \tau, \nu \rangle, \langle e', \tau', \nu' \rangle) \mid \\ \langle e, \tau, \nu \rangle \in E \wedge \langle e', \tau', \nu' \rangle \in E \wedge (e = e') \wedge (\tau < \tau') \wedge \\ \neg \exists \langle e, \tau'', \nu'' \rangle \in E . (\tau < \tau'') \wedge (\tau'' < \tau')\} \end{aligned}$$

Similarly, the set of experimental data point predecessor pairs is defined formally as:

$$\begin{aligned} PRED = \{(\langle e, \tau, \nu \rangle, \langle e', \tau', \nu' \rangle) \mid \\ \langle e, \tau, \nu \rangle \in E \wedge \langle e', \tau', \nu' \rangle \in E \wedge (e = e') \wedge (\tau' < \tau) \wedge \\ \neg \exists \langle e, \tau'', \nu'' \rangle \in E . (\tau' < \tau'') \wedge (\tau'' < \tau)\} \end{aligned}$$

Figure 2.3 shows an example set of time series data from the genetic network for the phage λ decision circuit that is typical of that collected using a microarray experiment in which an expression level for each species being monitored is reported. The set of species in this figure is $S = \{CI, CII, CIII, Cro, N\}$. There are 20 experiments, and each experiment has 21 data points (*i.e.,* $|E| = 420$). In general, each experiment can



Time	CI	CII	CIII	Cro	N
0	L	0	0	0	0
5	L	10	0	40	8
10	L	20	16	60	8
...	...	...	...	...	...
100	L	29	35	88	45

Figure 2.3. Example time series data for the phage λ decision circuit.

contain a different number of data points. An example data point is $\langle 1, 100, \nu \rangle$ where $\nu = \langle L, 29, 35, 88, 45 \rangle$. Note that L represents that species CI was mutated to a low state in that data point.

In addition, the time series data notation can also be used to represent experiments that should be performed by using the '$-$' symbol to represent data that should be measured. An example experiment to perform is shown in Figure 2.4. In this experiment, CI should be mutated to a high state and CII should be mutated to a low state. CIII and N should start at a 0 percent concentration and species Cro should have a 50 percent concentration. The experimenter should measure each species state every five minutes.

2.4 Discretizing Experimental Data

Our method discretizes the data from the time series experiments into a small number of *bins*, n. Our results indicate that an n of three or four bins per species performs the best. For simplicity, each species is discritized into the same number of bins, but in general the algorithms presented in this dissertation can be easily extended to allow different amounts of discritization per species, as is done in the implementation. The bins are defined by the *levels* that separate them. Formally, the levels, θ, for each species, $s \in S$, are $\langle \theta_0(s), \ldots, \theta_n(s) \rangle$ where $\theta_0(s)$ is 0, and $\theta_n(s)$ is ∞. These levels group the data into a small set of bins Φ, where $\Phi_j(s) = [\theta_j(s), \theta_{j+1}(s))$. The lowest bin for a species is $\Phi_0(s)$ and the highest bin is Φ_{n-1}. A *bin assignment*, $b \in \{0, .., n-1, *\}^{|S|}$, assigns each

$e = 21$

Time	CI	CII	CIII	Cro	N
0	H	L	0	50	0
5	H	L	–	–	–
10	H	L	–	–	–
...	...	...	...	...	...
100	H	L	–	–	–

Figure 2.4. Representation of the information needed to perform a time series data experiment.

$s \in S$ to a bin. The notation $b(s)$ indicates the bin assignment for species s in b. Note that a bin assignment of '$*$' for s indicates that there is no bin assignment to s. A bin assignment that includes $*$'s is called a *partial bin assignment*. The bin for a '$*$' is defined by $\Phi_*(s) = [0, \infty)$. The merger or union of two partial bin assignments, denoted $b \cup b'$, is accomplished by taking the bin assignment for each species and performing the union as follows:

$$b(s) \cup b'(s) = \begin{cases} b(s) & \text{if } b'(s) = * \\ b'(s) & \text{otherwise} \end{cases}$$

Note that this is only defined if $\forall s.(b(s) = b'(s) \vee b(s) = * \vee b'(s) = *)$. Also, $b(s)$++ increments the bin assignment of species s to the next highest bin. For example if $b(s) = 0$ then $b(s)$++= 1. Likewise $b(s)$−− decrements the bin assignment of species s to the next lowest bin. A set of bin assignments is denoted by B.

Discretizing the data can be done in many ways. The simplest approach is to divide the range of the data into equal size bins. Figure 2.5 shows a histogram representation of the values of species CIII in the phage λ decision circuit with values ranging from 0 to 100 where the dashed lines separate the data into three equal size bins, $[0, 33.3)$, $[33.3, 66.6)$, and $[66.6, 100)$.

To improve statistical significance, a better approach is to divide the data up into bins containing equal amounts of data. To calculate the amount of data that should be in each bin if the data are evenly divided, one divides the total number of data points by the number of bins. One can then place the levels such that each bin contains nearly the

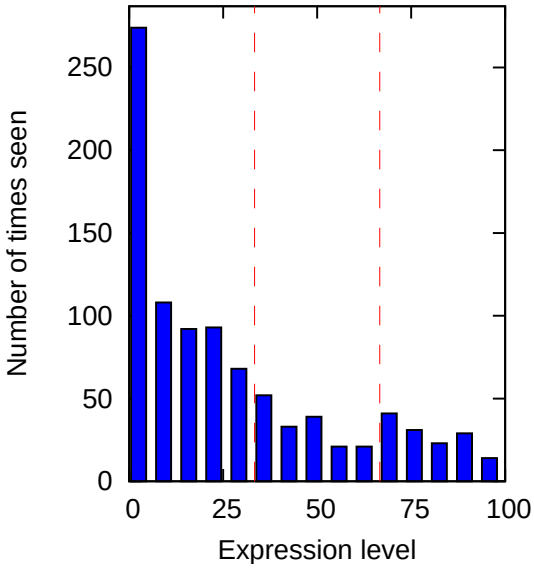


Figure 2.5. Time series data values and levels assigned for the CIII species at 33.3 and 66.6 percent. Notice that the data are broken into evenly spaced bins.

same amount of data. If there are 99 data points and 3 levels, ideally there would be 33 data points in each bin. In Figure 2.6, the dashed lines at 7 percent and 31 percent for the CIII species in the phage λ decision circuit divide the data so that each bin contains roughly the same number of data points.

There is, however, one complication with this approach. When a species spends most of its time at the same value, like 0 for unexpressed, it then becomes impossible to equally divide the state space so that each bin has equal data. Figure 2.7 shows the `DetermineLevels` function which divides the data so that each bin contains nearly equal amounts of data. This function first sorts the data by value and puts the lowest and highest levels to 0 and ∞, respectively. Next, it determines how many data points should optimally be in the next bin. It then walks the sorted data until it finds that many data points and assigns a bin to the next highest value. It then redivides the remaining data and starts again. For example, if there are 100 data points, and 50 of them are at 0, then there should be a bin for 0 which includes all 50 data points, and the remaining two bins should optimally have 25 data points in them.

A final approach for selecting levels is to allow the user to input the levels manually. A user may know at what level the species should activate or repress each other. This

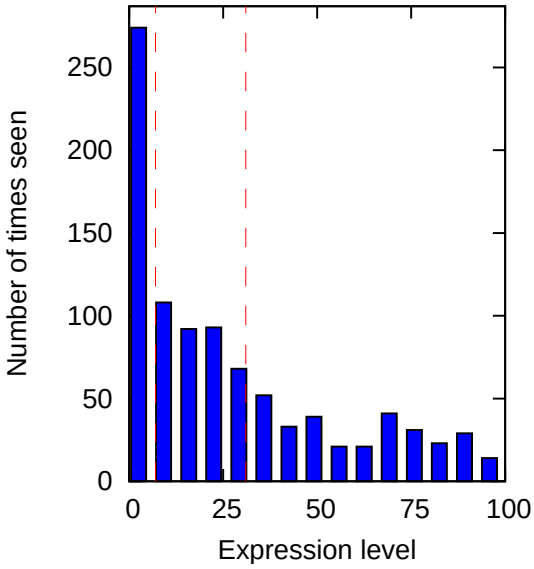


Figure 2.6. Time series data values and levels assigned for the CIII species at 7 and 31 percent. Notice that each bin contains approximately the same amount of data.

allows added flexibility to our method.

Once the levels are selected, individual time series data points can be assigned to bins using the `BinAssign` function shown in Figure 2.8. The `BinAssign` function takes a time series data point, the set of species, the levels for the species and the total number of bins. It selects a species and sets the bin for that species to 0 if the species is mutated low in that data point, or sets the bin to $n-1$ if the species is mutated high, or sets the bin to the middle bin if the value is unknown. If the species value did not match one of these cases, the bin is determined by first testing if the expression value for that species is below level one. If the species' expression value is below that level, it falls into bin 0. If the expression value is greater than this value, the `BinAssign` function uses the next highest level, until it finds the correct bin for that species. It does the same thing for each species in S. It then returns the created bin assignment.

Table 2.1 shows example time series data for the phage λ decision network and their bin assignments. The levels used to bin the data are: $\theta(CI) = \langle 0, 32, 72, \infty \rangle$, $\theta(CII) = \langle 0, 1, 14, \infty \rangle$, $\theta(CIII) = \langle 0, 1, 16, \infty \rangle$, $\theta(Cro) = \langle 0, 52, 71, \infty \rangle$, and $\theta(N) = \langle 0, 16, 27, \infty \rangle$.

```
DetermineLevels(Species Set S, Experiments E, Number of levels n)
  foreach s ∈ S
    stack ES := sort(E, s)
    θ_j(s) := 0
    θ_n(s) := ∞
    for j := 1 to n − 1
      θ_j(s) := ∞
      for k := 0 to |ES|/(n−j+1)
        ⟨e, τ, ν⟩ := pop(ES)
      do
        ⟨e′, τ′, ν′⟩ := pop(ES)
      while (ν(s) = ν′(s) ∧ |ES| > 0)
      if (|ES| ≠ 0) then
        θ_j(s) := ν′(s)
        push(ES, ⟨e′, τ′, ν′⟩)
  return θ
```

Figure 2.7. The `DetermineLevels` function.

```
BinAssign(Data ν, Species Set S, Levels θ, Number of Bins n)
  foreach s ∈ S
    if ν(s) = L then
      b(s) = 0
    else if ν(s) = H then
      b(s) = n − 1
    else if ν(s) = − then
      b(s) = (n−1)/2
    else
      for i := 1 to n − 1
        if ν(s) < θ_i(s) then
          b(s) := i − 1
          break
  return b
```

Figure 2.8. The `BinAssign` function.

Table 2.1. Example time series data and bin assignments for the phage λ decision circuit.

		Data Values					Bin Assignments				
Experiment	Time	CI	CII	CIII	Cro	N	CI	CII	CIII	Cro	N
1	0	0	0	0	0	0	0	0	0	0	0
	20	0	0	9	14	0	0	0	1	0	0
	40	0	12	7	41	23	0	1	1	0	1
	60	2	16	26	47	33	0	2	2	0	2
	80	11	27	37	61	43	0	2	2	1	2
	100	17	25	35	67	43	0	2	2	1	2
	120	20	36	34	66	46	0	2	2	1	2
	140	23	35	34	71	44	0	2	2	1	2
	160	23	28	33	78	42	0	2	2	2	2
	180	29	25	33	85	40	0	2	2	2	2
	200	32	21	31	84	39	0	2	2	2	2
2	0	0	0	0	0	0	0	0	0	0	0
	20	0	15	10	28	16	0	2	1	0	0
	40	5	24	29	48	15	0	2	2	0	0
	60	11	28	26	62	20	0	2	2	1	1
	80	14	43	23	61	24	0	2	2	1	1
	100	26	39	20	75	30	0	2	2	2	2
	120	32	28	16	71	42	0	2	1	1	2
	140	41	24	13	71	40	1	2	1	1	2
	160	41	16	11	70	37	1	2	1	1	2
	180	47	12	11	76	36	1	1	1	2	2
	200	50	7	9	83	33	1	1	1	2	2
3	0	0	0	0	0	0	0	0	0	0	0
	20	0	6	13	14	17	0	1	1	0	1
	40	0	9	17	35	44	0	1	2	0	2
	60	5	23	24	47	50	0	2	2	0	2
	80	5	42	29	61	66	0	2	2	1	2
	100	17	48	27	67	65	0	2	2	1	2
	120	23	50	23	66	61	0	2	2	1	2
	140	26	48	23	66	57	0	2	2	1	2
	160	29	40	21	65	54	0	2	2	1	2
	180	29	43	19	64	51	0	2	2	1	2
	200	38	37	17	64	46	1	2	2	1	2

2.5 Compressing Data

Although discretizing the data reduces the range of the data, the algorithms presented in this dissertation examine this data often, so a more compact and efficient representation is needed. In particular, data points that map to the same bin assignment can be combined. For each combined bin assignment, our method records both the number of data points that map to it as well as the number of times that each species expression value increases in the next time point or has increased from the previous time point. This *compressed data* (CD) structure is defined formally as $CD : S \rightarrow (\{0, .., n-1\}^{|S|} \rightarrow \{\mathbb{N} \times \mathbb{N}\})$.

The CD is computed using the `Compress` function shown in Figure 2.9. The `Compress` function maps a species $s \in S$ and a bin assignment to a tuple representing the number of times this bin assignment is seen in the data and the number of times species s increases in the next time point, or increased from the previous time point. The notation CD_s is shorthand for $CD(s)$ and refers to the mapping for species s. The notation $CD_s(b) = \langle \mathbb{N} \times \mathbb{N} \rangle$ refers to the mapping of a species and bin assignment to the tuple described above. The `Compress` function works as follows. First, for every experimental time point and species, s, the CD_s is initialized. Note that the function does not allocate all possible bin assignments, but only records those bin assignments that appear in the data, as in general the state space is sparse. This limits the size of the CD to be in the worst case the same size as the experimental data itself if, in the unlikely event that every row in the experimental data has a different bin assignment. Then, for every element of $SUCC \cup PRED$ and species, s, it checks that the data entry for s is not mutated or empty. It then checks if the species value is increasing between time points, and then the seen and increasing values are incremented for the associated bin assignment. If the species expression value is not increasing, then only the seen value is incremented.

Table 2.2 shows the compressed data of Table 2.1. Time 0 in experiment 1 contains each species in the phage λ decision circuit at its lowest bin assignment. Time 0 in experiment 2 and 3 also have each species at its lowest bin assignment, so these data values are compressed giving a total of 3 times that this state occurs in the data. By combining rows with the same discretized values, the size of the table is reduced from 33 entries to 14. By organizing the data in this way, finding data for a particular bin assignment becomes easier.

```
Compress(Experiments E, Species Set S, Levels θ)
  foreach ⟨e, τ, ν⟩ ∈ E
    foreach s ∈ S
      CD_s(BinAssign(ν, S, θ, n)) := ⟨0, 0⟩
  foreach (⟨e, τ, ν⟩, ⟨e', τ', ν'⟩) ∈ SUCC ∪ PRED
    if (ν(s) ∉ {L, H, −} ∧ ν' ∉ {L, H, −}) then
      foreach s ∈ S
        if ((τ < τ' ∧ ν(s) < ν'(s)) ∨ (τ' < τ ∧ ν'(s) < ν(s))) then
          CD_s(BinAssign(ν, S, θ, n)) := CD_s(BinAssign(ν, S, θ, n)) + ⟨1, 1⟩
        else
          CD_s(BinAssign(ν, S, θ, n)) := CD_s(BinAssign(ν, S, θ, n)) + ⟨0, 1⟩
  return CD
```

Figure 2.9. The `Compress` function.

2.6 Projecting Data

The algorithms in the next chapter often use partial bin assignments. In this case, it is useful to project the compressed data onto the species $S' \subseteq S$ where S' is the set of species with specified values in the partial bin assignment. The *projected compressed data* (PCD) structure is defined formally as $PCD_s : (\{0, .., n-1, *\}^{|S'|} \rightarrow \{\mathbb{N} \times \mathbb{N}\})$.

The PCD_s is computed using the `Project` function shown in Figure 2.10. The `Project` function maps a bin assignment consisting of a subset of the species, $S' \subseteq S$, to the increasing and occurrence information for a particular species s. The first thing the function does is to look through the domain of the CD_s and initialize each of these entries in the function with zeros. It uses the `PartialBA` function, shown in Figure 2.11, to remove unwanted species from the new bin assignment. The `PartialBA` function creates a new bin assignment with only the species that occur in the species set S'. The `Project` function then looks back through the data and calculates the PCD_s by summing all the entries that correspond to this partial bin assignment for species s.

Table 2.3 shows the projection of Table 2.2 with $c = CI$, and $S' = \{CI, CII\}$. Notice that there are only 5 rows in the table compared to the 14 from Table 2.2, and the 33 from Table 2.1.

Table 2.2. Compressed time series data for the phage λ decision circuit. Notice that by merging states there are now 14 rows compared to the original 33 shown in Table 2.1. This effect may become more pronounced with more data.

$b = \langle CI, CII, CIII, Cro, N\rangle$	$CD_{CI}(b)$	$CD_{CII}(b)$	$CD_{CIII}(b)$	$CD_{Cro}(b)$	$CD_N(b)$
$\langle 0,0,0,0,0\rangle$	$\langle 0,3\rangle$	$\langle 2,3\rangle$	$\langle 3,3\rangle$	$\langle 3,3\rangle$	$\langle 2,3\rangle$
$\langle 0,0,1,0,0\rangle$	$\langle 0,1\rangle$	$\langle 1,1\rangle$	$\langle 0,1\rangle$	$\langle 1,1\rangle$	$\langle 1,1\rangle$
$\langle 0,1,1,0,1\rangle$	$\langle 1,2\rangle$	$\langle 2,2\rangle$	$\langle 2,2\rangle$	$\langle 2,2\rangle$	$\langle 2,2\rangle$
$\langle 0,1,2,0,2\rangle$	$\langle 1,1\rangle$	$\langle 1,1\rangle$	$\langle 1,1\rangle$	$\langle 1,1\rangle$	$\langle 1,1\rangle$
$\langle 0,2,1,0,0\rangle$	$\langle 1,1\rangle$	$\langle 1,1\rangle$	$\langle 1,1\rangle$	$\langle 1,1\rangle$	$\langle 0,1\rangle$
$\langle 0,2,1,1,2\rangle$	$\langle 1,1\rangle$	$\langle 0,1\rangle$	$\langle 0,1\rangle$	$\langle 0,1\rangle$	$\langle 0,1\rangle$
$\langle 0,2,2,0,0\rangle$	$\langle 1,1\rangle$	$\langle 1,1\rangle$	$\langle 0,1\rangle$	$\langle 1,1\rangle$	$\langle 1,1\rangle$
$\langle 0,2,2,0,2\rangle$	$\langle 1,2\rangle$	$\langle 2,2\rangle$	$\langle 2,2\rangle$	$\langle 2,2\rangle$	$\langle 2,2\rangle$
$\langle 0,2,2,1,1\rangle$	$\langle 2,2\rangle$	$\langle 1,2\rangle$	$\langle 0,2\rangle$	$\langle 1,2\rangle$	$\langle 2,2\rangle$
$\langle 0,2,2,1,2\rangle$	$\langle 8,10\rangle$	$\langle 4,10\rangle$	$\langle 0,10\rangle$	$\langle 4,10\rangle$	$\langle 1,10\rangle$
$\langle 0,2,2,2,2\rangle$	$\langle 3,4\rangle$	$\langle 0,4\rangle$	$\langle 0,4\rangle$	$\langle 1,4\rangle$	$\langle 1,4\rangle$
$\langle 1,1,1,2,2\rangle$	$\langle 1,2\rangle$	$\langle 0,2\rangle$	$\langle 0,2\rangle$	$\langle 1,2\rangle$	$\langle 0,2\rangle$
$\langle 1,2,1,1,2\rangle$	$\langle 1,2\rangle$	$\langle 0,2\rangle$	$\langle 0,2\rangle$	$\langle 1,2\rangle$	$\langle 0,2\rangle$
$\langle 1,2,2,1,2\rangle$	$\langle 0,1\rangle$	$\langle 0,1\rangle$	$\langle 0,1\rangle$	$\langle 0,1\rangle$	$\langle 0,1\rangle$

```
Project(Compressed Data CD, Species s, Species Sets S, S')
  foreach b ∈ domain(CD_s)
    PCD_s(PartialBA(b, S, S')) := ⟨0, 0⟩
  foreach b ∈ domain(CD_s)
    PCD_s(PartialBA(b, S, S')) := PCD(PartialBA(b, S, S')) + CD_s(b)
  return PCD_s
```

Figure 2.10. The `Project` function.

```
PartialBA(Bin Assignment b, Species Sets S, S')
  foreach s ∈ S
    if s ∈ S' then
      b'(s) := b(s)
    else
      b'(s) := *
  return b'
```

Figure 2.11. The `PartialBA` function.

Table 2.3. Projected time series data for the phage λ decision circuit using the `Project` function on the data from Table 2.2, the species $s = CI$ and the species set $S' = \{CI, CII\}$.

$b = \langle CI, CII, CIII, Cro, N \rangle$	$PCD_{CI}(b)$
$\langle 0, 0, *, *, * \rangle$	$\langle 0, 4 \rangle$
$\langle 0, 1, *, *, * \rangle$	$\langle 2, 3 \rangle$
$\langle 0, 2, *, *, * \rangle$	$\langle 17, 21 \rangle$
$\langle 1, 1, *, *, * \rangle$	$\langle 1, 2 \rangle$
$\langle 1, 2, *, *, * \rangle$	$\langle 1, 3 \rangle$

CHAPTER 3

INFLUENCE VECTORS

It's not the money. It's not the fame. It's the influence.
–Clay Aiken

This chapter describes how our method represents the influences or connections between species in a genetic network. It also describes a function that determines if this representation is supported by the time series experimental data. This scoring function is the core of the algorithms presented in the next chapter. It then describes an optimized version of this function using the data structures described in Chapter 2. This chapter also describes how the function deals with sparse data.

3.1 Formal Representation

Our method represents the connections between species in a genetic network using *influence vectors*. An influence vector, i, is a vector over the species, S, that describes the types of influences that each individual species, $s \in S$, has on a particular species, c. The possible types of influence between species are activation 'a', repression 'r', no influence 'n', or unknown influence '?'. An influence is directed from a *parent* to a *child* in that the presence of a parent species has the described influence over the production of a child species. In a genetic network, it is possible (and likely) that multiple parent species may have an influence on a child species. Therefore, the influences between species is represented using a function that returns a vector over the set of species that indicates the type of influence that each species has over a given child species (*i.e.,* $i : S \rightarrow \{a, r, n, ?\}$). A genetic network is represented using a collection of influence vectors, $\mathcal{I}$ (*i.e.,* $\mathcal{I} : S \rightarrow (S \rightarrow \{a, r, n, ?\})$). The notation $\mathcal{I}(c)$ returns an influence vector, i, which indicates the parent species for child species c. The notation $i(s)$ returns the influence that parent s has in influence vector i. The function, $Act(i)$, returns the activating species in i, (*i.e.,* those species where $i(s)$ returns 'a'). The function, $Rep(i)$,

returns the repressing species in i, (*i.e.,* those species where $i(s)$ returns 'r'). The function $Par(i)$ returns the parents in i, (*i.e.,* $Act(i) \cup Rep(i)$). The size of in influence vector, i, is indicated by, $|i|$, and it is shorthand for $|Par(i)|$. The merger or union of two influences vectors, denoted $i \cup i'$, is constructed for each species s:

$$(i \cup i')(s) = i(s) \cup i'(s)$$

where the union of individual influences is defined in Table 3.1.

Consider a possible influence vector for the species CIII in the 5 species phage λ decision circuit of $i = \langle r, ?, n, r, a \rangle$ where the species ordering is $\langle CI, CII, CIII, Cro, N \rangle$. This influence vector indicates that species CI and Cro repress species CIII's expression, species CII has an unknown influence on CIII, species CIII has no influence on itself, and species N activates CIII's expression. This is expressed formally as:

$$\begin{aligned} Act(i) &= \{\text{N}\} \\ Rep(i) &= \{\text{CI}, \text{Cro}\} \\ Par(i) &= \{\text{CI}, \text{Cro}, \text{N}\} \end{aligned}$$

3.2 Graphical Representation

It is often convenient to represent influence vectors in a graphical form. The species in an influence vector can be thought of as nodes, and the influences between species as edges in a graphical structure. Since influences are directed from a parent to a child species, the graph is a directed graph. There are 4 types of edges in an influence vector, which means that there need to be 4 types of arcs or edges in the graph.

Edges in the graph come from the connections between parent species, p, to child species, c. When converting from an influence vector, $i = \mathcal{I}(c)$, to a directed graph

Table 3.1. The $\cup$ operator definition for two influence vectors. Corresponding influences are merged using this table. Note that an 'a' influence is never merged with an 'r' influence when using the algorithm presented in this dissertation.

$\cup$	a	r	n	?
a	a	$\emptyset$	a	a
r	$\emptyset$	r	r	r
n	a	r	n	n
?	a	r	n	?

representation, there is an activation edge, shown with the '$\rightarrow$' symbol, in the graph from p to c (*i.e.,* $p \rightarrow c$) if $i(p)$ returns 'a'. There is a repression edge, shown with the '$\dashv$' symbol, from p to c (*i.e.,* $p \dashv c$) if $i(p)$ returns 'r'. If the connection is unknown (*i.e.,* $i(s) =$'?'), the symbol, '?', appears as a label on the edge. If there is no connection (*i.e.,* $i(s) =$'n'), there is no edge in the graph.

For a given network of interest, biologists may have discovered some of the influences between species. This could have been by performing very precise experiments in a laboratory setting, for example. This information is called the *biological background knowledge*. This knowledge, however, is likely very incomplete, meaning that very few entries in the influence vectors can be updated from a '?' to either an 'a', 'r', or 'n'. Figure 3.1 shows the phage λ decision circuit where some influences for species CIII have been updated.

When first constructing an influence vector for the influences between the species in the network being studied, often no information is assumed to be known. This means that the all values in each influence vector, $i \in \mathcal{I}$, are set to '?'. Figure 3.2 shows the influence vectors representing no known information for CIII in the phage λ decision circuit. Note that this figure represents that species CIII does not influence itself. The algorithms presented in the next chapter currently do not learn self-regulatory effects, but we plan on extending these methods to look for these types of influences.

Figure 3.3 shows the directed graph representation of the phage λ decision circuit that has been discovered by biological experimentation for each of the species in the phage λ

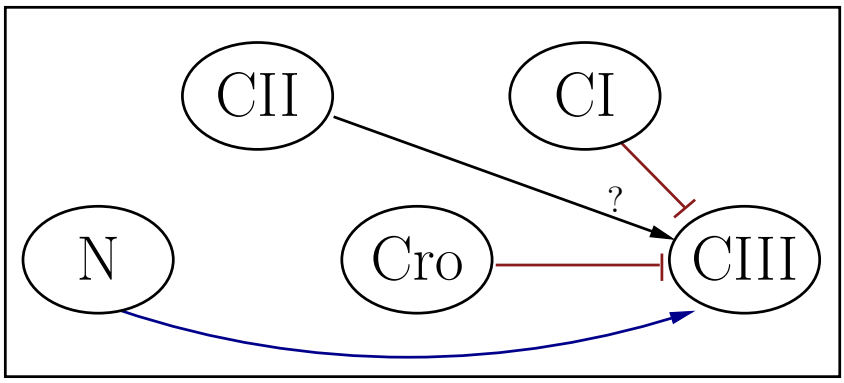


Figure 3.1. Graphical representation of the influence vector for species CIII in the phage λ decision circuit with some biological background knowledge added.

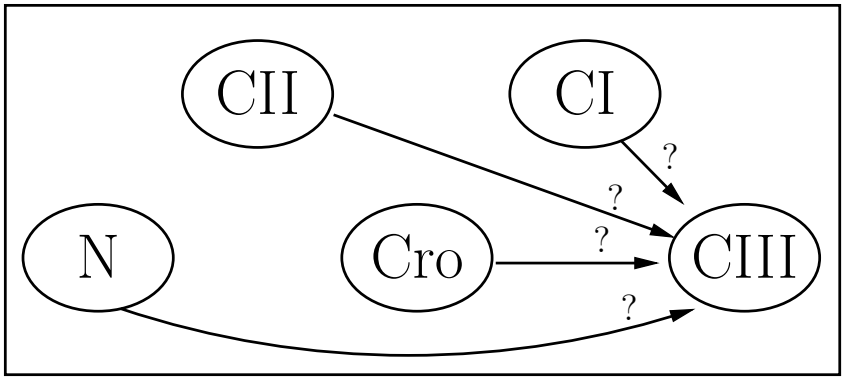


Figure 3.2. Graphical representation of the influence vector for species CIII in the phage λ decision circuit. This representation indicates that no information about any connections in the network is known. It is the typical starting point for learning methods.

decision circuit [89]. This network is one of the 8.47×10^{11} possible networks composed of five species under our setup. This is excluding connections representing unknown or '?' influences. For example, if there are 5 species (*i.e.*, $|S| = 5$) in the genetic network, as there are in the phage λ decision circuit, then each species can be influenced by each of the 5 species. Each of these influence can be either an 'a', 'r', or 'n' connection. This gives $3^{|5|}$ different influences per species, and $3^{|5|^2}$ when these are combined with the influences for all the species. In general, the maximum number of potential networks with no '?' influences are $3^{|S|^2}$, since there are $3^{|S|}$ values for the influence vector for each species, and there are $|S|$ species.

3.3 Scoring: Theory

Given the large number of potential networks, it is crucial to be able to reduce the number of networks considered, which is the subject of Chapter 4. It is also necessary to be able to assign a score to each influence vector that makes up the potential networks being considered, which is the subject of the remainder of this chapter. This section presents the theory on how our method scores potential influence vectors while the remaining sections present the algorithms for doing so efficiently. As our method looks for differences in expression between two time points it constructs a first order Markov model.

A score for an influence vector, i, is determined by calculating probabilities for a species, s, increasing in expression both into and out of valid data points that are in a

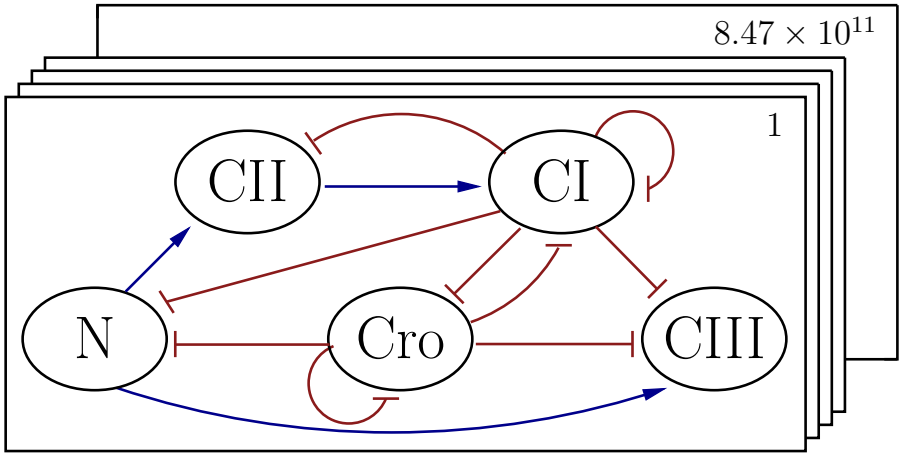


Figure 3.3. Graphical representation of the phage λ decision circuit network, as well as all other possible 5 species networks.

partial bin assignment, b. These probabilities are of the form:

$$\mathcal{P}(incr(s) \mid valid(s) \cap bin(b)) = \frac{\mathcal{P}(incr(s) \cap valid(s) \cap bin(b))}{\mathcal{P}(valid(s) \cap bin(b))} \tag{3.1}$$

The data point pairs that have valid data, meaning no mutations or unknown values, for species, s, are determined as follows:

$$\begin{aligned} valid(s) = \{(\langle e, \tau, \nu\rangle, \langle e', \tau', \nu'\rangle) \in SUCC \cup PRED \mid \\ \nu(s) \notin \{L, H, -\} \wedge \nu'(s) \notin \{L, H, -\}\} \end{aligned} \tag{3.2}$$

The $bin(b)$ function returns those data point pairs in the set $SUCC \cup PRED$ where the data values of the first data point in the pair fall within the bins specified in the partial bin assignment, b. This is expressed formally as:

$$bin(b) = \{(\langle e, \tau, \nu\rangle, \langle e', \tau', \nu'\rangle) \in SUCC \cup PRED \mid \forall s' \in S.\nu(s') \in \Phi_{b(s')}(s')\} \tag{3.3}$$

The probability of a data point pair having valid data for the species of interest, and being in a partial bin assignment, b, is determined as follows:

$$\mathcal{P}(valid(s) \cap bin(b)) = \frac{|valid(s) \cap bin(b)|}{|SUCC \cup PRED|} \tag{3.4}$$

The $incr(s)$ function returns the data point pairs in $SUCC$ where the expression level of species s increases between the first and second data points in the pair and those data

point pairs in $PRED$ where the expression level of species s increases between the second and first data point in the pair. This is expressed formally as:

$$\begin{aligned} incr(s) = \{(\langle e, \tau, \nu\rangle, \langle e', \tau', \nu'\rangle) \in SUCC \mid \nu(s) < \nu'(s)\} \\ \cup \{(\langle e, \tau, \nu\rangle, \langle e', \tau', \nu'\rangle) \in PRED \mid \nu'(s) < \nu(s)\} \end{aligned} \tag{3.5}$$

The probability of a data point pair increasing, having valid data for the species of interest, s, and being in a partial bin assignment, b, is determined as follows:

$$\mathcal{P}(incr(s) \cap valid(s) \cap bin(b)) = \frac{|incr(s) \cap valid(s) \cap bin(b)|}{|SUCC \cup PRED|} \tag{3.6}$$

Using Equations 3.1, 3.4 and 3.6, the probability of species, s, increasing in expression given that it is a valid data points and in a bin assignment, b, is determined as follows:

$$\mathcal{P}(incr(s) \mid valid(s) \cap bin(b)) = \frac{|incr(s) \cap valid(s) \cap bin(b)|}{|valid(s) \cap bin(b)|} \tag{3.7}$$

In order to score an influence vector, i, a probability of the form of Equation 3.7 is determined for all possible partial bin assignments using each $s' \in Par(i)$. Consider the influence vector $i = \langle n, r, r, n, n\rangle$ for the phage λ decision circuit where the order of the species is $\langle CI, CII, CIII, Cro, N\rangle$. Note that $Par(i) = \{CII, CIII\}$. Partial bin assignments are created with a bin assignment for these two species and no bin assignment for the other species. All partial bin assignments, and their associated probabilities, for this influence vector are shown in Table 3.2, or graphically in Figure 3.4.

Table 3.2. Bin assignments, probabilities, ratios, and votes for the influence vector $\langle n, r, r, n, n\rangle$ where the species order is $\langle CI, CII, CIII, Cro, N\rangle$ and the species of interest is $s = N$.

$\langle CI, CII, CIII, Cro, N\rangle$	Probability	Ratio	Vote
$\langle *, 0, 0, *, *\rangle$	40		
$\langle *, 0, 1, *, *\rangle$	49	1.23	$votes_a$
$\langle *, 0, 2, *, *\rangle$	70	1.75	$votes_a$
$\langle *, 1, 0, *, *\rangle$	58	1.45	$votes_a$
$\langle *, 1, 1, *, *\rangle$	42	1.05	$votes_n$
$\langle *, 1, 2, *, *\rangle$	38	0.95	$votes_n$
$\langle *, 2, 0, *, *\rangle$	66	1.65	$votes_a$
$\langle *, 2, 1, *, *\rangle$	38	0.95	$votes_n$
$\langle *, 2, 2, *, *\rangle$	26	0.65	$votes_f$

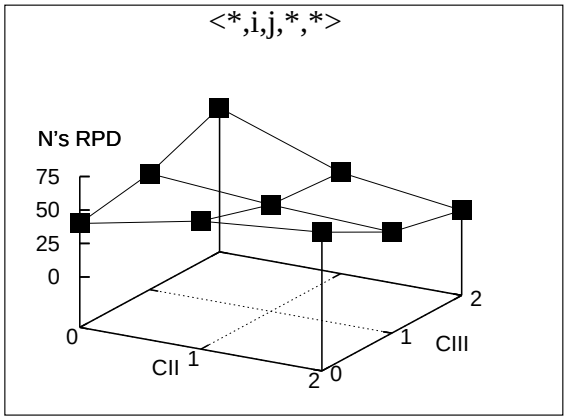


Figure 3.4. N's probability of increasing with the influence vector $i = \langle n, r, r, n, n \rangle$, where the species order is $\langle CI, CII, CIII, Cro, N \rangle$. The probabilities used are listed in Table 3.2.

To determine trends in the data, a ratio is formed of two probabilities of the form of Equation 3.7 using two partial bin assignments, b and b'. The partial bin assignment, b', used for comparison against partial bin assignment, b, is called the *base*. The base is a partial bin assignment with the lowest bin assignment for the species in the $Act(i)$ set and the highest bin assignment for the species in the $Rep(i)$ set, unless there are more repressors in the influence vector in which case the roles are reversed. For example, for the influence vector $i = \langle a, r, n, n, n \rangle$, the base bin assignment would be $b' = \langle 0, 2, *, *, * \rangle$ assuming $n = 3$. For the influence vector $i = \langle n, r, r, n, n \rangle$, the base would be $b' = \langle *, 0, 0, *, * \rangle$. The base, b', is constructed as follows:

$$b'(s) = \begin{cases} * & \text{if } i(s) = \text{`}n\text{'} \\ 0 & \text{if } (i(s) = \text{`}a\text{'} \wedge |Rep(i)| \leq |Act(i)|) \vee (i(s) = \text{`}r\text{'} \wedge |Rep(i)| > |Act(i)|) \\ n-1 & \text{otherwise} \end{cases}$$

To evaluate an influence vector, i, a probability ratio is determined using the probability for the base, b', and the probability calculated for each other partial bin assignment, b as follows:

$$\frac{\mathcal{P}(incr(s) \mid valid(s) \cap bin(b))}{\mathcal{P}(incr(s) \mid valid(s) \cap bin(b'))} = \frac{|incr(s) \cap valid(s) \cap bin(b')|}{|valid(s) \cap bin(b')|} * \frac{|valid(s) \cap bin(b)|}{|incr(s) \cap valid(s) \cap bin(b)|} \tag{3.8}$$

This ratio represents the change in expression between a partial bin assignment and its' base which expresses a general trend in the data. Table 3.2 shows the ratios created for the example influence vector.

The development of the base and probability ratios is such that a ratio greater than one indicates that the species has more expression over the base, and a ratio less than one indicates that the species has less expression over the base. If there are more activating influences in the influence vector (*i.e.,* $|Rep(i)| \leq |Act(i)|$), then a ratio larger than one indicates support for the influence vector and a ratio less than one does not support the influence vector. If there are more repressing influences in the influence vector (*i.e.,* $|Rep(i)| > |Act(i)|$), then a ratio less than one indicates support for the influence vector, and a ratio greater than one indicates that the data do not support the influence vector. Each ratio is used to cast a vote for or against an influence vector. Since ratios near one are indeterminate, our method also allows a ratio to yield a neutral vote. Figure 3.5(a) shows the thresholds used for votes when there are more activating species, and Figure 3.5(b) shows thresholds used for votes when there are more repressing influences in the influence vector. The votes cast for the example influence vector are shown in Table 3.2.

The final score is determined by using the following equation:

$$score = \frac{votes_f - votes_a}{votes_f + votes_a + votes_n}$$

Note that a score greater than zero indicates support for the influence vector being scored which a negative score indicates there is no support for the influence vector. For our

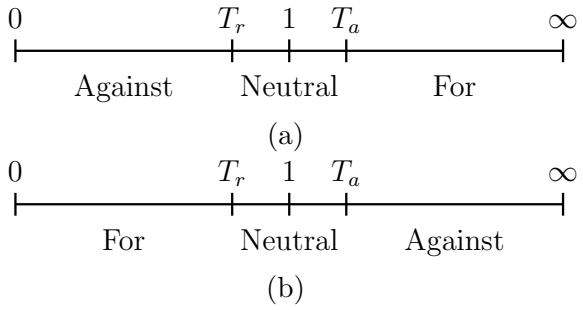


Figure 3.5. (a) Thresholds used to cast a vote when there are more activating influences in the influence vector (*i.e.,* $|Rep(i)| \leq |Act(i)|$). (b) Thresholds used to cast a vote where there are more repressing influences in the influence vector (*i.e.,* $|Rep(i)| > |Act(i)|$).

example, the resulting score would be -0.25 which indicates that this is not a correct influence vector.

When scoring an influence vector, i, for species s, the probability of increase can be influenced by the level of s. For example, it is often the case that when s is at a high concentration that its rate of increase reduces as degradation begins to dominate. Therefore, considering each level of s separately can give a better picture of how other species influence its rate of increase. Similarly, when comparing two influence vectors, i and i', it is useful to control for the species in i', when evaluating the score for i and vice versa. In both of these cases, our method partitions the set of bins using the species in a control set, G. When controlling for G while evaluating an influence vector, i, the partial bin assignments considered now include those with assignments to all species in $Par(i) \cup G$. Breaking the data up in this way can be thought of as being similar to performing a mutational experiment on the data where our method fixes those species in G to specific levels.

This is illustrated with an example from the phage λ decision circuit where the child species s is N, the influence vector i is $\langle n, r, r, n, n\rangle$, representing that both CII and CIII repress N, and the set of species $G = \{N\}$, with each species having a total of 3 bins. The partial bin assignments created for this configuration, along with their probabilities, are shown in Table 3.3. Figure 3.6 represents the probabilities graphically.

The base is also calculated in a slightly different way when controlling for the species in G. The base bin assignment, b', agrees with the values in b for each of the members in the G set. This is defined formally as:

$$b'(s) = \begin{cases} b(s) & \text{if } s \in G \\ * & \text{if } i(s) = \text{`}n\text{'} \\ 0 & \text{if } (i(s) = \text{`}a\text{'} \wedge |Rep(i)| \leq |Act(i)|) \\ & \quad \vee (i(s) = \text{`}r\text{'} \wedge |Rep(i)| > |Act(i)|) \\ n-1 & \text{otherwise} \end{cases} \tag{3.9}$$

Note that the base determination is now dependent on the partial bin assignment, b, that it is being ratioed against. In Table 3.3, for example, the base influence vectors are $\langle *, 0, 0, *, 0\rangle, \langle *, 0, 0, *, 1\rangle, \langle *, 0, 0, *, 2\rangle$. Now, each probability ratio calculated must use the proper base. In Table 3.3, the base used for a bin assignment, b, is the one that occurs the closest above b, as the table is organized by the bin assignments to the species in G. One ratio is determined for each of the probabilities in Table 3.3 excluding the bases. The ratios, and their associated votes, are shown in Table 3.3 for the example

Table 3.3. Bin assignments, probabilities, ratios, and votes for the influence vector $\langle n, r, r, n, n\rangle$ where the species order is $\langle CI, CII, CIII, Cro, N\rangle$ and $G = \langle N\rangle$ and the species of interest is $s = N$.

$\langle CI, CII, CIII, Cro, N\rangle$	Probability	Ratio	Vote
$\langle *, 0, 0, *, 0\rangle$	40	base	
$\langle *, 0, 1, *, 0\rangle$	58	1.45	$votes_a$
$\langle *, 0, 2, *, 0\rangle$	83	2.08	$votes_a$
$\langle *, 1, 0, *, 0\rangle$	67	1.66	$votes_a$
$\langle *, 1, 1, *, 0\rangle$	55	1.37	$votes_a$
$\langle *, 1, 2, *, 0\rangle$	59	1.47	$votes_a$
$\langle *, 2, 0, *, 0\rangle$	100	2.50	$votes_a$
$\langle *, 2, 1, *, 0\rangle$	44	1.09	$votes_n$
$\langle *, 2, 2, *, 0\rangle$	36	0.90	$votes_n$
$\langle *, 0, 0, *, 1\rangle$	55	base	
$\langle *, 0, 1, *, 1\rangle$	40	0.72	$votes_f$
$\langle *, 0, 2, *, 1\rangle$	50	0.90	$votes_n$
$\langle *, 1, 0, *, 1\rangle$	54	0.98	$votes_n$
$\langle *, 1, 1, *, 1\rangle$	37	0.67	$votes_f$
$\langle *, 1, 2, *, 1\rangle$	41	0.75	$votes_n$
$\langle *, 2, 0, *, 1\rangle$	0	0.00	$votes_f$
$\langle *, 2, 1, *, 1\rangle$	42	0.76	$votes_n$
$\langle *, 2, 2, *, 1\rangle$	27	0.49	$votes_f$
$\langle *, 0, 0, *, 2\rangle$	27	base	
$\langle *, 0, 1, *, 2\rangle$	22	0.81	$votes_n$
$\langle *, 0, 2, *, 2\rangle$	50	1.85	$votes_a$
$\langle *, 1, 0, *, 2\rangle$	30	1.11	$votes_n$
$\langle *, 1, 1, *, 2\rangle$	28	1.04	$votes_n$
$\langle *, 1, 2, *, 2\rangle$	28	1.04	$votes_n$
$\langle *, 2, 0, *, 2\rangle$	100	3.70	$votes_a$
$\langle *, 2, 1, *, 2\rangle$	30	1.11	$votes_n$
$\langle *, 2, 2, *, 2\rangle$	24	0.88	$votes_n$

influence vector. The final score for this influence vector is -0.16, which indicates that this is not a correct influence vector.

3.4 Scoring: Basic Implementation

This section describes a basic implementation of the scoring function that uses raw time series data which is shown in Figure 3.7. This function determines a score for a given influence vector, i, for species s, controlling for a set of species, G. This algorithm also uses S which is a listing of the species, E, which is the time series data, θ which are the

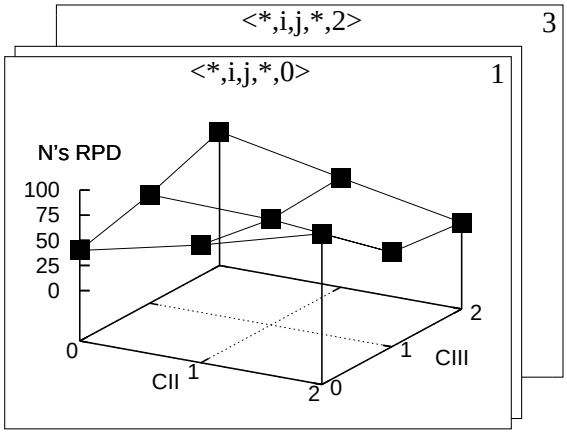


Figure 3.6. N's probability of increasing with the influence vector $i = \langle n, r, r, n, n \rangle$, where the species order is $\langle CI, CII, CIII, Cro, N \rangle$ and the set $G = \{N\}$. The probabilities used are listed in Table 3.3.

levels used to bin the data, n which is the number of bins for each species, the threshold values T_a and T_r which are used to calculate votes, the value T_i which represents the score returned for no connections, and finally the threshold value T_s which is needed by the `FindProb` function.

The goal of the `Score` function is to find probability ratios and determine votes for, against, and neutral for the given influence vector. In the first step, the function clears the votes. The function then sets all connections of type '?' to type 'n'. Next, the function checks if all species in i have no connection to s, which means that no species influence species s, and returns the score of T_i in this case. If this is not the case, the function loops over all partial bin assignments for the species in $Par(i) \cup G$. This set is constructed using the `createPartialBinAssignments` function, which constructs the set defined below:

$$\{b \in \{0, \ldots, n-1, *\}^{|S|} \mid (s \in G \cup Par(i) \wedge b(s) \neq *) \vee (s \notin G \cup Par(i) \wedge b(s) = *)\}$$

In the worst case, the total number of bin assignments created is $(n-1)^{|S|}$ where n is the number of bins for each species. For each partial bin assignment, b, the function finds a base probability as described below. If this probability is valid, (*i.e.,* $probB \neq -1$), then the probability of bin assignment, b, is calculated. If this is a valid probability, the function uses the two probabilities to construct a ratio. If there are more activators than repressors, a vote *for* the influence vector is recorded (*i.e.,* $votes_f{+}{+}$) if the ratio is larger than T_a. If the ratio is less than T_r, then a vote *against* it is recorded (*i.e.,*

```
Score(Species s, IV i, Species Set G, Species Set S, Experiments E,
      Levels θ, Number of Bins n, Thresholds T_a, T_r, T_i, T_s)
  ⟨votes_f, votes_a, votes_n⟩ := ⟨0, 0, 0⟩
  foreach s ∈ S
    if i(s) = '?' then i(s) = 'n'
  if (i(s) = (n, ..., n)) then return T_i
  foreach b ∈ createPartialBinAssignment(S, Par(i) ∪ G, n)
    probB = FindBaseProbability(s, i, G, S, E, θ, n, T_s, b)
    if probB ≠ −1 then
      probA = FindProb(s, i, G, S, E, θ, n, T_s, b)
      if probA ≠ −1 then
        if |Rep(i)| < |Act(i)| then
          if probA/probB > T_a then votes_f++
          else if probA/probB < T_r then votes_a++
          else votes_n++
        else
          if probA/probB < T_r then votes_f++
          else if probA/probB > T_a then votes_a++
          else votes_n++
  if (votes_f + votes_a + votes_n = 0) then return 0
  return (votes_f − votes_a)/(votes_f + votes_a + votes_n)
```

Figure 3.7. The basic implementation of the `Score` function.

$votes_a$++). Otherwise, a neutral vote is cast (*i.e.,* $votes_n$++). However, if there are more repressors in i, then a vote for the influence vector is cast if the ratio is less than T_r, or a vote against the influence vector is cast if the ratio is larger than T_a. Otherwise a neutral vote is cast. The reason for the difference in the way votes are cast is that if there are more activators, the probability of increasing should go up between the base and b, thus the ratio should be larger than one. It should go down if there are more repressors, and the ratio should be less than one in that case. If there are no votes cast, then a score of 0 is returned. When all the votes have been tallied, a final score is determined by taking $votes_f$ and subtracting $votes_a$ and dividing by the total number of votes (*i.e.,* $votes_f + votes_a + votes_n$). The result is a score between -1 and 1. A score close to 1 indicates that the experimental data supports the influence vector.

To determine the base probability for the bin assignment b, a probability is calculated using the species s increasing in a base configuration. This configuration uses the partial bin assignment where the species in the G set agree with those in the bin assignment

b. This is done with the `FindBaseProbability` algorithm shown in Figure 3.8. This algorithm creates the base, b', from a partial bin assignment, b, as described in Equation 3.9. It returns -1 if these two partial bin assignments are the same, as ratios are only created for different partial bin assignments. If the partial bin assignments are different, it computes the probability for the base and returns the computed value.

The `FindProb` function, shown in Figure 3.9, calculates the probability of species s increasing in the bin assignment b. This probability calculation represents one probability in Table 3.3. As shown in Figure 3.9, the basic way of calculating a probability for a bin assignment b, is to look at every pair of data points in the $SUCC \cup PRED$ set. If the data in the first time point of the pair agree with the partial bin assignment, b, this data point pair is used to calculate a probability. The function increments the *occur* count, and then determines if the data values are increasing into or out of this time point and increases the $numIncr$ count if that is the case. When all data point pairs are found that agree with b, the probability is just the number of pairs in which s increases (*i.e., numIncr*) over the number of data point pairs considered (*i.e., occur*). If there are not enough time points in the data, which is determined by using the T_s threshold, then a score of -1 is returned. The complexity of the `FindProb` function is $O(|SUCC \cup PRED| * |S|)$. This makes the total complexity of the basic implementation to be $O((n-1)^{|S|} * |SUCC \cup PRED| * |S|)$

```
FindBaseProbability(Species s, IV i, Species Set G, Species Set S,
      Experiments E, Levels θ, Number of Bins n, Threshold T_s,
      Partial Bin Assignment b)
  foreach s' ∈ G
    b'(s') = b(s')
  foreach s' ∈ S − (Par(i) ∪ G)
    b'(s') = *
  foreach s' ∈ Par(i) − G
    if (i(s') = 'a' ∧ |Rep(i)| ≤ |Act(i)|) ∨ (i(s') = 'r' ∧ |Rep(i)| > |Act(i)|) then
      b'(s') = 0
    else
      b'(s') = n − 1
  if b = b' then
    return -1
  score = FindProb(s, i, G, S, E, θ, n, T_s, b')
  return score
```

Figure 3.8. The basic implementation of the `FindBaseProbability` function.

```
FindProb(Species s, IV i, Species Set G, Species Set S,
      Experiments E, Levels θ, Number of Bins n, Threshold T_s,
      Bin Assignment b)
  ⟨occur, numIncr⟩ := ⟨0, 0⟩
  foreach (⟨e, τ, ν⟩, ⟨e', τ', ν'⟩) ∈ SUCC ∪ PRED
    if (ν(s) ∉ {H, L, −} ∧ ν'(s) ∉ {H, L, −}) then
      b' = binAssign(ν, G ∪ Par(i), θ, n)
      if PartialBA(b', S, G ∪ Par(i)) = b then
        occur++
        if ((v(s) < v'(s) ∧ τ(s) < τ'(s)) ∨ ((v'(s) < v(s) ∧ τ'(s) < τ(s))) then
          numIncr++
  if occur < T_s then return −1
  return numIncr/occur
```

Figure 3.9. The basic implementation of the `FindProb` function.

which is approximately $O((n-1)^{|S|} * |E| * |S|)$. Therefore, the cost of this function clearly increases as the number of species and the amount of time series data increases.

Basic time series data for the phage λ decision circuit are shown in Table 2.1. These data are used as an example through the rest of this section. Using the implementation described above to calculate the probability of CIII increasing, first a bin assignment is selected for the members of the G set. In this example, assume that $G = \{CI, CII, CIII\}$ and that $b(CI) = b(CII) = b(CIII) = 0$ and that the influence vector, i, is $\langle n, n, n, a, a\rangle$. Next, a bin assignment is selected from the `createBinAssignment` function. Assume that $b = \langle 0, 0, 0, 0, 1\rangle$. Next, a base probability is calculated for the bin assignment $\langle 0, 0, 0, 0, 0\rangle$ which is 100 percent. Next the `FindProb` function is called to calculate $probB$, and every time point pair in $SUCC \cup PRED$ created from in Table 2.1 is looked at to find those time points that match the bin assignment b. As this bin assignment does not occur in the data, no probability ratios can be computed and the `FindProb` function would return a -1. However if the bin assignment is $b = \langle 0, 0, 1, 0, 0\rangle$, then a probability of increasing can be calculated. The bin assignment occurs in experiment 1 at time 20, and if the species of interest is $s = CI$ then the probability returned would be 0 as CI does not increase into or out of this time point. If however the species of interest is CII, the probability returned would be 50 percent as it increases out of, but not into, this time point.

3.5 Scoring: Optimized Implementation

Since the `Score` function is called many times by the other algorithms described in this dissertation, it is crucial that it is implemented in an efficient manner. Therefore, this section presents an optimized implementation of the scoring function. The basic implementation considers each possible partial bin assignment for the species in the $Par(i) \cup G$ set, which has $(n-1)^{|S|}$ entries in the worst case. However, many partial bin assignments may never actually occur in the data. In this case, an optimized strategy for looking through the data would be to only look at the actual bin assignments that occur in the data, thereby avoiding an expensive iteration through bin assignments that are not present. This can be done by compressing and projecting the data using the `Compress` and `Project` functions described in Section 2.5 and Section 2.6. The result is that the number of partial bin assignments considered is now $|\text{domain}(PCD)|$, which is $|E|$ in the worst case. While it is possible for $|E|$ to be larger than $(n-1)^{|S|}$, it is typically substantially smaller. The improved `Score` function is shown in Figure 3.10. Like the basic function, in the first step, the function clears the votes. The function then sets all connections of type '?' to type 'n'. Next, the function checks if all species in i have no connection to s and returns the score of T_i in this case. If this is not the case, then unlike the basic function, this function constructs the PCD and loops over all partial bin assignments in the domain of PCD. The function then follows the same steps as the basic implementation, except that *probB* and *probA* are calculated using the PCD instead of E. By using the PCD, the improved function uses only the time series data entries that occur and skips potentially vast empty regions of the state space. Note that the complexity of creating the PCD is $O(|E| * |S|)$.

The use of the PCD structure also greatly simplifies the `FindProb` function, as shown in Figure 3.11. As the data have been compressed and projected, there is no longer the need to look through every time series data entry to select those entries that match the partial bin assignment. The probabilities can be calculated from a single entry in the PCD structure. The complexity of the `FindProb` function is now $O(1)$. Therefore, the overall complexity of the `Score` function has been reduced from $O((n-1)^{|s|} * |E| * |S|)$ to only $O(|E| * |S|)$.

This optimized score function results in a significant speedup. Consider, for example, the possible bin assignments to the 5 species in the phage λ decision circuit. Table 3.4 compares all the state combinations that are possible against the actual seen values from

```
Score(Species s, IV i, Species Set G, Species Set S,
      Compressed Data CD, Number of Bins n, Thresholds T_a, T_r, T_i, T_s)
  ⟨votes_f, votes_a, votes_n⟩ := ⟨0, 0, 0⟩
  foreach s ∈ S
    if i(s) = '?' then i(s) = 'n'
  if (i = (n, ..., n)) then return T_i
  PCD := Project(CD, s, S, Par(i) ∪ G)
  foreach b ∈ domain(PCD)
    probB = FindBaseProbability(s, i, G, PCD, n, T_s, b)
    if probB ≠ −1 then
      probA := FindProb(s, PCD, T_s, b)
      if probA ≠ −1 then
        if |Rep(i)| < |Act(i)| then
          if probA/probB > T_a then votes_f++
          else if probA/probB < T_r then votes_a++
          else votes_n++
        else
          if probA/probB < T_r then votes_f++
          else if probA/probB > T_a then votes_a++
          else votes_n++
  if votes_f + votes_a + votes_n = 0 then return 0
  return (votes_f − votes_a)/(votes_f + votes_a + votes_n)
```

Figure 3.10. An improved implementation of the `Score` function.

```
FindProb(Species s, Projected Compressed Data PCD,
         Threshold T_s, Partial Bin Assignment b)
  ⟨numIncr, occur⟩ := PCD(b)
  if occur < T_s then
    return −1
  else
    return numIncr/occur
```

Figure 3.11. The optimized implementation of the `FindProb` function.

Table 3.4. This table shows the savings of looking through 243 state to only 12 state by only looking through the sorted map values in our small example compared to searching through the entire state space. Although this may seem like an extreme case, it helps illustrate the time saving if the state space is sparse.

States explored in a full state space search involving 5 species with 3 levels each						
0 0 0 0 0	0 1 0 2 2	0 2 1 2 1	1 0 2 2 0	1 2 0 1 2	2 0 1 1 1	2 1 2 1 0
0 0 0 0 1	0 1 1 0 0	0 2 1 2 2	1 0 2 2 1	1 2 0 2 0	2 0 1 1 2	2 1 2 1 1
0 0 0 0 2	0 1 1 0 1	0 2 2 0 0	1 0 2 2 2	1 2 0 2 1	2 0 1 2 0	2 1 2 1 2
0 0 0 1 0	0 1 1 0 2	0 2 2 0 1	1 1 0 0 0	1 2 0 2 2	2 0 1 2 1	2 1 2 2 0
0 0 0 1 1	0 1 1 1 0	0 2 2 0 2	1 1 0 0 1	1 2 1 0 0	2 0 1 2 2	2 1 2 2 1
0 0 0 1 2	0 1 1 1 1	0 2 2 1 0	1 1 0 0 2	1 2 1 0 1	2 0 2 0 0	2 1 2 2 2
0 0 0 2 0	0 1 1 1 2	0 2 2 1 1	1 1 0 1 0	1 2 1 0 2	2 0 2 0 1	2 2 0 0 0
0 0 0 2 1	0 1 1 2 0	0 2 2 1 2	1 1 0 1 1	1 2 1 1 0	2 0 2 0 2	2 2 0 0 1
0 0 0 2 2	0 1 1 2 1	0 2 2 2 0	1 1 0 1 2	1 2 1 1 1	2 0 2 1 0	2 2 0 0 2
0 0 1 0 0	0 1 1 2 2	0 2 2 2 1	1 1 0 2 0	1 2 1 1 2	2 0 2 1 1	2 2 0 1 0
0 0 1 0 1	0 1 2 0 0	0 2 2 2 2	1 1 0 2 1	1 2 1 2 0	2 0 2 1 2	2 2 0 1 1
0 0 1 0 2	0 1 2 0 1	1 0 0 0 0	1 1 0 2 2	1 2 1 2 1	2 0 2 2 0	2 2 0 1 2
0 0 1 1 0	0 1 2 0 2	1 0 0 0 1	1 1 1 0 0	1 2 1 2 2	2 0 2 2 1	2 2 0 2 0
0 0 1 1 1	0 1 2 1 0	1 0 0 0 2	1 1 1 0 1	1 2 2 0 0	2 0 2 2 2	2 2 0 2 1
0 0 1 1 2	0 1 2 1 1	1 0 0 1 0	1 1 1 0 2	1 2 2 0 1	2 1 0 0 0	2 2 0 2 2
0 0 1 2 0	0 1 2 1 2	1 0 0 1 1	1 1 1 1 0	1 2 2 0 2	2 1 0 0 1	2 2 1 0 0
0 0 1 2 1	0 1 2 2 0	1 0 0 1 2	1 1 1 1 1	1 2 2 1 0	2 1 0 0 2	2 2 1 0 1
0 0 1 2 2	0 1 2 2 1	1 0 0 2 0	1 1 1 1 2	1 2 2 1 1	2 1 0 1 0	2 2 1 0 2
0 0 2 0 0	0 1 2 2 2	1 0 0 2 1	1 1 1 2 0	1 2 2 1 2	2 1 0 1 1	2 2 1 1 0
0 0 2 0 1	0 2 0 0 0	1 0 0 2 2	1 1 1 2 1	1 2 2 2 0	2 1 0 1 2	2 2 1 1 1
0 0 2 0 2	0 2 0 0 1	1 0 1 0 0	1 1 1 2 2	1 2 2 2 1	2 1 0 2 0	2 2 1 1 2
0 0 2 1 0	0 2 0 0 2	1 0 1 0 1	1 1 2 0 0	1 2 2 2 2	2 1 0 2 1	2 2 1 2 0
0 0 2 1 1	0 2 0 1 0	1 0 1 0 2	1 1 2 0 1	2 0 0 0 0	2 1 0 2 2	2 2 1 2 1
0 0 2 1 2	0 2 0 1 1	1 0 1 1 0	1 1 2 0 2	2 0 0 0 1	2 1 1 0 0	2 2 1 2 2
0 0 2 2 0	0 2 0 1 2	1 0 1 1 1	1 1 2 1 0	2 0 0 0 2	2 1 1 0 1	2 2 2 0 0
0 0 2 2 1	0 2 0 2 0	1 0 1 1 2	1 1 2 1 1	2 0 0 1 0	2 1 1 0 2	2 2 2 0 1
0 0 2 2 2	0 2 0 2 1	1 0 1 2 0	1 1 2 1 2	2 0 0 1 1	2 1 1 1 0	2 2 2 0 2
0 1 0 0 0	0 2 0 2 2	1 0 1 2 1	1 1 2 2 0	2 0 0 1 2	2 1 1 1 1	2 2 2 1 0
0 1 0 0 1	0 2 1 0 0	1 0 1 2 2	1 1 2 2 1	2 0 0 2 0	2 1 1 1 2	2 2 2 1 1
0 1 0 0 2	0 2 1 0 1	1 0 2 0 0	1 1 2 2 2	2 0 0 2 1	2 1 1 2 0	2 2 2 1 2
0 1 0 1 0	0 2 1 0 2	1 0 2 0 1	1 2 0 0 0	2 0 0 2 2	2 1 1 2 1	2 2 2 2 0
0 1 0 1 1	0 2 1 1 0	1 0 2 0 2	1 2 0 0 1	2 0 1 0 0	2 1 1 2 2	2 2 2 2 1
0 1 0 1 2	0 2 1 1 1	1 0 2 1 0	1 2 0 0 2	2 0 1 0 1	2 1 2 0 0	2 2 2 2 2
0 1 0 2 0	0 2 1 1 2	1 0 2 1 1	1 2 0 1 0	2 0 1 0 2	2 1 2 0 1	
0 1 0 2 1	0 2 1 2 0	1 0 2 1 2	1 2 0 1 1	2 0 1 1 0	2 1 2 0 2	
States explored when using only those values that occur in the data						
0 0 0 0 0	0 0 1 0 0	0 1 1 0 0	0 1 1 0 1	0 1 1 1 1	0 2 1 0 0	
0 2 1 0 1	0 2 1 0 2	0 2 2 0 0	0 2 2 0 1	0 2 2 0 2	0 2 2 1 1	

Table 2.1. It is easy to see from this table that searching through all possible states is not very efficient, as there are many states that do not appear in the data. The state space increases exponentially as the number of bins used to encode the data increases. It also increases exponentially as the number of species passed to the `Score` function increases. These reasons make it more efficient to only search through those states that are actually observed in the data.

There is one further optimization that we make to the `Score` function which does not improve upon the worst case complexity, but does improve the speed of the function in general. This optimization is used when the `Score` function is called where $|Par(i) \cup G| = 2$. To optimize this case, the `Project` function can precompute these PCDs. These PCDs can be created by traversing the time series data a single time. Table 3.5 shows the PCDs that are created for the phage λ decision circuit example for the species CIII. If $G = CIII$ and $i = \langle a, n, n, n, n \rangle$, PCD one in Table 3.5 is returned by the `Project` function.

3.6 Scoring: Sparse Data Issues

When data are sparse, as is typically the case, the base bin assignment may have little or no data, making it not possible to calculate a base probability. In this case, no votes are recorded for any ratio that requires this base probability. This section describes how an alternative base probability can be calculated in some cases allowing more votes to be recorded.

To determine an alternate base probability, first note that all partial bin assignments for a set of species specified in an influence vector can be organized into a lattice. For example, the influence vector $\langle a, a \rangle$ with 3 bins per species results in the lattice in Figure 3.12. The ideal base partial bin assignment for these two species is the bottom element, $\langle 0, 0 \rangle$, which appears at lattice level 0. The lattice is organized based upon the expected activity of the influencing species. If the two activators are at their lowest bins, lattice level 0, then the species that they influence should be at its lowest activity level. If either activating species increases to their next highest bin (*i.e.,* lattice level 1) then activity of the species that they influence should increase in expression. Finally, if the two activators are at their highest bins, lattice level 4, then the species they influence should be at its highest level of production.

A similar lattice can be constructed for the influence vector $\langle a, r \rangle$, however with one

Table 3.5. The PCDs created for the single parent cases in the phage λ decision circuit assuming the species of interest is CIII. The $PCD_{CIII}(b)$ column is shown only for comparison purposes with Table 2.2 and do not necessarily need to be stored. Also note that other PCDs are created for the other species in the phage λ decision circuit example.

PCD	i Level	G Set Level	$PCD_{CIII}(b)$	Probability.
1	CI	CIII		
	0	0	$\langle 3,3\rangle$	100
	0	1	$\langle 18,20\rangle$	90
	0	2	$\langle 9,10\rangle$	90
2	CII	CIII		
	0	0	$\langle 3,3\rangle$	100
	0	1	$\langle 1,1\rangle$	100
	1	1	$\langle 7,8\rangle$	87
	2	1	$\langle 10,11\rangle$	90
	2	2	$\langle 9,10\rangle$	90
3	Cro	CIII		
	0	0	$\langle 3,3\rangle$	100
	0	1	$\langle 17,18\rangle$	94
	0	2	$\langle 7,7\rangle$	100
	1	1	$\langle 1,2\rangle$	50
	1	2	$\langle 2,3\rangle$	66
4	N	CIII		
	0	0	$\langle 3,3\rangle$	100
	0	1	$\langle 6,6\rangle$	100
	0	2	$\langle 1,1\rangle$	100
	1	1	$\langle 7,9\rangle$	77
	1	2	$\langle 6,7\rangle$	85
	2	1	$\langle 5,5\rangle$	100
	2	2	$\langle 2,2\rangle$	100

key difference. The lowest level of the lattice starts with the species having an activation influence at the lowest bin and the species having a repressive influence at the highest bin, as is shown in Figure 3.13. The reason the lattice is organized in this manner is that, similar to before, the species these two species influence is at the lowest level of activity at lattice level 0, and the highest level of activity at lattice level 4. For example, when the activating species is in its lowest bin and the repressing species is in its highest bin, the rate of production of the species they influence should be at its minimum activity level. As the activating species is increased, or the repressive species decreased the species they influence increases in activity.

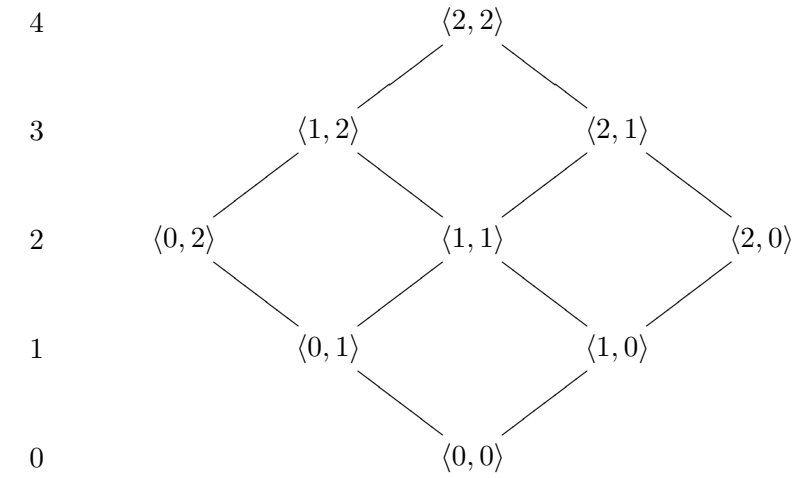


Figure 3.12. The lattice organization for the influence vector $\langle a, a \rangle$.

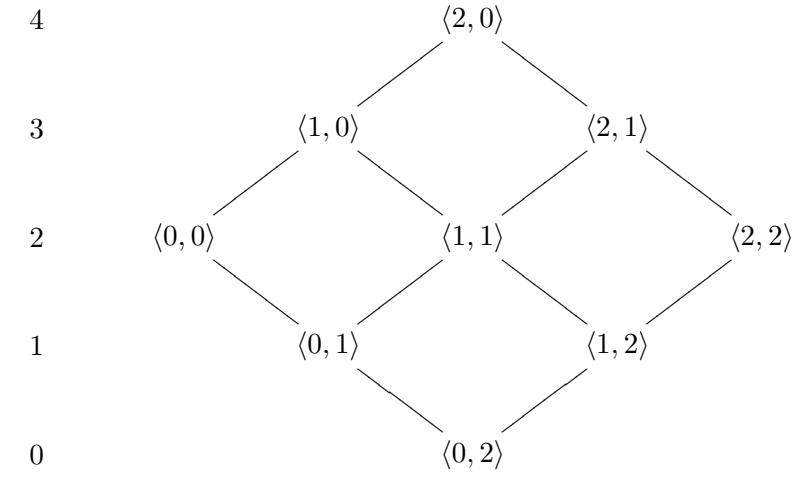


Figure 3.13. The lattice organization for the influence vector $\langle a, r \rangle$.

In general, a lattice is organized in the following manner. First, if $|Rep(i)| < |Act(i)|$ then the lowest level of the lattice contains the activating species in their lowest bins, and the repressing species in their highest bins. Otherwise, the lowest level of the lattice contains the opposite assignment to the bins. To determine the level of the lattice for a particular partial bin assignment, the `LatticeLevel` function, shown in Figure 3.14, is used. The `LatticeLevel` function works as follows. It first determines if there is a lattice by checking that there are parents in the influence vector i. It then determines how the lattice is organized by calculating if there are more activators or repressors in the influence vector. The algorithm then calculates how far up the lattice the bin assignment is by summing up how far each bin assignment is from the base level. This number is returned as the level the bin assignment falls on. The partial bin assignments organized in a lattice represent a partial ordering. If $b < b'$ then b occurs on a lower level of the lattice than b' and there is a direct connection between them.

Ideally, the base probability for a partial bin assignment, b, would be calculated from the partial bin assignment that appears on lattice level zero. If the bin assignment that appears on lattice level zero is not present in the data, then the next best partial bin assignments to compare against would be all partial bin assignments on lattice level one

```
LatticeLevel(Bin Assignment b, IV i, Number of Bins n)
  level := 0
  if |Par(i)| = 0 then
    return level
  else if |Rep(i)| < |Act(i)| then
    foreach s ∈ Act(i)
      level := level + b(s)
    foreach s ∈ Rep(i)
      level := level + (n − 1) − b(s)
    return level
  else
    foreach s ∈ Act(i)
      level := level + (n − 1) − b(s)
    foreach s ∈ Rep(i)
      level := level + b(s)
    return level
```

Figure 3.14. The `LatticeLevel` function.

which are comparable to b. Assuming that b appears on level two or higher, there may be more than one comparable partial bin assignment on level one. As only one base probability is used, the algorithm averages the probability of all partial bin assignments at level one that are comparable to b. For example, if $b = \langle 1, 1 \rangle$, then the base probability would be calculated from $\langle 0, 1 \rangle$ and $\langle 1, 0 \rangle$. If there is not enough information at level one, then the next highest level is used. This continues until a probability can be calculated or the algorithm reaches the level of the lattice on which b occurs. The `FindProb` function shown in Figure 3.15 is used to calculate the probability for those bin assignments on the lattice level that are directly comparable. The function first sets *numIncr* and *occur* to zero. It then looks at all partial bin assignments in the set B, and sums the *numIncr* and *occur* values that appear in PCD for those partial bin assignments. If there are not enough data, then a score of -1 is returned. Otherwise the probability of species s increasing in the set, B, is returned.

The advanced implementation of the `FindBaseProbability` function is shown in Figure 3.16. This function calculates the base probability as follows. It first partitions those bin assignments, b', in the domain of PCD that have the same bin assignment for the species in G, and are also related to b, based on their lattice level. It then calculates a probability using the `FindProb` function starting at the lowest lattice level. If no valid probability is found, then it uses the next highest lattice level until either a valid probability is found, or no more partitions exist.

Dealing with sparse data increases the complexity of the `Score` function but allows more votes to be computed, which generates better results. The complexity of the `Score` function increases from

```
FindProb(Species s, Projected Compressed Data PCD,
        Threshold T_s, Partial Bin Assignment Set B)
  ⟨numIncr, occur⟩ := ⟨0, 0⟩
  foreach b ∈ B
    ⟨numIncr, occur⟩ := ⟨numIncr, occur⟩ + PCD(b)
  if occur < T_s then
    return −1
  else
    return numIncr / occur
```

Figure 3.15. The advanced implementation of the `FindProb` function.

```
FindBaseProbability(Species s, IV i, Species Set G,
        Projected Compressed Data PCD, Number of Bins n,
        Threshold T_s, Partial Bin Assignment b)
  foreach b' ∈ domain(PCD)
    if (∀s ∈ G.b'(s) = b(s)) then
      if b' < b then
        P_j = P_j ∪ {b'} where j = LatticeLevel(b, i, n)
  for j = 0 to LatticeLevel(b, i, n) − 1
    α = FindProb(s, PCD, T_s, P_j)
    if α ≠ −1 then return α
  return −1
```

Figure 3.16. The advanced implementation of the `FindBaseProbability` function.

$$O(|E| * |S|) \tag{3.10}$$

to

$$O(|E| * |S| + |E|^2 * |S|) \tag{3.11}$$

because of the added complexity of the `FindBaseProbability` function. The complexity of this function comes from looking at every partial bin assignment occurring in the data and calculates the lattice level for that partial bin assignment. This complexity can be reduced to:

$$O(|E| * |S| + |E|^2) \tag{3.12}$$

because the lattice level of each partial bin assignment could be computed in the `Project` function, and stored with the bin assignment at no added complexity.

CHAPTER 4

LEARNING INFLUENCE VECTORS

A little learning is a dangerous thing, but a lot of ignorance is just as bad.
–Bob Edwards

The difference between what the most and the least learned people know is inexpressibly trivial in relation to that which is unknown.
–Albert Einstein

Only those who have learned a lot are in a position to admit how little they know.
–L. Carte

Learning influences between species in genetic networks is currently done in many ways. Section 1.2 describes many different approaches to the problem and each approach has distinct advantages and disadvantages. This chapter focuses on our approach, the `GeneNet` algorithm. The `GeneNet` algorithm, shown in Figure 4.1, breaks the problem up into three different learning tasks. The first is searching for an initial set of species that seem to influence a species. The second is determining if these species seem to act together or separately. The final task is selecting the influence vector that best represents the information found in the data.

The inputs to the `GeneNet` algorithm are the set of species of interest, S, the set of experiments, E, the initial influence vector collection, $\mathcal{I}$, which includes any background knowledge, the number of bins used to group the data, n, and the set of threshold values, T, that are used when scoring potential influence vectors. The default threshold values are shown in Table 4.1. The meaning of these values are discussed when describing the functions or algorithms they appear in. First, the `GeneNet` algorithm determines the levels used to bin the data for each species. The default number of bins used is four. Next the data are compressed as explained in Section 2.5 to form the CD structure to optimize scoring efficiency. Next, each species, s, is considered separately to determine the species

```
GeneNet(Species Set S, Expts E, Influences I, Number of Bins n,
        Thresholds T)
  θ := DetermineLevels(S, E, n)
  CD := Compress(E, S, θ)
  foreach s ∈ S
    I := CreateInfluenceVectorSet(s, S, CD, I, θ, n, T)
    I := CombineInfluenceVectors(s, I, S, CD, I, θ, n, T)
    if |I| > 0 then
      I(s) := CompeteInfluenceVectors(s, I, S, CD, θ, n, T)
  return I
```

Figure 4.1. Algorithm for finding genetic network connectivity.

Table 4.1. Various thresholds used in the `GeneNet` algorithm.

Threshold	Default Value	Note
T_a	1.15	A threshold passed to the `Score` function which is used to cast a vote in favor of an activating influence vector when a probability ratio is greater than this threshold value.
T_r	0.75	A threshold passed to the `Score` function which is used to cast a vote in favor of a repressing influence vector when a probability ratio is less than this threshold value.
T_i	0.5	A threshold passed to the `Score` function representing the score of an influence vector with no influences in it (*i.e.*, $\|i\| = 0$).
T_n	2	This value is used in the `CreateInfluenceVectorSet` algorithm and represents the minimum number of influence vectors constructed in this algorithm.
T_j	2	This value is used in the `CombineInfluenceVectors` algorithm to determine the maximal size of merged influence vectors.
T_m	0	A threshold used in the `CombineInfluenceVectors` algorithm to determine if two influence vectors have a close enough score to even be considered for merging.
T_t	0.025	This value is used when relaxing thresholds.
T_s	10	A threshold used in the `FindProb` function to determine if there is a significant amount of data.

that may activate or repress the gene that produces s. The `CreateInfluenceVectorSet` algorithm derives a set of potential influence vectors, I, to assign to $\mathcal{I}(s)$. Each of these vectors differs from the initial value of $\mathcal{I}(s)$ in that at most one unknown entry is set to 'a' or 'r' while all other unknown entries are set to 'n'. Next, `CombineInfluenceVectors` determines if any members of I can be combined to form a larger influence vector with a higher score. Finally, if the $|I|$ is greater than zero, the members of I are competed against each other to determine the final vector of influences on s. The result is a new influence vector, $i = \mathcal{I}(s)$, which includes only those activation or repression influences supported by the data, and it no longer includes any unknown influences. This section describes this algorithm in more detail and uses the learning of influences for the CIII species in the phage λ decision circuit as a running example. Initially in the running example, the algorithm starts with no information known about the connections to species CIII, except that CIII does not influence itself. This is shown graphically in Figure 4.2. Note to simplify the presentation, only three bins are used in the examples.

4.1 Creating The Influence Vector Set

The `CreateInfluenceVectorSet` algorithm shown in Figure 4.3 is used to compute an initial set of potential influence vectors, I, for a species s. In particular, each species s' with an unknown influence in the background knowledge, $\mathcal{I}(s)$, is considered as a potential activator or repressor. First, the set of influence vectors, I, is set to the empty set. Next, a score, α, is obtained from $\mathcal{I}(s)$. For every species, s', that has an unknown connection in the background knowledge, a new influence vector, i, is created from $\mathcal{I}(s)$

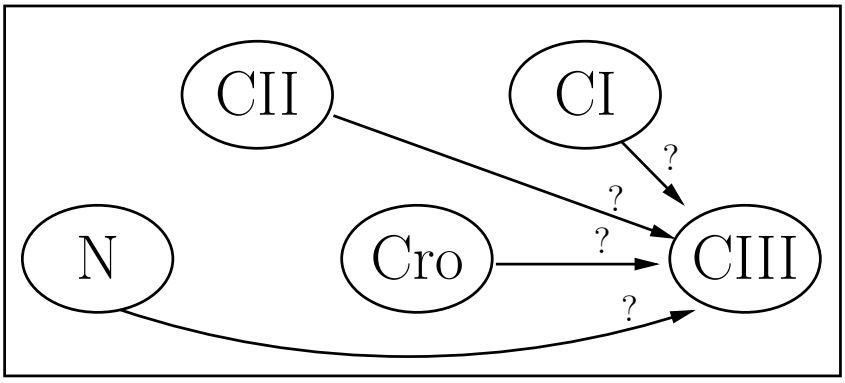


Figure 4.2. Representation of initial knowledge for species CIII.

```
CreateInfluenceVectorSet(Species s, Species Set S, Compressed Data
        CD, Influences I, Levels θ, Number of Bins n, Thresholds T)
  I := ∅
  do
    α := Score(s, I(s), {s}, S, n, CD, T_a, T_r, T_i, T_s)
    foreach s' such that I(s') = ?
      i := I(s)
      i(s') := a
      foreach s'' ∈ (S − {s'})
        if (i(s'') =?) then i(s'') := n
      if (Score(s, i, {s}, S, CD, n, T_a, T_r, T_i, T_s) ≥ α) then
        I := I ∪ i
      else
        i(s') := r
        if (Score(s, i, {s}, S, CD, n, T_a, T_r, T_i, T_s) ≥ α) then
          I := I ∪ i
    (T_a, T_r, T_i) := RelaxThresholds(T_a, T_r, T_i, T_t)
  while |I| < T_n
  return I
```

Figure 4.3. The `CreateInfluenceVectorSet` algorithm.

where the influence for s' is set to 'a', and every other species with unknown influence is set to 'n'. If the score of this new influence vector is greater than α, then it is added to I. Otherwise, $i(s')$ is changed to 'r', and this new vector is scored. If the score is greater than α, then it is retained. If, after considering all species with unknown influence, the size of I is less than T_n, then the `RelaxThresholds` function is called to relax the thresholds. The threshold T_n is the value representing the minimum number of influence vectors that should be found in this algorithm. The T_t threshold is the value used to relax the other thresholds. For example, the T_a threshold is decreased and the T_r threshold is increased by the T_t threshold. If both T_r and T_a have been relaxed to 1, then the T_i threshold is relaxed. The entire process is repeated except the newly relaxed thresholds are passed to the *Score* function to allow more influence vectors to be created.

With CIII as the species of interest, the `CreateInfluenceVectorSet` algorithm considers eight possible influence vectors containing only one influence as shown in Table 4.2. The probabilities and ratios determined by the `Score` function for the eight influence vectors that appear in Table 4.2 are shown in Table 4.3. The `Score` function uses these

Table 4.2. The eight influence vectors containing only a single influence constructed from the blank influence vector for CIII graphically represented in Figure 4.2

Influence Vectors			
CI	CII	Cro	N
$\langle a, n, n, n, n\rangle$	$\langle n, a, n, n, n\rangle$	$\langle n, n, n, a, n\rangle$	$\langle n, n, n, n, a\rangle$
$\langle r, n, n, n, n\rangle$	$\langle n, r, n, n, n\rangle$	$\langle n, n, n, r, n\rangle$	$\langle n, n, n, n, r\rangle$

Table 4.3. Probabilities and ratios determined by the `Score` function for the influence vectors from Table 4.2 where the species list is $\langle CI, CII, CIII, Cro, N\rangle$.

IV	$P(inc(CIII) \mid bin(b) \cap val(CIII))$				Ratios	
		B_0(CI)	B_1(CI)	B_2(CI)	B_1(CI)/B_0(CI)	B_2(CI)/B_0(CI)
$\langle a, n, n, n, n\rangle$	B_0(CIII)	19.0%	1.7%	1.0%	0.09	0.05
$\langle r, n, n, n, n\rangle$	B_1(CIII)	17.1%	2.6%	1.2%	0.15	0.07
	B_2(CIII)	11.6%	2.7%	1.1%	0.23	0.09
		B_0(CII)	B_1(CII)	B_2(CII)	B_1(CII)/B_0(CII)	B_2(CII)/B_0(CII)
$\langle n, a, n, n, n\rangle$	B_0(CIII)	3.1%	13.7%	–	4.32	–
$\langle n, r, n, n, n\rangle$	B_1(CIII)	4.4%	7.4%	12.6%	1.65	2.83
	B_2(CIII)	19.4%	5.5%	6.8%	0.28	0.35
		B_0(Cro)	B_1(Cro)	B_2(Cro)	B_1(Cro)/B_0(Cro)	B_2(Cro)/B_0(Cro)
$\langle n, n, n, a, n\rangle$	B_0(CIII)	11.5%	1.8%	1.5%	0.16	0.13
$\langle n, n, n, r, n\rangle$	B_1(CIII)	14.2%	4.7%	3.1%	0.33	0.23
	B_2(CIII)	9.7%	5.0%	4.2%	0.52	0.43
		B_0(N)	B_1(N)	B_2(N)	B_1(N)/B_0(N)	B_2(N)/B_0(N)
$\langle n, n, n, n, a\rangle$	B_0(CIII)	5.4%	2.9%	3.7%	0.53	0.68
$\langle n, n, n, n, r\rangle$	B_1(CIII)	9.3%	7.2%	6.7%	0.78	0.72
	B_2(CIII)	8.6%	6.4%	6.1%	0.75	0.71

ratios to calculate votes and then a final score for the influence vectors. These votes, and the scores calculated from them, are shown in Table 4.4.

For the influence vector $\langle a, n, n, n, n\rangle$, a score of -1.00 is returned from the Score function. As this is less than the T_i threshold, this edge $i(CI)$ is set to 'r' and the resulting influence vector $\langle r, n, n, n, n\rangle$ is scored. The score obtained from this vector is 1.00 and it is added to the set of potential influence vectors. The `CreateInfluenceVectorSet` then constructs the other influence vectors in turn. As each influence vector with an 'a' edge has a score lower than the T_i threshold these influence vectors are all discarded and the influence vector with an 'r' influence is scored. Of the eight influence vectors shown in Table 4.4 only three are retained which are shown in Table 4.5. As there are three vectors found, the thresholds passed to the `Score` function do not need to be relaxed and

Table 4.4. Votes and scores calculated in the `Score` function from the probabilities in Table 4.3 for influence vectors containing only a single influence directed towards species CIII in the phage λ decision circuit.

$\langle CI, CII, CIII, Cro, N\rangle$	votes$_f$	votes$_a$	votes$_n$	Score
$\langle a, n, n, n, n\rangle$	0	6	0	-1.00
$\langle r, n, n, n, n\rangle$	6	0	0	1.00
$\langle n, a, n, n, n\rangle$	3	2	0	0.20
$\langle n, r, n, n, n\rangle$	2	3	0	-0.20
$\langle n, n, n, a, n\rangle$	0	6	0	-1.00
$\langle n, n, n, r, n\rangle$	6	0	0	1.00
$\langle n, n, n, n, a\rangle$	0	5	1	-0.83
$\langle n, n, n, n, r\rangle$	5	0	1	0.83

Table 4.5. Influence vectors in the I set for the species CIII in the phage λ decision circuit which are returned by the `CreateInfluenceVectorSet` .

$\langle CI, CII, CIII, Cro, N\rangle$
$\langle r, n, n, n, n\rangle$
$\langle n, n, n, r, n\rangle$
$\langle n, n, n, n, r\rangle$

the influence vectors do not need to be rescored. However, if no influence vectors had a high enough score, the T_a and the T_r thresholds would be relaxed to $T_a = T_a - T_t$ and $T_r = T_r + T_t$ and the influence vectors would be rescored. In general, the T_a threshold is never relaxed below 1.0, and the T_r threshold is never relaxed above 1.0.

4.2 Combining Influence Vectors

After the initial influence vectors are determined, the `CombineInfluenceVectors` algorithm shown in Figure 4.4 is applied to construct new influence vectors by forming combinations of those found initially. This algorithm computes the set of influences that are of size k, starting at a size of two larger than the background knowledge for species s, by combining two influence vectors in the I set. For each of the combined influence vectors, i', the α_{min} and α_{max} score are computed from the influence vector scores that are a subset of i' and occur in I. An influence vector i is a subset of another vector i' if it has matching 'a' or 'r' influence in the same spots, or has 'n' influences where 'a' or

```
CombineInfluenceVectors(Species s, IV Set I, Species Set S,
        Compressed Data CD, Influences 𝓘, Levels θ,
        Number of Bins n, Thresholds T)
  for k = |𝓘(s)| + 2 to T_j
    I' = {i ∪ i' | i ∈ I ∧ i' ∈ I ∧ |i ∪ i'| = k}
    foreach i' ∈ I'
      α_min := ∞
      α_max := −∞
      foreach i ∈ I such that i ⊆ i'
        α := Score(s, i, {s}, S, CD, n, T_a, T_r, T_i, T_s)
        if α < α_min then
          α_min := α
        if α > α_max then
          α_max := α
      if α_min + T_m < α_max then
        I' := I' − {i'}
      else
        α' := Score(s, i', {s}, S, CD, n, T_a, T_r, T_i, T_s)
        if α' < α_max then
          I' := I' − {i'}
    I := RemoveSubsets(I ∪ I')
  I := Filter(s, 𝓘(s), I, S, CD, n, T)
  return I
```

Figure 4.4. The `CombineInfluenceVectors` algorithm.

'r' influences appears in i'. Formally $i \subseteq i'$ if $\forall s.(i(s) = n \vee i(s) = i'(s))$. For example, $\langle a, r, n\rangle \subseteq \langle a, r, a\rangle$, $\langle r, n, n\rangle \not\subseteq \langle n, r, n\rangle$, and $\langle r, n, n\rangle \not\subseteq \langle a, n, n\rangle$. Next, the algorithm determines if α_{max} and α_{min} are within T_m of each other and removes i' from I' if they are not. If the scores are close to each other, the algorithm scores the combined influence vector i'. If this score is less than α_{max}, the influence vector is removed from the set. Otherwise it remains in the set. After all combined influence vectors in I' are considered, the newly created vectors, I' are merged with I and any vectors that are subsets of other vectors are removed from I by the *RemoveSubsets* function shown in Figure 4.5. This process repeats until all influence vectors are merged up to size T_j. Finally, any influence vectors that do not have a score better than the background knowledge are filtered from the set using the `Filter` function shown in Figure 4.6. This function scores the background knowledge, and returns every influence vector in I that has a score that

```
RemoveSubsets(IV Set I)
  foreach i ∈ I
    foreach i' ∈ I − {i}
      if i ⊆ i' then
        I := I − {i}
        break
  return I
```

Figure 4.5. The `RemoveSubsets` function.

```
Filter(Species s, IV i,  IV Set I, Species Set S,
       Compressed Data CD, Number of Bins n, Thresholds T)
  α := Score(s, i, {s}, S, CD, n, T_a, T_r, T_i, T_s)
  foreach i' ∈ I
    α' := Score(s, i', {s}, S, CD, n, T_a, T_r, T_i, T_s)
    if α' > α then
      I' := I' ∪ {i'}
  return I'
```

Figure 4.6. The `Filter` function.

is better than the background knowledge score.

Using the phage λ decision circuit example, the `CombineInfluenceVectors` algorithm starts with the three influence vectors shown in Table 4.5. The `CombineInfluenceVectors` algorithm then selects two of these influences i and i' and scores them. If their scores are within the T_m threshold it tries to merge them. For example, the first two influence vectors $\langle r, n, n, n, n\rangle$ and $\langle n, n, n, r, n\rangle$ are scored, and the scores obtained are 1.0 in both cases. Since these scores are equal, they are within the T_m threshold and are merged to $\langle r, n, n, r, n\rangle$. It then calculates the merged score with the `Score` function, and a score of 1.0 is returned in this case. Figure 4.7 represents the 27 probabilities that are calculated during the `Score` function for the merged influence vector. As the score of 1.0 is as good as both single influence vector scores this vector is added to the list of influence vectors. After all other combinations are tried, the `RemoveSubsets` function removes both $\langle r, n, n, n, n\rangle$ and $\langle n, n, n, r, n\rangle$ vectors as $\langle r, n, n, n, n\rangle \subseteq \langle r, n, n, r, n\rangle$ and $\langle n, n, n, r, n\rangle \subseteq \langle r, n, n, r, n\rangle$. Table 4.6 shows the scores for the three influence vectors

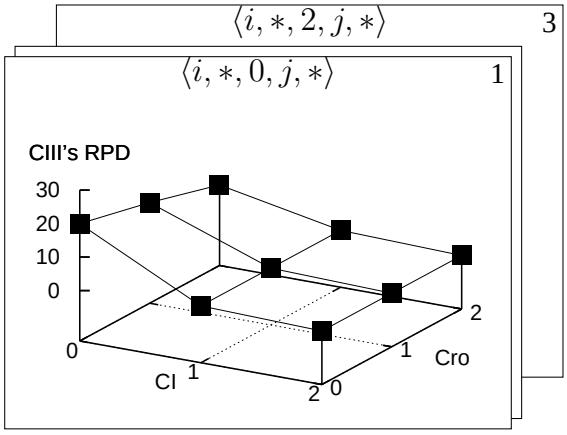


Figure 4.7. CIII's probability of increasing with the influence vector $i = \langle r, n, n, r, n \rangle$, where the species order is $\langle CI, CII, CIII, Cro, N \rangle$, and $G = \{CIII\}$.

Table 4.6. Votes and scores calculated in the `CombineInfluenceVectors` algorithm for influence vectors containing both single and multiple influences directed towards species CIII in the phage λ decision circuit.

$\langle CI, CII, CIII, Cro, N \rangle$	$votes_f$	$votes_a$	$votes_n$	Score
$\langle r, n, n, n, n \rangle$	6	0	0	1.0
$\langle n, n, n, r, n \rangle$	6	0	0	1.0
$\langle n, n, n, n, r \rangle$	5	0	1	0.833
$\langle r, n, n, r, n \rangle$	24	0	0	1.0
$\langle r, n, n, n, r \rangle$	21	0	3	0.875
$\langle n, n, n, r, r \rangle$	23	0	1	0.958
$\langle r, n, n, r, r \rangle$	71	1	6	0.897

with a single parent for CIII as well as for influence vectors found by merging these vectors. As can be seen from the table, only the $\langle r, n, n, r, n \rangle$ vector mentioned above has a score at least as good as both of the influence vectors merged to create it and is the only one added to the I set. Therefore, the influence vector set I contains only the two influence vectors shown in Table 4.7 at this point.

Once all four of the merged vectors are scored for the phage λ decision circuit, the I set is filtered to remove influence vectors that have a score less than the background knowledge. All the influence vectors pass this threshold in the phage λ decision circuit example. In general, if the bounds are relaxed in the `CreateInfluenceVectorSet` algo-

Table 4.7. Influence vectors that makeup the I set for the species CIII in the phage λ decision circuit which are returned by the `CombineInfluenceVectors` algorithm.

$\langle CI, CII, CIII, Cro, N \rangle$
$\langle n, n, n, n, r \rangle$
$\langle r, n, n, r, n \rangle$

rithm allowing influence vectors with a low score through to then potentially be merged, these influence vectors would get filtered at this step.

4.3 Competing Influence Vectors

The `CompeteInfluenceVectors` algorithm shown in Figure 4.8 competes the remaining influence vectors until only one remains. It begins by selecting two influence vectors i and i' from I. In general, when choosing a pair of influence vectors, the algorithm ranks the set of influence vectors according to their score. The influence vector with the highest score is paired with the influence vector with the lowest score, the second highest against the second lowest, and so on until all vectors are paired. The algorithm does this to allow the best influence vectors to not eliminate each other

```
CompeteInfluenceVectors(Specie s, IV Set I, Species S,
      Compressed Data CD, Levels θ, Number of Bins n, Thresholds T)
  while |I| > 1
    (i, i') = selectPair(I)
    (T'_a, T'_r) := (T_a, T_r)
    do
      α := Score(s, i, (Par(i') − Par(i)) ∪ {s}, S, CD, n, T'_a, T'_r, T_i, T_s)
      α' := Score(s, i', (Par(i) − Par(i')) ∪ {s}, S, CD, n, T'_a, T'_r, T_i, T_s)
      (T'_a, T'_r) := StrengthenThresholds(T'_a, T'_r, T_t)
    while (α = α' ∧ T'_r > 0 ∧ T'_a < 5)
    if (α > α') ∨ (α = α' ∧ |i| < |i'|) then
      I := I − {i'}
    else
      I := I − {i}
  i := select(I)
  return i
```

Figure 4.8. The `CompeteInfluenceVectors` algorithm.

until the last few rounds of the competition. Next, it makes a copy of the thresholds used during scoring in case they need to be strengthened. It then obtains a score for i controlling for any species in i' and not in i by performing the following set operation $(Par(i') - Par(i)) \cup \{s\}$. For example, with $i = \langle a, a, n, n, n\rangle$ and $i' = \langle a, n, n, n, a\rangle$, $(Par(i') - Par(i)) \cup \{s\} = (\{CI, N\} - \{CI, CII\}) \cup \{CIII\} = \{CIII, N\}$, as the first species, CI, appears in both sets and is removed by the set subtraction operation. Similarly, the `CompeteInfluenceVectors` algorithm computes a score for i' controlling for those species with influence in i but not in i'. If both influence vectors scores are tied initially, it then computes a stronger set of thresholds which it uses to rescore the influence vectors. The thresholds are strengthened using the `StrengthenThresholds` function which increases the T'_a threshold and decreases the T'_r threshold by the T_t threshold until the scores are no longer tied or the T'_r is less than zero, and T'_a is greater than 5. If the thresholds are strengthened too much, then the influence vector with the most influences in it is removed. If there is a clear winner, then the influence vector with the lowest score is removed from I. This process continues until only one influence vector remains which becomes the final result for this species.

In the phage λ decision circuit example, after the `CombineInfluenceVectors` algorithm is used, there are only two remaining influence vectors for the species CIII as shown in Table 4.7 which are competed. When competing $\langle n, n, n, n, r\rangle$ against $\langle r, n, n, r, n\rangle$, the `Score` function is called twice. The first time it scores the first vector, controlling for those species in the second influence vector. Figure 4.9 represents the probabilities and ratios that would be found in the `Score` function during this competition step. Next the second influence vector is scored, controlling for the first influence vector. Figure 4.10 represents the probabilities and ratios that would be found in the `Score` function during this competition step. The votes and scores obtained while competing the two influence vectors are shown in Table 4.8. The result is that the `GeneNet` algorithm has determined that CIII is repressed by both Cro and CI as it has the higher score.

The `GeneNet` algorithm also learns influence vectors for the other species in the phage λ decision circuit network. These influence vectors are shown in Figure 4.11. All the arcs reported are in the actual phage λ decision circuit, shown in Figure 3.3. Our method, however, does not find 2 activation arcs, 1 repression arc and the 2 self-arcs.

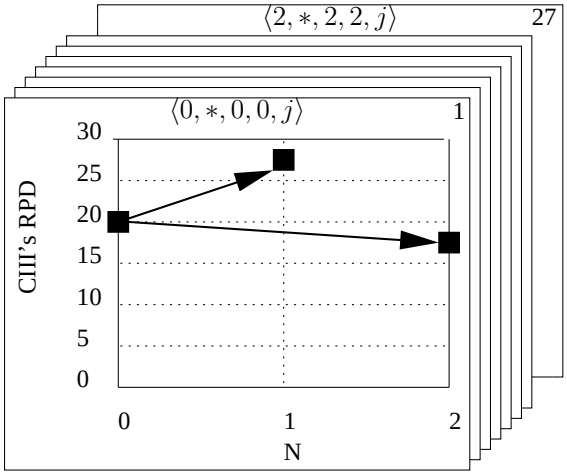


Figure 4.9. CIII's probability of increasing with the influence vector $i = \langle n, n, n, n, r\rangle$, where the species order is $\langle CI, CII, CIII, Cro, N\rangle$, and $G = \{CI, CIII, Cro\}$.

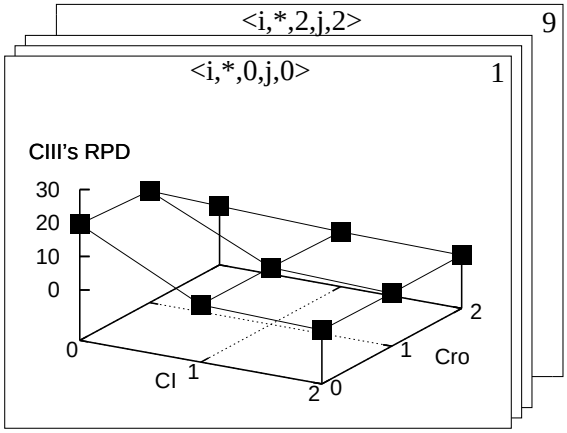


Figure 4.10. CIII's probability of increasing with the influence vector $i = \langle r, n, n, r, n\rangle$, representing $\{CI, Cro\} \dashv CIII$ and $G = \{CIII, N\}$.

Table 4.8. Scores from the `CreateInfluenceVectorSet` algorithm for species CIII.

$\langle CI, CII, CIII, Cro, N\rangle$	$votes_f$	$votes_a$	$votes_n$	Score
$\langle n, n, n, n, r\rangle$	12	27	15	0.27
$\langle r, n, n, r, n\rangle$	71	0	1	0.98

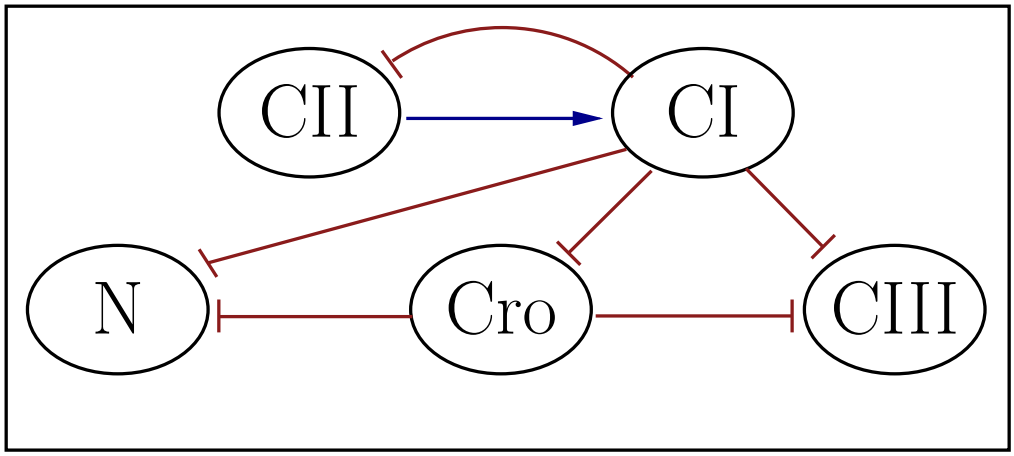


Figure 4.11. The influence vectors that `GeneNet` learned for the species in the phage λ decision circuit. All the arc reported are in the actual phage λ decision circuit influence vectors shown in Figure 3.3. There are 2 activation arcs, 1 repression arc, and the 2 self-arcs that are not found by our learning method.

4.4 Complexity Analysis

Theoretically, one way to approach the problem of learning a genetic network would be to look at every possible network and select the one with the best score. The cost of performing this action would be:

$$O(|\mathcal{I}| * score) \tag{4.1}$$

where $|\mathcal{I}|$ is the total number of possible networks, and *score* is the complexity to determine a score for each network. By assuming that $\mathcal{I}$ contains influence vectors, the maximum number of potential networks with $|S|$ species can be determined by assuming that network is a fully connected graph with three types of edges. These edges are activation, repression, or no influence. As a fully connected directed graph with self arcs has a total of $|S|^2$ edges, and each edge can be one of three different influences, this give a total of $3^{|S|^2}$ unique network topologies which changes Equation 4.1 to:

$$O(3^{|S|^2} * score). \tag{4.2}$$

By performing a local analysis for each species in S instead of looking at the global structure, the size of the search space can be further reduced to:

$$O(|S| * 3^{|S|} * score). \tag{4.3}$$

as each of the species in S has potentially $3^{|S|}$ ways in which it can be influenced. Figure 4.12 shows every influence vector of size three that would need to be looked at in the state space of Equation 4.3 for a particular $s \in S$.

The `CreateInfluenceVectorSet` algorithm looks at the connections between individual species and then selects either, activation, repression, or no influence for each individual influence. This further reduces the complexity to:

$$O(|S| * (|S| + 2^{|S|}) * score) = O(|S| * 2^{|S|} * score) \tag{4.4}$$

since each call to the `CreateInfluenceVectorSet` algorithm looks at $|S|$ influence vectors but reduces the remaining search space to $2^{|S|}$. Figure 4.13 shows a possible set of influence vectors that can potentially be explored after this reduction where the first influence is selected as a potential activator and the second two as potential repressors.

Each level of the lattice in Figure 4.13 can be thought of as choosing that many influences to change to either the activation or repression influences selected for that species. For example, at lattice level zero, there are no influences. At lattice level one, each influence vector has exactly one influence, and so on. The total number of influences can be expressed using the choose function, and Equation 4.4 can be expressed as:

$$O(|S| * \sum_{k=0}^{|S|} \binom{|S|}{k} * score) \tag{4.5}$$

where

$$\binom{|S|}{k} = \frac{|S|!}{k!(|S|-k)!}. \tag{4.6}$$

3			$\langle aaa \rangle$	$\langle aar \rangle$	$\langle ara \rangle$	$\langle arr \rangle$	$\langle raa \rangle$	$\langle rar \rangle$	$\langle rra \rangle$	$\langle rrr \rangle$		
2	$\langle aan \rangle$	$\langle arn \rangle$	$\langle ana \rangle$	$\langle anr \rangle$	$\langle ran \rangle$	$\langle rrn \rangle$	$\langle rna \rangle$	$\langle rnr \rangle$	$\langle naa \rangle$	$\langle nar \rangle$	$\langle nra \rangle$	$\langle nrr \rangle$
1				$\langle ann \rangle$	$\langle rnn \rangle$	$\langle nan \rangle$	$\langle nrn \rangle$	$\langle nna \rangle$	$\langle nnr \rangle$			
0						$\langle nnn \rangle$						

Figure 4.12. Every influence vector of size 3 organized by lattice level.

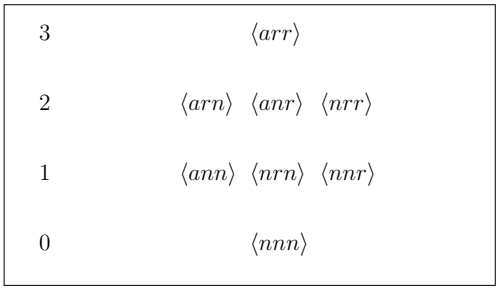


Figure 4.13. Every influence vector of size 3 limited by the `CreateInfluenceVectorSet` algorithm.

The T_j threshold, used in the `CombineInfluenceVectors` algorithm sets a limit as to the maximum number of influences an influence vector can contain, or how high up the lattice influence vectors are created. Using this threshold reduces Equation 4.5 to:

$$O(|S| * \sum_{k=0}^{T_j} \binom{|S|}{k} * score). \tag{4.7}$$

This is bound by:

$$O(|S| * (|S| + \ldots + |S|^{T_j}) * score) = O(|S| * |S|^{T_j} * score). \tag{4.8}$$

When $|S| >> T_j$ this threshold results in a significant improvement in complexity over Equation 4.4.

Each of the influence vectors must be competed against each other and the best one selected. This problem is broken into two parts in the `GeneNet` algorithm. These parts are competing two influence vectors where one is a subset of the other and competing unrelated influence vectors. The `CombineInfluenceVectors` algorithm competes related influence vectors by combining influence vectors starting at lattice level 0 and working up to lattice level T_j where if an influence vector has a better score than the subsets, it is declared the winner. By competing influence vectors in this manner, large portions of the lattice are pruned when combinations do not result in a better score than each individual influence vector separately. The `CompeteInfluenceVectors` algorithm competes unrelated influence vectors. This algorithm is computationally more costly as the G set contains more entries than it would when being combined in the `CombineInfluenceVectors`, as there are more species being controlled for during a competition of unrelated influence vectors.

The scoring function, as discussed in Section 3.3, is implemented such that its' complexity is limited to $O(|E| * |S|)$, or $O(|S| * |E| + |E|^2)$ if the better base probability function is used. Using the better base probability function the total complexity of the `GeneNet` algorithm becomes:

$$O(|S| * |S|^{T_j} * (|S| * |E| + |E|^2)). \tag{4.9}$$

CHAPTER 5

MODEL CREATION

Modeling is like building a house. If the end product were only a blueprint, people would say, "What is the value?"

–Jan Popkin

. . . modeling is like, you know, hard and stuff.

–Posted on a web page forum.

For many developers, modeling is like flossing. You know you probably should, but you can't stand the hassle.

–Bill Camarda

Once influence vectors are learned for a genetic network, they can be translated into genetic network models. The *Systems Biology Markup Language* (SBML) is an emerging standard for describing genetic network reactions [103]. It is a detailed reaction-based model, meaning that cellular events are described by their chemical reactions. As cellular functions are chemical in nature, this modeling language is capable of representing all the events that are likely to occur in a cell. Although SBML can represent most any event that is likely to occur in a cell, there are many events that are not directly involved in genetic network regulation, such as cell division. The models our method creates from influence vectors contain only those regulatory events that occur between species in the genetic network of interest.

Genetic networks represented by influence vectors and directed graphs abstract reactions and genetic species. The species in an influence vector are only a subset of the actual species that occur in a genetic network. These species are represented by a circle in the directed graph representation. Activation and repression reactions are abstracted to a single edge in the directed graph representation, or an 'a' or 'r' in an influence vector. This section describes how our method translates these abstractions into an

SBML model that more closely reflects the actual molecular reactions and species that occur in a genetic network. This section uses the directed graph representation of two species from the phage λ decision circuit, shown in Figure 5.1.

5.1 Defining SBML Species

In order to transform the high level network into an SBML model, our method must specify the species that are used in the SBML code. These species include those in the directed graph representation as well as several others that are needed for the other molecular pathways described in this section. As each species in the directed graph representation needs to be produced in some way, our method defines a promoter for each species. The names of the promoters correspond to their associated protein. The protein RNAP is also defined. A species is defined in SBML by giving it a species id, a name, an initial compartment, and an initial amount. The species id and name field have a similar function in that they are used to define unique species. Note that in the SBML code our method generates, the species id and name fields are the same in order to avoid confusion. The compartment field is used when describing where the protein is located in the cell. For instance, it can be inside or outside the cell. Our method only uses the 'default' category as our method currently does not model species moving across cellular membranes. The initial amount field is used to specify how many molecules of the protein are in the cell. Our method specifies an amount of 1.0 for promoters as there is usually only a single promoter found on the DNA strand for each gene. Our method also specifies a quantity of 30 RNAP molecules [7]. All other protein amounts are set to 0.0 except in mutational cases where a gene is mutated to a high amount.

Figure 5.2 shows the graphical representation of the species created from the high

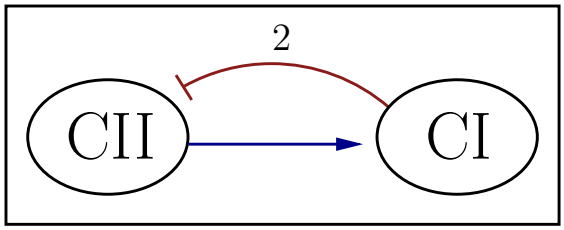


Figure 5.1. Graphical representation of the connections between two of the species in the phage λ decision circuit.

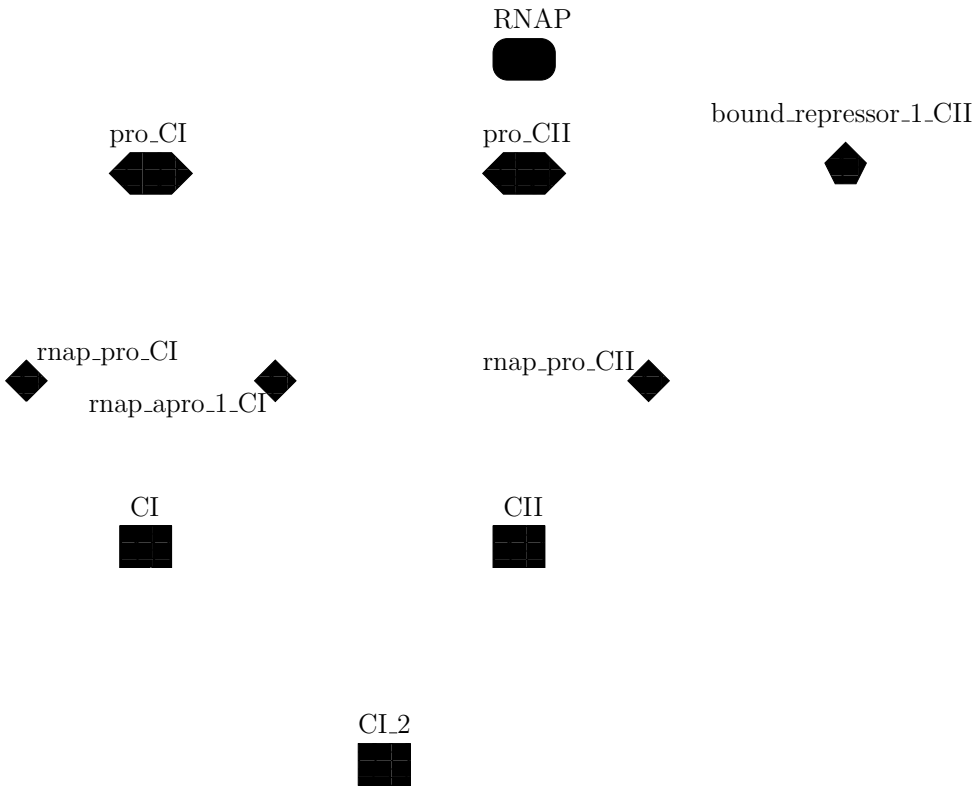


Figure 5.2. Graphical representation of the molecules for the phage λ decision circuit.

level model shown in Figure 5.1 when modeling the phage λ decision circuit. The SBML code that describes these species is shown in Figure 5.3. Notice that RNAP and the species CI and CII appear in the model, as well as their promoters pro_CI and pro_CII. The other species in the model are described when describing the reactions they are a part of. These reaction are governed by rate laws. The parameters for these rate laws are shown in Table 5.1 and are also discussed when describing the reactions they are a part of.

5.2 Adding Reactions

Proteins degrade over time. This process is typically defined as a simple reaction in the SBML code that takes a molecule of the protein and produces nothing. The reaction is not reversible, meaning that once it occurs, it cannot be undone. As actual degradation

```
<listOfSpecies>
  <species id = "CI"  name = "CI" compartment = "default"
        initialAmount = "0.0"/>
  <species id = "CII"  name = "CII" compartment = "default"
        initialAmount = "0.0"/>
  <species id = "rnap"  name = "RNAP" compartment = "default"
        initialAmount = "30"/>
  <species id = "pro_CI"  name = "pro_CI" compartment = "default"
        initialAmount = "1.0"/>
  <species id = "rnap_pro_CI"  name = "rnap_pro_CI" compartment =
        "default" initialAmount = "0.0"/>
  <species id = "rnap_apro_1_CI"  name = "rnap_apro_1_CI" compartment
         = "default" initialAmount = "0.0"/>
  <species id = "pro_CII"  name = "pro_CII" compartment = "default"
        initialAmount = "1.0"/>
  <species id = "rnap_pro_CII"  name = "rnap_pro_CII" compartment =
        "default" initialAmount = "0.0"/>
  <species id = "bound_rep_1_CII"  name = "bound_rep_1_CII" compartment
        = "default" initialAmount = "0.0"/>
  <species id = "CI_2"  name = "CI_2" compartment = "default"
        initialAmount = "0.0"/>
</listOfSpecies>
```

Figure 5.3. SBML representing some species created from the high level description of the phage λ decision circuit.

Table 5.1. Parameters used in the generation of SBML code.

Parameter	Value	Note
initialAmount	1.0	Promoter
initialAmount	0.0	Species
initialAmount	30.0	RNAP
koc	0.01	Basal
koc	0.1	Main production pathways
k_deg	0.0003	Degradation
k_deg	0.9	Uninfluenced
kf_un	0.9	Uninfluenced
kf_d	20	Dimer
kf_rep	0.8	Repression
kf	0.033	RNAP binding
kf	0.00033	Activated RNAP binding
kr	1.0	All kr cases

breaks a molecule apart, this matches what occurs in nature. The most important part of a degradation reaction is the value of the kinetic rate parameter describing how fast a protein degrades. The default for this parameter is 0.0003. This number represents a half-life of about 40 minutes. If a gene is not influenced by any species in the network, our method tries to model a randomized walk, and uses a rate of 0.9 in this case. As with all the information that our method automatically constructs for the high-level model, if a researcher has more detailed information they can edit the SBML code before simulation to produce a more accurate representation.

Figure 5.4 shows the degradation reaction for the proteins CI and CII in the phage λ decision circuit. The SBML code for the degradation of protein CI is shown in Figure 5.5. Protein CII would have a similar degradation reaction in the SBML code.

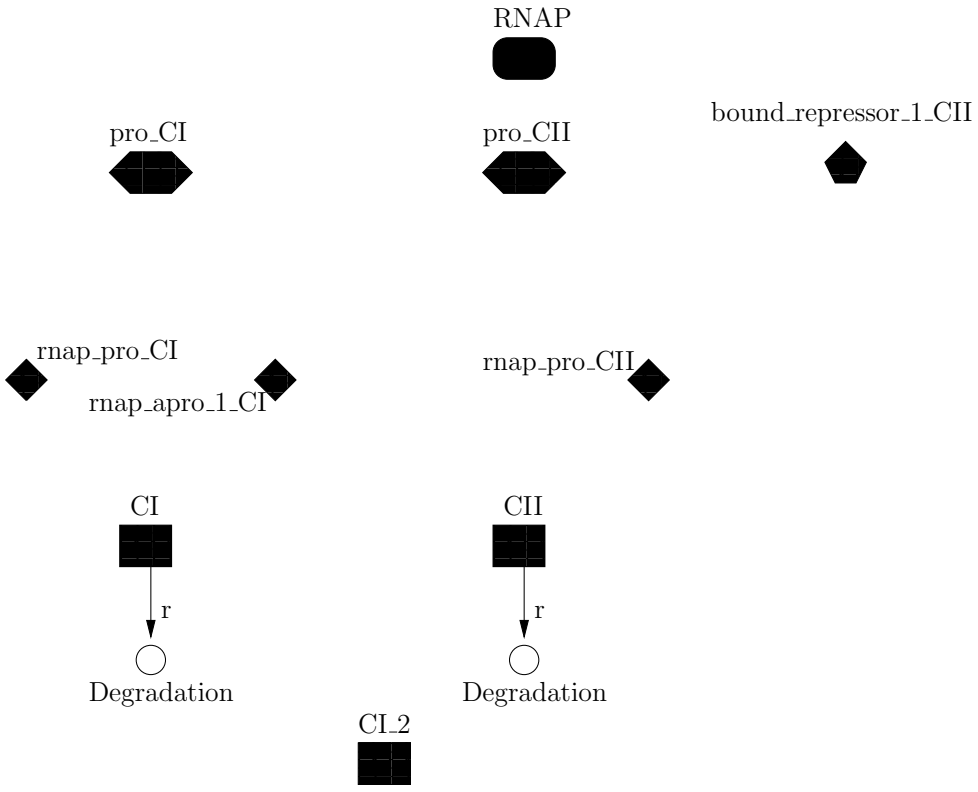


Figure 5.4. Graphical representation of the degradation reactions for CI and CII of the phage λ decision circuit.

```
<reaction id = "degradation_CI" reversible="false" >
  <listOfReactants>
    <speciesReference species = "CI" stoichiometry = "1"/>
  </listOfReactants>
  <kineticLaw>
    <math xmlns="http://www.w3.org/1998/Math/MathML">
      <apply>
        <times/>
        <ci>k_deg</ci>
        <ci>CI</ci>
      </apply>
    </math>
    <listOfParameters>
      <parameter id = "k_deg" name = "k_deg" value = "0.0003"/>
    </listOfParameters>
  </kineticLaw>
</reaction>
```

Figure 5.5. SBML code describing the degradation of the species CI.

The production of a species is governed by two processes. The first process involves RNAP binding to the promoter by itself, and the second process involves another species binding to the promoter first and providing RNAP with an easier chance of binding. With both of these processes, after RNAP is bound to the promoter, through transcription and translation, the species is produced.

Figure 5.6 shows a graphical representation of the first of the two processes described above for the species CI and CII of the phage λ decision circuit. The binding of RNAP to the promoter site is shown by the two top reactions, or circles in the figure, with RNAP and the promoter for the gene which creates a third molecule. For example, RNAP binds with CI to form a molecule of rnap_pro_CI. This naming convention means that the new species is representing the fact that RNAP is bound to the promoter. The SBML code for this reaction is shown in Figure 5.7. This reaction is reversible since the RNAP molecule falls off of the promoter, as it would in a true cellular reaction, leaving a molecule of pro_CI and a molecule of RNAP. The default equilibrium constant is 0.033 for this reaction. If a gene is not influenced by any species in the network, our method tries to model a randomized walk for that species and uses an equilibrium constant of 0.9 in this case.

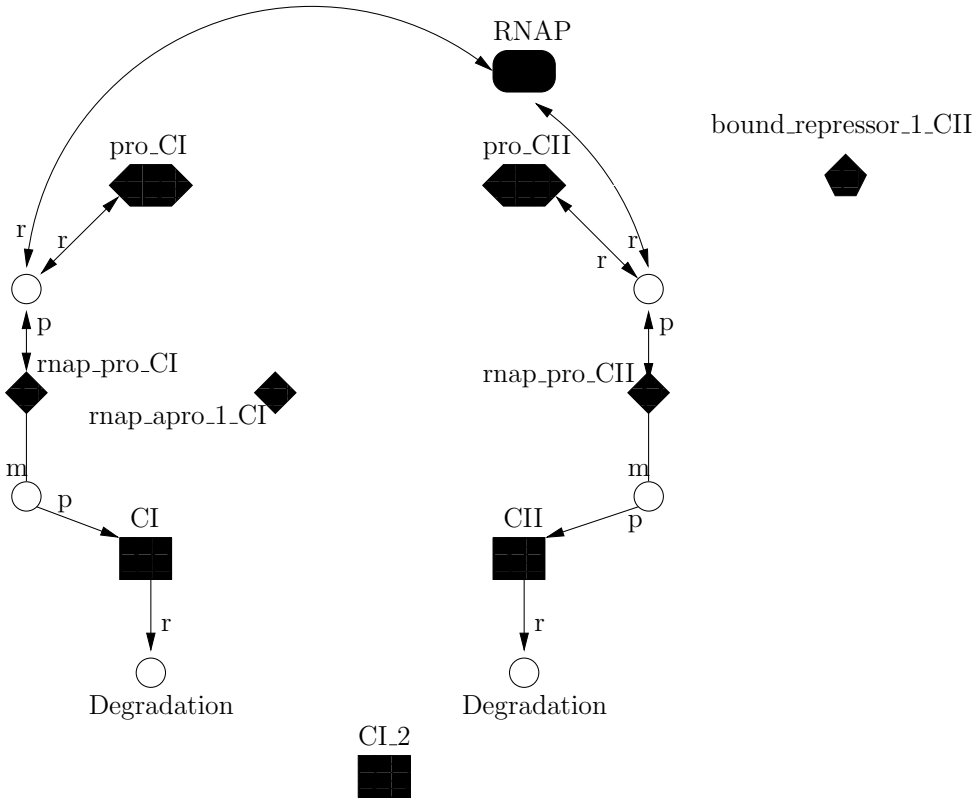


Figure 5.6. Graphical representation of the species production pathway involving only RNAP binding to the promoter to produce a species for species CI and CII of the phage λ decision circuit.

```
<reaction id = "R_rnap_CI" reversible="true">
  <listOfReactants>
    <speciesReference species = "rnap" stoichiometry = "1"/>
    <speciesReference species = "pro_CI" stoichiometry = "1"/>
  </listOfReactants>
  <listOfProducts>
    <speciesReference species = "rnap_pro_CI" stoichiometry = "1"/>
  </listOfProducts>
  <kineticLaw>
    <math xmlns="http://www.w3.org/1998/Math/MathML">
      <apply>
        <minus/>
        <apply>
          <times/>
          <ci>kf</ci>
          <ci>pro_CI</ci>
          <ci>rnap</ci>
        </apply>
        <apply>
        <times/>
        <ci>kr</ci>
        <ci>rnap_pro_CI</ci>
        </apply>
      </apply>
    </math>
    <listOfParameters>
      <parameter id = "kf" name = "kf" value = "0.033"/>
      <parameter id = "kr" name = "kr" value = "1.0"/>
    </listOfParameters>
  </kineticLaw>
</reaction>
```

Figure 5.7. SBML code describing the binding of RNAP to the promoter for CI.

The second reaction in the production of a species from RNAP binding to a promoter is represented as the reaction that produces CI and the reaction that produces CII in Figure 5.6. These reactions represent transcription and translation and are modeled as a single reaction. The reaction for the production of species CI is represented by the arcs from rnap_pro_CI to CI in Figure 5.6. This reaction creates a molecule of CI and a molecule of rnap_pro_CI, since rnap_pro_CI is not otherwise consumed by the reaction. The SBML code for this reaction is shown in Figure 5.8. Notice that this reaction is not reversible, meaning that once a molecule of CI is created, it is not somehow destroyed by rebinding with RNAP and the promoter. This mimics what would occur in an actual cellular reaction. As CI is activated by CII, the rate of this reaction is referred to as the *basal* rate and is typically lower than the activated rate. The value of 0.01 is used for this reactions rate. If a species is not activated, then the value of 0.1 is used for this rate. For example, as CII is not activated in this example, it would have a rate of 0.1 for this

```
<reaction id = "R_rnap_pro_CI" reversible="false">
  <listOfReactants>
    <speciesReference species = "rnap_pro_CI" stoichiometry = "1"/>
  </listOfReactants>
  <listOfProducts>
    <speciesReference species = "CI" stoichiometry = "1"/>
    <speciesReference species = "rnap_pro_CI" stoichiometry = "1"/>
  </listOfProducts>
  <kineticLaw>
    <math xmlns="http://www.w3.org/1998/Math/MathML">
      <apply>
        <times/>
        <ci>koc</ci>
        <ci>rnap_pro_CI</ci>
      </apply>
    </math>
    <listOfParameters>
      <parameter id = "koc" name = "koc" value = "0.01"/>
    </listOfParameters>
  </kineticLaw>
</reaction>
```

Figure 5.8. SBML code describing the production of species CI from species rnap_pro_CI.

reaction.

The reactions involving activation, or a species binding to an operator to allow RNAP a better chance at binding, is also modeled. This pathway is only created for species that have activators. Figure 5.9 shows the reactions that are involved in CII activating CI in the phage λ decision circuit. The first reaction is where CII and RNAP bind to the promoter pro_CI to create a complex rnap_apro_1_CI. The SBML code for this is shown in Figure 5.10. Notice that the reaction is reversible. The forward rate used for this reaction is 0.00033.

Several reactions that would occur in a cell are still abstracted in our model. Although our SBML code shows that both the CII and RNAP bind to the promoter at the same time, this is not normally the case in actual cellular events. The species CII would usually

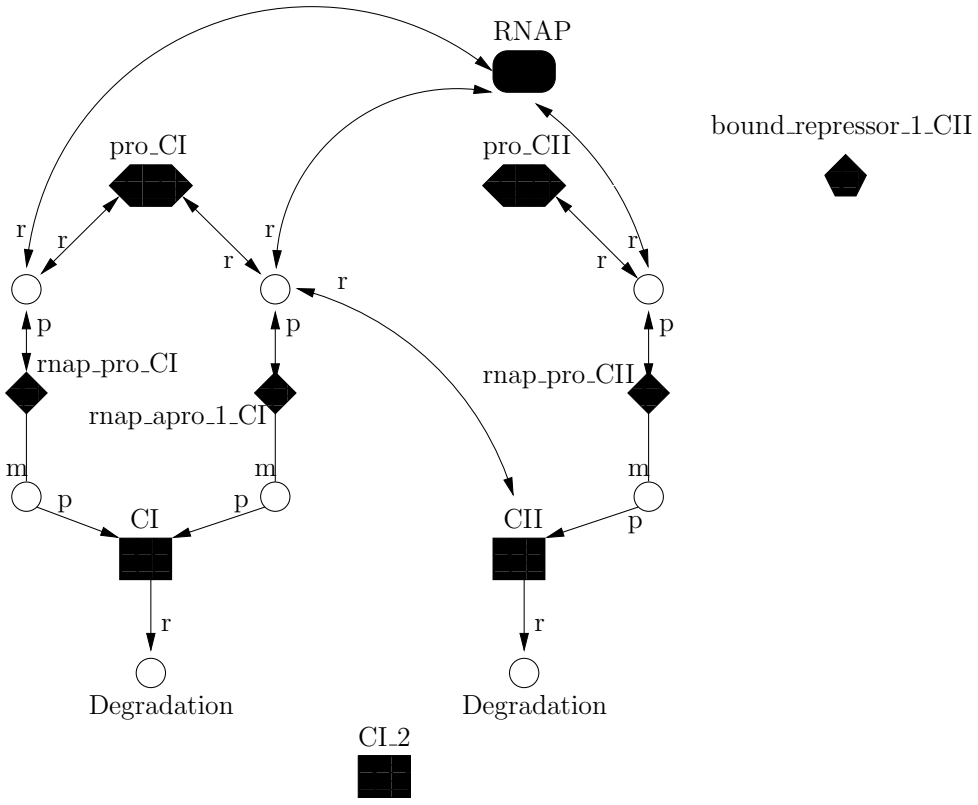


Figure 5.9. Graphical representation of the activation pathway for species CI of the phage λ decision circuit.

```
<reaction id = "R_1_CII_activates_CI" reversible="true">
  <listOfReactants>
    <speciesReference species = "rnap" stoichiometry = "1"/>
    <speciesReference species = "pro_CI" stoichiometry = "1"/>
    <speciesReference species = "CII" stoichiometry = "1"/>
  </listOfReactants>
  <listOfProducts>
    <speciesReference species = "rnap_apro_1_CI" stoichiometry = "1"/>
  </listOfProducts>
  <kineticLaw>
    <math xmlns="http://www.w3.org/1998/Math/MathML">
      <apply>
        <minus/>
          <apply>
            <times/>
            <ci>kf</ci>
            <ci>rnap</ci>
            <ci>pro_CI</ci>
            <ci>CII</ci>
          </apply>
        <apply>
          <times/>
          <ci>kr</ci>
          <ci>rnap_apro_1_CI</ci>
        </apply>
      </apply>
    </math>
    <listOfParameters>
      <parameter id = "kf" name = "kf" value = "0.00033"/>
      <parameter id = "kr" name = "kr" value = "1.0"/>
    </listOfParameters>
  </kineticLaw>
</reaction>
```

Figure 5.10. SBML code for the first reaction in the activation of species CI by species CII.

bind first, allowing RNAP a greater chance to bind. The reaction described replaces these two reactions with the abstracted one.

The second reaction in the activation pathway is the creation of a molecule of CI from the complex rnap_apro_1_CI. The SBML code for this is shown in Figure 5.11. This reaction has a rate of 0.1, and is not reversible. This reaction is similar to the production reaction described in Figure 5.8.

The process of two molecules coming together to form a dimer is also modeled. In the phage λ decision circuit example, the species CI can form a dimer, or a molecule of CI_2, as is shown in Figure 5.12. The SBML code for this reaction is shown in Figure 5.13. This reaction has a rate of 20 and is reversible, allowing the CI dimer to break apart into two molecules of CI.

Another pathway that needs to be created from the high level representation is a repression pathway. This pathway is only created for species that are repressed.

```
<reaction id = "R_1_activation_part2_CII_to_CI" reversible="false">
  <listOfReactants>
    <speciesReference species = "rnap_apro_1_CI" stoichiometry = "1"/>
  </listOfReactants>
  <listOfProducts>
    <speciesReference species = "CI" stoichiometry = "1"/>
    <speciesReference species = "rnap_apro_1_CI" stoichiometry = "1"/>
  </listOfProducts>
  <kineticLaw>
    <math xmlns="http://www.w3.org/1998/Math/MathML">
      <apply>
        <times/>
        <ci>koc</ci>
        <ci>rnap_apro_1_CI</ci>
      </apply>
    </math>
    <listOfParameters>
      <parameter id = "koc" name = "koc" value = "0.1"/>
    </listOfParameters>
  </kineticLaw>
</reaction>
```

Figure 5.11. SBML code for the second reaction in the activation of species CI by species CII.

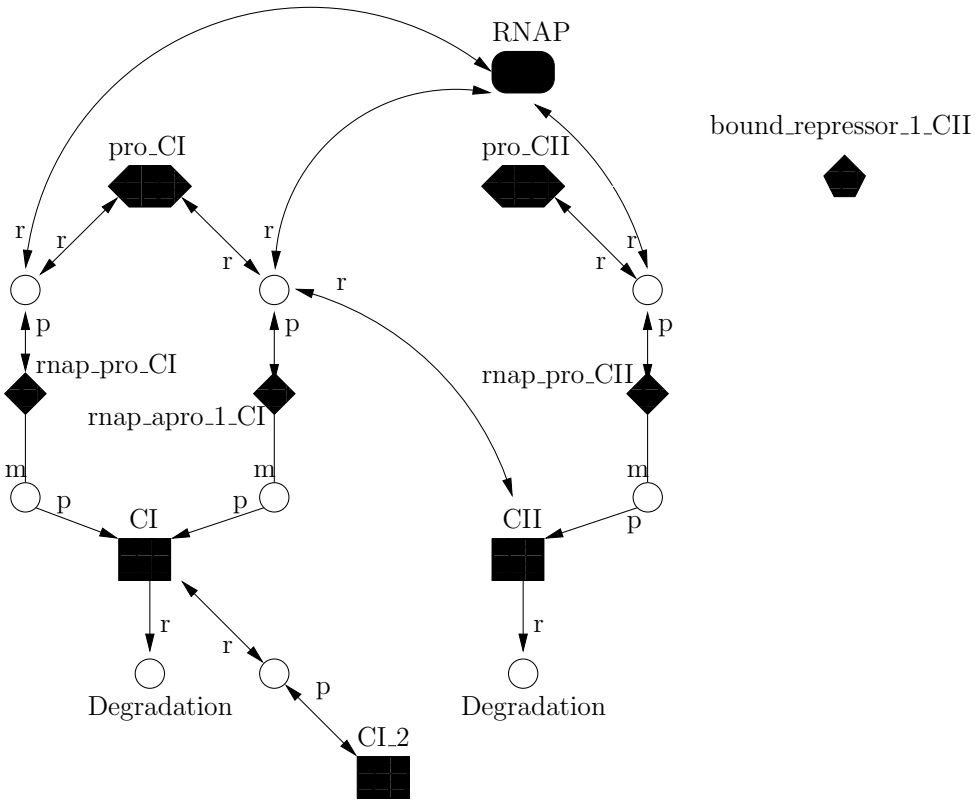


Figure 5.12. Graphical representation of the dimerization reaction of CI of the phage λ decision circuit.

```
<reaction id = "dimerization_CI_2" reversible="true">
  <listOfReactants>
    <speciesReference species = "CI" stoichiometry = "2"/>
  </listOfReactants>
  <listOfProducts>
    <speciesReference species = "CI_2" stoichiometry = "1"/>
  </listOfProducts>
  <kineticLaw>
    <math xmlns="http://www.w3.org/1998/Math/MathML">
      <apply>
        <minus/>
        <apply>
          <times/>
            <ci>kf_d</ci>
            <ci>CI</ci>
            <ci>CI</ci>
        </apply>
        <apply>
          <times/>
          <ci>kr</ci>
          <ci>CI_2</ci>
        </apply>
      </apply>
    </math>
    <listOfParameters>
      <parameter id = "kf_d" name = "kf_d" value = "20"/>
      <parameter id = "kr" name = "kr" value = "1.0"/>
    </listOfParameters>
  </kineticLaw>
</reaction>
```

Figure 5.13. SBML code describing the dimerization reaction of species CI.

Figure 5.14 shows the reaction that is involved in the repression of species CII by species CI dimer in the phage λ decision circuit. The reaction that occurs is that CI_2 and the pro_CII bind together to form a molecule of bound_rep_1_CII. In effect, this removes the pro_N species from the system, as there is only 1 molecule of it, so that the pathways involving RNAP binding to the promoter do not occur. The SBML code for this is shown in Figure 5.15. Notice that the reaction is reversible, meaning that the molecule that represses can fall off as it would in an actual cell. The forward rate for this reaction is 0.8. Note that Figure 5.14 is the complete genetic network for our example.

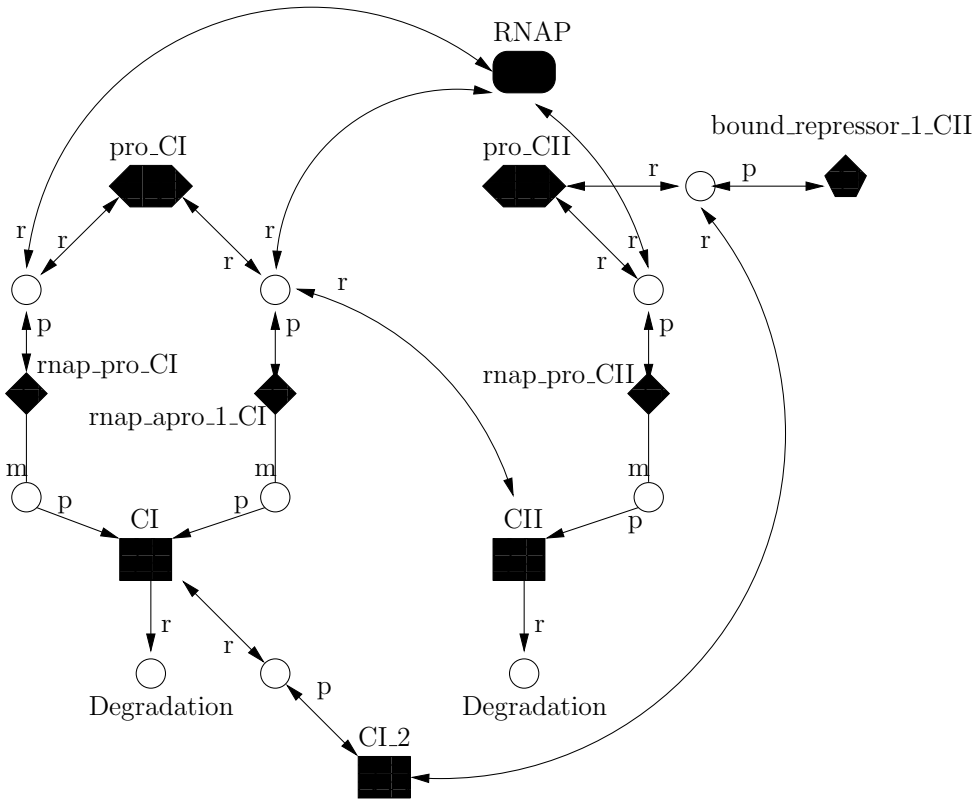


Figure 5.14. Graphical representation of repression reaction of the CI dimer on the CII promoter of the phage λ decision circuit.

```
<reaction id = "R_1_CI_2_represses_CII" reversible="true">
    <listOfReactants>
      <speciesReference species = "pro_CII" stoichiometry = "1"/>
      <speciesReference species = "CI_2" stoichiometry = "1"/>
    </listOfReactants>
    <listOfProducts>
      <speciesReference species="bound_rep_1_CII" stoichiometry="1"/>
    </listOfProducts>
    <kineticLaw>
    <math xmlns="http://www.w3.org/1998/Math/MathML">
      <apply>
        <minus/>
        <apply>
          <times/>
          <ci>kf_rep</ci>
          <ci>pro_CII</ci>
          <ci>CI_2</ci>
        </apply>
        <apply>
          <times/>
          <ci>kr</ci>
          <ci>bound_rep_1_CII</ci>
        </apply>
      </apply>
    </math>
    <listOfParameters>
      <parameter id = "kf_rep" name = "kf_rep" value = "0.8"/>
      <parameter id = "kr" name = "kr" value = "1.0"/>
    </listOfParameters>
    </kineticLaw>
</reaction>
```

Figure 5.15. SBML code for the repression of species CII by species CI dimer.

5.3 SBML Code For Mutations

There are two basic types of mutations. These are mutations where a species is set artificially high, or artificially low. Our method handles each case separately.

When a species is mutated low, it should either not be present or behave as if it is not present. Setting an initial concentration of 0 for the species in the SBML code is one way in which to assure that in simulation there are no molecules of that species at the start of a simulation. As there should also be no molecules of that species generated

during the simulation, the reactions that produce that species should be removed. These reactions include the main RNAP binding to the promoter reaction as well as the cases where the gene is activated by other genes. Once this is done there should be no molecules of that species produced during simulation. Therefore, the reactions in which the species is involved do not necessarily need to be removed from the SBML code as they should never occur.

When a species is mutated high it should behave as if it is always present, and in high concentrations. The initial amount for these species is set to 100. This way, the protein produced from the gene is present during the start of a simulation. Our method also removes the degradation reaction for this species so that the protein does not degrade during the simulation. Although the species would typically degrade in actuality, by removing the degradation reaction our method can better simulate the actual behavior that would occur with this type of experiment. This could mimic the experiment where a cell is in an environment that provides a continuous supply of the protein. This could also have been implemented with another reaction that continuously produces the protein.

CHAPTER 6

EXPERIMENTAL SUGGESTION

This chapter describes how the `GeneNet` algorithm can be used to determine what experiments a researcher should perform to maximize the effectiveness of learning genetic networks using the `GeneNet` algorithm. This chapter first describes a formalism for describing influence vectors that are discarded by the `GeneNet` algorithm. The chapter then describes all the places in the `GeneNet` algorithm where potentially correct influence vectors are pruned from the search space due to insufficient data and suggests changes to the various algorithms to record, or store, these influence vectors. This chapter then describes how these stored influence vectors can be used to suggest experiments to the researcher to reduce the amount of experiments needed to learn a genetic network.

6.1 Contenders

The `GeneNet` algorithm has been designed to choose the best influence vector for each species of interest. There are generally several other influence vectors that are discarded in favor of a higher scoring influence vector during the execution of the `GeneNet` algorithm. These influence vectors are considered to be *contenders*, and are stored using the `AddContender` function described in Section 6.3.

The `AddContender` function takes a tuple of the form:

$$\langle\langle i_c, \alpha_c\rangle, case, \langle i_r, \alpha_r\rangle\rangle$$

where i_c is a contender influence vector, α_c is its' score, $case \in \{1, 2, 3, 4, 5, 6, 7\}$, is the reason it is discarded, i_r is the influence vector causing it to be discarded, and α_r is its' score. The cases are described in greater detail in Section 6.2.

6.2 Selecting Contenders

In order to suggest experiments that would help in learning a genetic network, a limited number of influence vectors that may be correct must be found. As the majority

of the influence vectors discarded during the execution of the `GeneNet` algorithm represent incorrect influence vectors, only the best should be used when trying to suggest experiments. This section describes seven cases, shown in Table 6.1, in which the `GeneNet` algorithm discards influence vectors. A subset of these are retained as contenders and used for suggesting experiments. This section explains how each `GeneNet` algorithm is modified to select contenders using the thresholds shown in Table 6.2.

Table 6.1. The cases in the GeneNet algorithm where influence vectors are discarded.

Case	Algorithm	Reason
1	`CreateInfluenceVectorSet`	Not better than background knowledge
2	`CombineInfluenceVectors`	Influence vector scores not close enough
3	`CombineInfluenceVectors`	Merged influence vector's score too low
4	`RemoveSubsets`	Influence vector merged
5	`Filter`	Not better than background knowledge
6	`Filter`	Background knowledge filtered
7	`CompeteInfluenceVectors`	Lost a competition

Table 6.2. The P thresholds used in the GeneNet algorithm.

Threshold	Default Value	Note
P_i	0.3	A threshold used for Case 1 to compare against background knowledge.
P_m	0.3	A threshold used for Case 2 to compare the score of two influence vectors.
P_a	0.1	A threshold used for Case 3 to determine if the merged influence vector should be retained.
P_b	0.05	A threshold used for Case 4 to determine if subset influence vectors should be retained.
P_f	0.1	A threshold used for Case 5 to retain influence vectors that almost score better than background knowledge.
P_q	0.1	A threshold used for Case 6 to retain background knowledge if an influence vector scores higher than it.
P_l	0.1	A threshold used during competition to retain a losing influence vector.

Case 1 in Table 6.1 occurs in the `CreateInfluenceVectorSet` algorithm. This algorithm creates an initial set of influence vectors from the background knowledge where at most one unknown influence is set to 'a' or 'r'. If the score of this new influence vector is not greater than the background knowledge score, than it is discarded. Figure 6.1 shows the updated `CreateInfluenceVectorSet` algorithm that calls the `AddContender` function on those species that have scores within P_i of the background knowledge. As the bounds may be relaxed during execution of the algorithm, contenders are first added into the set C, which is set to the empty set at the beginning of every round. Using the phage λ decision circuit example, the learned influence vector for species CIII is

```
CreateInfluenceVectorSet(Species s, Species Set S, Compressed Data
        CD, Influences I, Levels θ, Number of Bins n, Thresholds T)
  I := ∅
  do
    α := Score(s, I(s), {s}, S, n, CD, T_a, T_r, T_i, T_s)
    foreach s' such that I(s') = ?
      i := I(s)
      i(s') := a
      C := ∅
      foreach s'' ∈ (S − {s'})
        if (i(s'') =?) then i(s'') := n
      Score := Score(s, i, {s}, S, n, CD, T_a, T_r, T_i, T_s)
      if (Score ≥ α) then
        I := I ∪ i
      else if (Score ≥ α − P_i) then
        C := C ∪ {⟨⟨i, Score⟩, 1, ⟨I(s), α⟩⟩}
      else
        i(s') := r
        Score := Score(s, i, {s}, S, n, CD, T_a, T_r, T_i, T_s)
        if (Score ≥ α) then
          I := I ∪ i
        else if (Score ≥ α − P_i) then
          C := C ∪ {⟨⟨i, Score⟩, 1, ⟨I(s), α⟩⟩}
    (T_a, T_r) := RelaxThresholds(T_a, T_r, T_t)
  while |I| < T_n
  foreach ⟨c, type, R⟩ ∈ C
    AddContender(c, type, R)
  return I
```

Figure 6.1. The `CreateInfluenceVectorSet` algorithm updated to create contenders.

$\langle r, n, n, r, n \rangle$. The influence vectors under Case 1 in Table 6.3 are discarded in the `CreateInfluenceVectorSet` algorithm, as their score did not pass the background knowledge score of 0.5. For P_i of 0.3, the influence vector $\langle n, r, n, n, n \rangle$ would be retained as a contender.

Cases 2 and 3 in Table 6.1 occur in the `CombineInfluenceVectors` algorithm. This algorithm merges two influence vectors into a single influence vectors containing the influences of both. Case 2 discards the merged influence vector, i', if the scores of the influence vectors that are in the subset of i' are not within the T_m threshold of each other. However, if the scores of these influence vectors are within P_m of each other, then the merged influence vector is added to the contenders. Case 3 discards i' if its' score is not greater than the score of each individual influence vector in its' subset. However if the score of i' when adding to P_a is better than each of its subsets' scores, then i' is added to the contenders. Figure 6.2 shows the updated `CombineInfluenceVectors` algorithm that includes the function calls to the `AddContender` function for Cases 2 and 3. In the phage λ decision circuit example, there are no examples for Case 2. The influence vectors in Table 6.3 under Case 3 are discarded because the scores of the influence vectors are not better than the scores of the influence vectors merged to create them. For a P_a of 0.05, both influence vectors are considered contenders.

Case 4 in Table 6.1 occurs in the `RemoveSubsets` algorithm. This algorithm removes influence vectors that have been merged into a larger influence vector. However, if the

Table 6.3. Influence vectors considered for the species CIII during execution of the `GeneNet` algorithm where the species ordering is $\langle CI, CII, CIII, Cro, N \rangle$.

$\langle i_c, \alpha_c \rangle$	Case	$\langle i_r, \alpha_r \rangle$	Contender
$\langle \langle a, n, n, n, n \rangle$, -1.0$\rangle$	1	$\langle \langle n, n, n, n, n \rangle, 0.5 \rangle$	no
$\langle \langle n, a, n, n, n \rangle$, 0.2$\rangle$	1	$\langle \langle n, n, n, n, n \rangle, 0.5 \rangle$	yes
$\langle \langle n, r, n, n, n \rangle$, -0.2$\rangle$	1	$\langle \langle n, n, n, n, n \rangle, 0.5 \rangle$	no
$\langle \langle n, n, n, a, n \rangle$, -1.0$\rangle$	1	$\langle \langle n, n, n, n, n \rangle, 0.5 \rangle$	no
$\langle \langle n, n, n, n, a \rangle$, -0.833$\rangle$	1	$\langle \langle n, n, n, n, n \rangle, 0.5 \rangle$	no
$\langle \langle r, n, n, n, r \rangle$, 0.875$\rangle$	3	$\langle \langle n, n, n, n, r \rangle, 1.0 \rangle$	yes
$\langle \langle n, n, n, r, r \rangle$, 0.958$\rangle$	3	$\langle \langle n, n, n, r, r \rangle, 1.0 \rangle$	yes
$\langle \langle r, n, n, r, r \rangle$, 0.897$\rangle$	3	$\langle \langle n, n, n, r, r \rangle, 1.0 \rangle$	yes
$\langle \langle r, n, n, n, n \rangle$, 1.0$\rangle$	4	$\langle \langle r, n, n, r, n \rangle, 1.0 \rangle$	yes
$\langle \langle n, n, n, r, n \rangle$, 1.0$\rangle$	4	$\langle \langle n, n, n, r, r \rangle, 1.0 \rangle$	yes
$\langle \langle n, n, n, n, r \rangle$, 0.27$\rangle$	7	$\langle \langle r, n, n, r, n \rangle, 0.99 \rangle$	no

```
CombineInfluenceVectors(Species s, IV Set I, Species Set S,
        Compressed Data CD, Influences 𝓘, Levels θ,
        Number of Bins n, Thresholds T)
  for k = |𝓘(s)| + 2 to T_j
    I' = {i ∪ i' | i ∈ I ∧ i' ∈ I ∧ |i ∪ i'| = k}
    foreach i' ∈ I'
      α_min := ∞
      α_max := −∞
      foreach i ∈ I such that i ⊆ i'
        α := Score(s, i, {s}, S, n, CD, T_a, T_r, T_i, T_s)
        if α < α_min then
          i_min := i
          α_min := α
        if α > α_max then
          i_max := i
          α_max := α
      if α_min + T_m < α_max then
        I' := I' − {i'}
        if α' + P_m ≥ α_max then
          α' := Score(s, i', {s}, S, n, CD, T_a, T_r, T_i, T_s)
          AddContender(⟨i', α'⟩, 2, ⟨i_min, α_min⟩)
      else
        α' := Score(s, i', {s}, S, n, CD, T_a, T_r, T_i, T_s)
        if α' < α_max then
          I' := I' − {i'}
          if α' + P_a ≥ α_max then
            AddContender(⟨i', α'⟩, 3, ⟨i_max, α_max⟩)
    I := RemoveSubsets(s, I ∪ I', S, CD, n, T)
  I := Filter(s, 𝓘(s), I, S, CD, n, T)
  return I
```

Figure 6.2. The `CombineInfluenceVectors` algorithm updated to create contenders.

score of the larger influence vector minus P_b is not greater than the subset, the subset is added to the contenders. Figure 6.3 shows the updated `RemoveSubsets` algorithm that includes the call to the `AddContender` function for Case 4. In the phage λ decision circuit example, the influence vectors in Table 6.3 under Case 4 are discarded because they are merged into a larger influence vector. For a P_b greater than 0, they would each be considered contenders.

Cases 5 and 6 in Table 6.1 occur in the `Filter` function. For Case 5, the function

```
RemoveSubsets(Species s, IV Set I, Species Set S,
      Compressed Data CD, Number of Bins n, Thresholds T_a, T_r, T_i, T_s)
  foreach i ∈ I
    foreach i' ∈ I − {i}
      if i ⊆ i' then
        I := I − {i}
        α := Score(s, i, {s}, S, n, CD, T_a, T_r, T_i, T_s)
        α' := Score(s, i', {s}, S, n, CD, T_a, T_r, T_i, T_s)
        if α' − P_b < α then
          AddContender(⟨i, α⟩, 4, ⟨i', α'⟩)
        break
  return I
```

Figure 6.3. The `RemoveSubsets` function.

removes those influence vectors that have a lower score than the background knowledge. These influence vectors are considered contenders if their score is greater than the background knowledge minus P_f. For Case 6, it is checked whether the maximum score of the influence vectors that beat background knowledge is within P_q of the background knowledge score. In this case, background knowledge is added into the set. Figure 6.4 shows the updated `Filter` function that includes the call to the `AddContender` function for these two cases.

Case 7 in Table 6.1 occurs in the `CompeteInfluenceVectors` algorithm. This algorithm competes influence vectors against each other and removes the losing influence vector. If the score of the losing influence vector plus P_l is greater than the winners score, the losing influence vector is considered a contender. Figure 6.5 shows the updated `CompeteInfluenceVectors` algorithm that includes the call to the `AddContender` function for Case 7. In the phage λ decision circuit example, the influence vector in Table 6.3 under Case 7 is discarded because it lost in the competition performed in the `CompeteInfluenceVectors` algorithm to another influence vector. For a P_l equal to 0.1, this influence vector is not considered a contender.

6.3 Suggesting Experiments

When learning genetic networks, there are often several possible competing theories, called contenders throughout this chapter, as to what the connections should be in the genetic networks. It would be helpful to an experimenter to use this information to

```
Filter(Species s, IV i,  IV Set I, Species Set S,
       Compressed Data CD, Number of Bins n, Thresholds T)
  α := Score(s, i, {s}, S, n, CD, T_a, T_r, T_i, T_s)
  α_max = −∞
  foreach i' ∈ I
    α' := Score(s, i', {s}, S, n, CD, T_a, T_r, T_i, T_s)
    if α' > α then
      I' := I' ∪ {i'}
      if α' > α_max then
        i_max := i'
        α_max := α'
    else if α' > α − P_f then
      AddContender(⟨i', α'⟩, 5, ⟨i_max, α_max⟩)
  if |A| > 0 ∧ α_max < α + P_q then
    AddContender(⟨i, α⟩, 6, ⟨i_max, α_max⟩)
  return I'
```

Figure 6.4. The `Filter` function updated to create contenders.

```
CompeteInfluenceVectors(Specie s, IV Set I, Species S,
       Compressed Data CD, Levels θ, Number of Bins n, Thresholds T)
  while |I| > 1
    (i, i') = selectPair(I)
    (T'_a, T'_r) := (T_a, T_r)
    do
      α := Score(s, i, (Par(i') − Par(i)) ∪ {s}, S, n, CD, T'_a, T'_r, T_i, T_s)
      α' := Score(s, i', (Par(i) − Par(i')) ∪ {s}, S, n, CD, T'_a, T'_r, T_i, T_s)
      (T'_a, T'_r) := RelaxThresholds(T'_a, T'_r, T_t)
    while (α = α' ∧ T'_r > 0 ∧ T'_a < 5)
    if (α > α') ∨ (α = α' ∧ |i| < |i'|) then
      I := I − {i'}
      if (α < α' + P_l) then
        AddContender(⟨i', α'⟩, 7, ⟨i, α⟩)
    else
      I := I − {i}
      if (α + P_l > α') then
        AddContender(⟨i, α⟩, 7, ⟨i', α'⟩)
  i := select(I)
  return i
```

Figure 6.5. The `CompeteInfluenceVectors` algorithm updated to create contenders.

learn which parts of the network are the least clear, so that extra experiments can be performed on these parts. This section describes how the seven cases in Table 6.1 can be used to suggest experiments using the `AddContender` function shown in Figure 6.6. The `AddContender` function uses the `Suggest` map, which maps a set of species to the associated integer value containing how often the set of species is suggested as a potential mutation experiment.

Case 1 and Case 5 in Table 6.1 describes how influence vectors with a score close, but not greater than, the background knowledge are filtered from competition and considered contenders. As this influence vector is filtered due to background knowledge, then mutating the differing species is suggested by the `AddContender` function. Consider the example shown in Figure 6.7 where the contender is species A represses species C and the reason is an influence vector with no connections between these species. Consider a mutational experiment where species A is mutated low using a gene knockout experiment. In this experiment, there would be more data points added into the case where species A is low. If the real network is $\langle r, n, n\rangle$ the probability of species C increasing in expression in this base case would increase, as there would be no species A to repress it, thus adding support for this influence vector to perhaps increase its score above that of the background knowledge.

Case 2 in Table 6.1 involves two influence vectors being unable to merge due to their individual scores not being within T_m of each other. In this case, the `AddContender` function suggests an experiment to mutate the species in the lowest scoring influence vector to hopefully bring that score up and the two scores within T_m of each other. Case 3 in Table 6.1 involves a merged influence vector having a score just lower than the maximum scoring subset. In this case, the `AddContender` function suggests an experiment

`AddContender(`$\langle$`IV` i_c`, Score` $\alpha_c\rangle$`, Case` c`,` $\langle$`IV` i_r`, Score` $\alpha_r\rangle$`)`

Case c

`1, 5:` $Suggest(Par(i_c) - Par(i_r)) := Suggest(Par(i_c) - Par(i_r)) + 1$

`2, 3:` $Suggest(Par(i_r)) := Suggest(Par(i_r)) + 1$

`4, 6:` $Suggest(Par(i_c)) := Suggest(Par(i_c)) + 1$

`7:` $A := (Par(i_c) - Par(i_r)) \cup (Par(i_r) - Par(i_c))$

$Suggest(A) := Suggest(A) + 1$

Figure 6.6. The `AddContender` algorithm.

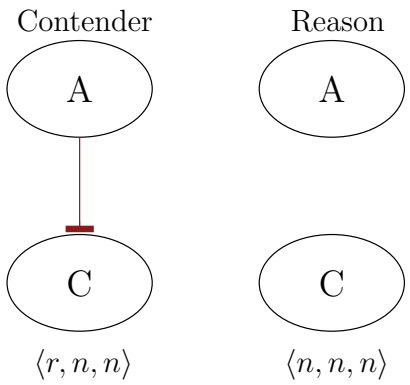


Figure 6.7. A contender illustrating Case 1 and Case 5 in Table 6.1.

to mutate the species in the highest scoring influence vector. The goal of this would be to lower the score of the highest scoring influence vector below that of the merged influence vector. As an example, consider Figure 6.8, where species A and species B influencing species C is the contender, while species A influencing species C is the reason. In both Case 2 and Case 3, a mutational experiment would be suggested for species A to either bring up the score for Case 2 as it is lower than it should be, or bring down the score for Case 3, as it is higher than it should be.

Case 4 and Case 6 in Table 6.1 involve an influence vector having a score that is just above the score of one of its' subsets. In this case, the `AddContender` function suggests an experiment to mutate all the species in the subset to increase its' score. If the contender

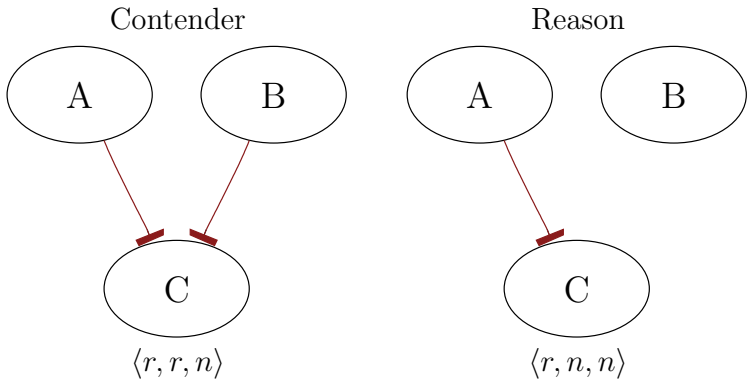


Figure 6.8. A contender illustrating Case 2 and Case 3 in Table 6.1.

is background knowledge and it contains no influences, (*i.e.,* $Par(i) = \emptyset$), more wild-type experiments are suggested in this case. As an example, consider Figure 6.9, where species A influencing species C is the contender, while species A and species B influencing species C is the reason. A mutational experiment is suggested for species A to bring up the score for this influence vector.

Finally, Case 7 in Table 6.1 involves two influence vectors being competed and if their scores are similar, the lower scoring one is considered a contender. In this case, the `AddContender` function suggests mutating all nonshared species. For example, consider the two influence vectors shown in Figure 6.10. Performing an experiment on either species A, or species D only adds support to the influence vector of which they are a part, however, performing a mutational experiment in which both species A and species D are mutated could add support to both influence vectors, and the influence vector that is the real one should see the most benefit from such an experiment. The `AddContender` function suggests an experiment in which both species A and species D are mutated in this case.

After the `GeneNet` algorithm is run and all the contenders have been added to the `Suggest` map using the `AddContender` function, the `GeneNet` algorithm uses a greedy approach for suggesting experiments. It looks in the `Suggest` map for the set of species which have been suggested the largest number of times and suggests to the researcher that they perform an experiment in which the species in the set are mutated low. If several sets have the same value, it selects the set with the least amount of species. If there are still sets with the same score it selects randomly from those sets.

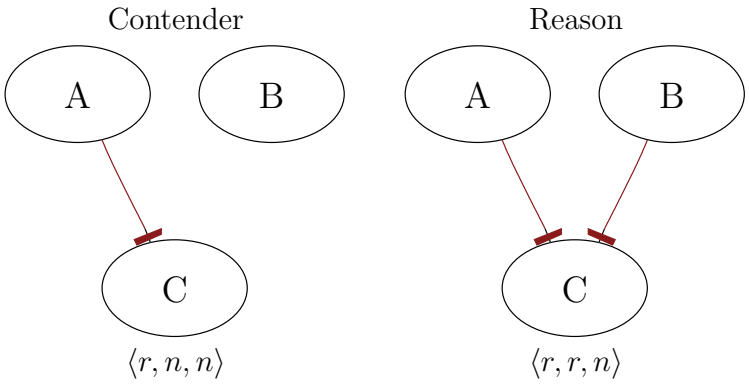


Figure 6.9. A contender illustrating Case 4 and Case 6 in Table 6.1.

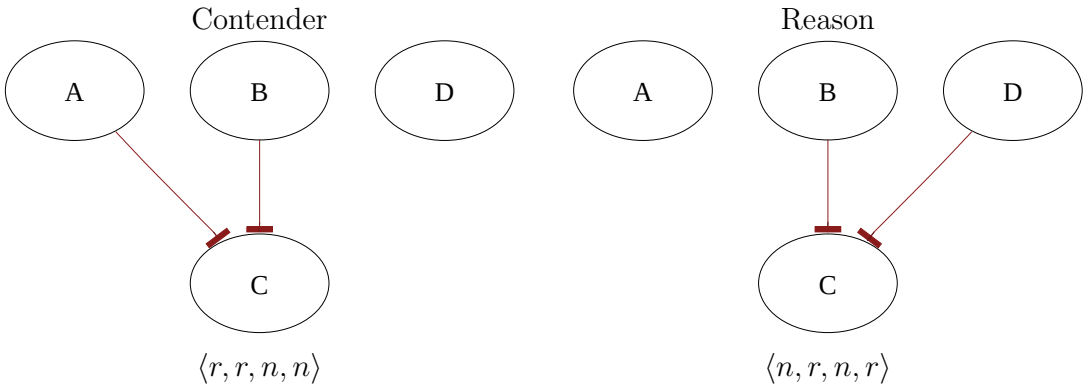


Figure 6.10. A contender illustrating Case 7 in Table 6.1.

CHAPTER 7

CASE STUDIES

All of the algorithms described in this dissertation have been implemented in the `GeneNet` tool that has been integrated with the `BioSim` tool being developed in our research group [14]. This chapter describes the ways in which the `GeneNet` algorithm is evaluated. It first describes how influence vector models are used to create experimental data that can be used by the `GeneNet` algorithm to learn a genetic network model. It then evaluates high level changes to the `GeneNet` algorithm and the effects these changes have on learning genetic network models. Next, it evaluates how changing the parameters or thresholds effects the ability of the algorithm to learn networks. It then describes how much data, and in what form, the `GeneNet` algorithm needs to learn good genetic network models. The `GeneNet` algorithm is then evaluated against the DBN method of Yu *et al.* Finally, an error analysis is performed on learned genetic networks to determine in what places the `GeneNet` algorithm is most likely to lose the best answer.

7.1 Setup

To evaluate our method, it must be tested on known networks. As there are currently very few known genetic networks, and even fewer with available time series data, we generate synthetic data for several networks of various sizes and types. The data generated are used to tune the parameters of the `GeneNet` algorithm. The evaluation procedure is shown in Figure 7.1. The black boxes have been added or updated from the original outline shown in Figure 1.1.

The evaluation procedure starts with an initial collection of influence vectors. This high level description of the gene interactions are translated into a very detailed reaction-based model expressed in the *Systems Biology Markup Language* (SBML) as described in Chapter 5. When an SBML model is created using the methods described in Chapter 5, many more species are introduced into the model than would normally be measured in an experiment. These include all the promoters as well as intermediate species. When

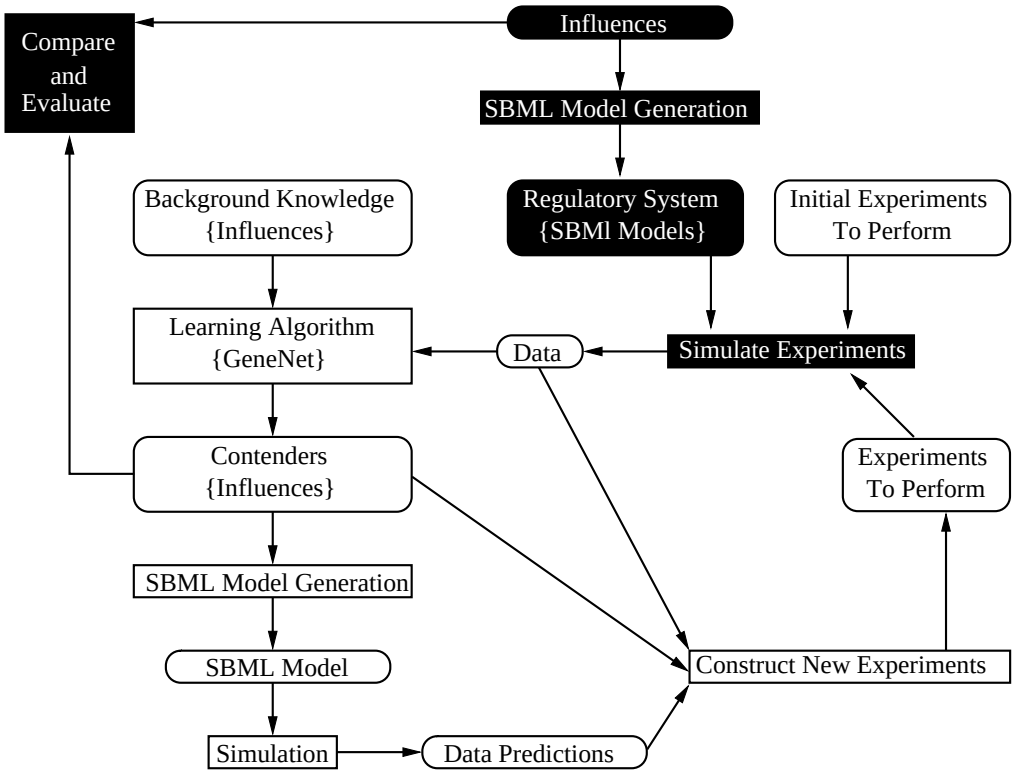


Figure 7.1. Data flow for the `GeneNet` algorithm evaluation.

learning genetic networks, these species are ignored.

As SBML is a purely reaction-based model it can be simulated with Gillespie's *stochastic simulation algorithm* ([32]), a version of which is implemented in our `BioSim` tool [14]. As discussed in Section 1.3, stochastic simulation of a genetic network model is often better suited for genetic networks. The simulation produces the time evolutions of species in a well-stirred biochemical system. Although time series data usually tell how many molecules of each species there are, or perhaps even more accurately the relative amount of each species, that is not exactly what is found in the time series data generated by the tool. Most species are involved in reactions which convert them to a different intermediate species. For example, in the phage λ decision circuit model created in Chapter 5, a molecule of Cro can bind to the N promoter creating a molecule of species Bound_repressor_1_N. This effectively removes one molecule of Cro from the output of the tool. The amount of the Cro molecule or species presented by the tool in the data is therefore not the amount of what is really in the system, but rather only

the amount that exists in a free form. This effectively introduces structured noise into the data, and becomes even more problematic if a species exists in many intermediate forms. To avoid this problem we could potentially look at all species which are composed of smaller species and add them back into the species amount. However, we currently do not do this as the systems that we have studied are small in nature and the connectivity is limited.

Once a model is constructed, a limited number of data points from simulation are selected to produce a synthetic time series data run for the original SBML model. The `GeneNet` algorithm uses these data to construct a network. At this point the original network is compared to the result of our method.

We have constructed 68 networks in which we evaluate our method. Figure 7.2 shows 48 four-gene networks inspired by the synthetic networks found in Guet *et al.* [37]. This extended set of networks includes those in Barker *et al.* [10]. These networks are constructed under the following assumptions. As stated in Guet *et al.,* the four species are TetR, LacI, CI, and GFP. The GFP promoter is always repressed by CI. Each of the other three promoters can be either repressed by TetR, LacI, or CI, or activated by CI. We have extended this formalism to allow a fourth option, in which a promoter can be repressed by either LacI or CI. The algorithm is also examined on the 10 randomly connected 10-gene networks shown in Figure 7.3 and the ten 20-gene networks from Yu *et al.* shown in Figure 7.4 [117].

In general both recall and precision are measured. Recall and precision are computed using the `RecallPrecision` function shown in Figure 7.5. This function takes the set of species S, the correct influences, $\mathcal{I}$, and the learned influences, $\mathcal{I}'$. For every species, s, it finds the associated influence vectors i and i' for this species. It then selects another species and determines if an activating or repressing arc is in i in which case the total number of arcs is incremented. If the same arc appears in the learned influence vector, i', then the correct counter is incremented. If an activating or repressing arc appears in the reported arcs, i', then the total number reported is incremented. Finally, recall and precision are calculated and returned. Recall is the number of correct arcs divided by the number of total arcs, and precision is the number of correct arcs divided by the number of total arcs. The `GeneNet` algorithm could have also been evaluated on the number of arcs that it correctly identifies as being absent. However, in general the species are sparsely connected so even a naive approach that reports that every species is unconnected

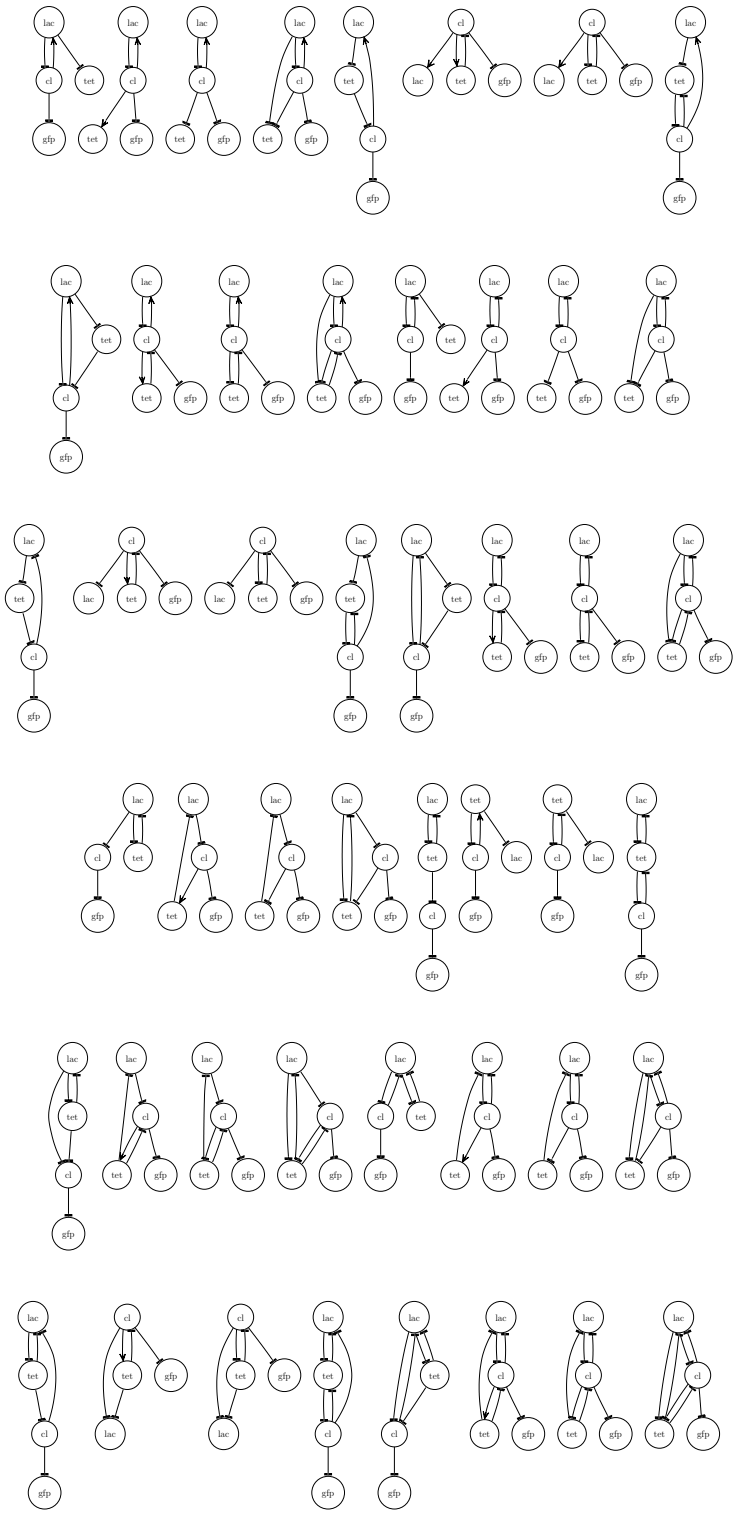


Figure 7.2. The 48 4-gene networks inspired by Guet *et al.* [37].

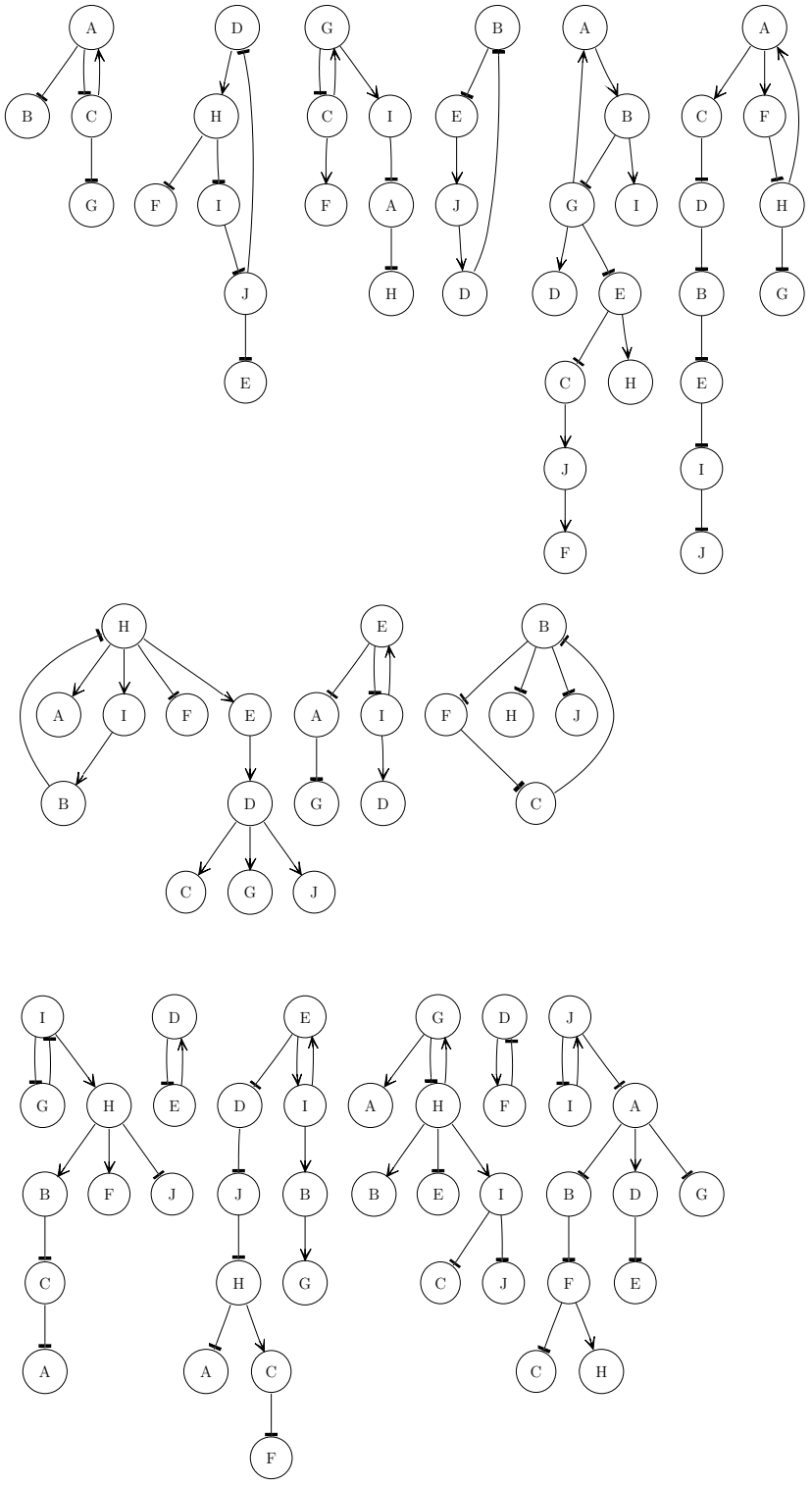


Figure 7.3. 10 randomly connected 10-gene networks.

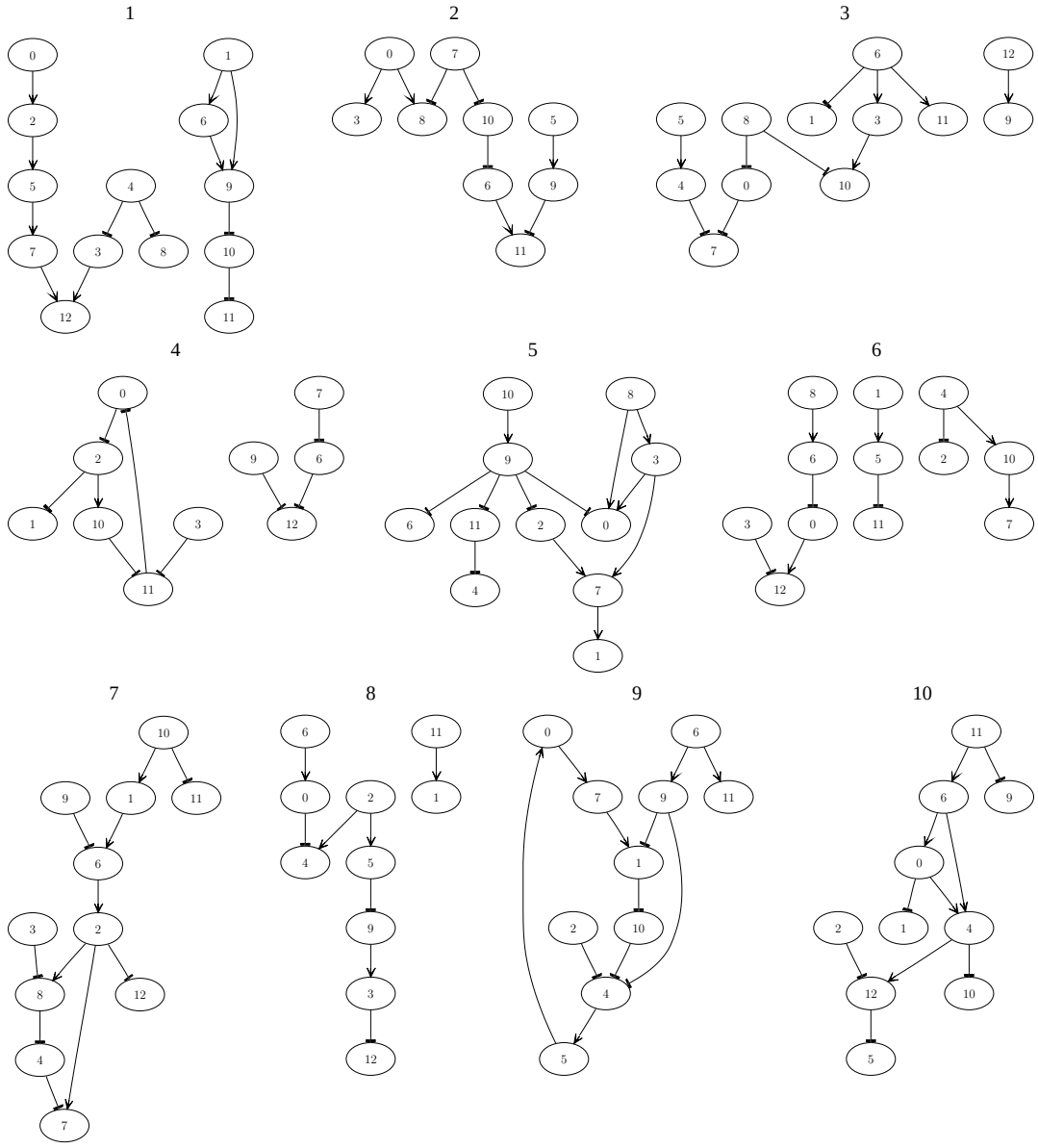


Figure 7.4. The 10 20-gene networks from Yu *et al.*.

```
RecallPrecision(Species Set S, Influences I, Influences I')
  ⟨correct, totalArcs, totalReported⟩ := ⟨0, 0, 0⟩
  foreach s ∈ S
    i := I(s)
    i' := I'(s)
    foreach s' ∈ S
      if i(s') = 'a' or i(s') = 'r' then
        totalArcs := totalArcs + 1
          if i(s') = i'(s') then
            correct := correct + 1
      if i'(s') = 'a' or i'(s') = 'r' then
        totalReported := totalReported + 1
  recall := correct / totalArcs
  precision := correct / totalReported
  return ⟨recall, precision⟩
```

Figure 7.5. The `RecallPrecision` function used to calculate recall and precision.

would predict correct more than 90 percent of the time. As this case is less interesting biologically as well, the ability to correctly predict nonexisting arcs is not evaluated in this section.

Throughout the rest of this chapter, many settings, parameters, and thresholds are evaluated. Table 7.1 shows the default settings used. Unless the setting is the one being tested, the default settings are those shown in Table 7.1. Table 7.2 shows the default time series data configuration.

7.2 Varying the GeneNet Algorithm

This section describes how a few high level changes to the `GeneNet` algorithm affect its ability to correctly learn genetic regulatory networks. The first change is in what data are used to calculate a probability in the `Score` function. The second change is the method used to calculate a base probability in the same function. The third and fourth changes involve binning the data. The time series data used in this section contain 20 experiments with 50 data points generated for each of these networks.

Section 2.3 describes how probabilities are calculated using data contained in both the $SUCC$ and $PRED$ sets. Table 7.3 shows how the `GeneNet` algorithm performs in recall and precision when using probabilities calculated from only the $SUCC$ or $PRED$ sets as well as the performance using both sets. As shown in the table, using the $SUCC$

Table 7.1. Default setting for the `GeneNet` algorithm.

Setting	Default Value
Set of Data Points	*SUCC*
`FindBaseProb` function	Advanced
Binning Method	Equal data per bin
Number of Bins	4
T_i	0.5
T_n	2
T_j	2
T_m	0
T_a	1.15
T_r	0.75
T_t	0.025
T_s	10

Table 7.2. Default setting for time series data generation.

Setting	Default Value
Number of Experiments	20
Number of Points Per Experiment	50
Measurement Interval	400
Duration	19600

Table 7.3. Recall, precision, and runtime results using data from *SUCC*, *PRED*, or both.

	Recall	Precision	Total Runtime
PRED	0.34	0.38	52.06
SUCC	0.68	0.79	53.85
Both	0.55	0.7	59.48

set gives the best results. The likely reason for this is that the $SUCC$ set gives the state before the species rises (*i.e.,* cause and effect) while the $PRED$ set gives the state after the rise (*i.e.,* correlation). Due to these results, the `GeneNet` algorithm uses only the $SUCC$ set as default.

As described in Section 3.4 and Section 3.6, a base probability can be calculated with the `FindBaseProbability` function from either a single partial bin assignment, or by using a more advanced algorithm that allows more probabilities to be calculated but at added complexity. The differences in recall and precision of these two approaches is evaluated as shown in Table 7.4. As seen in the table, using the advanced `FindBaseProbability` function performs slightly better in recall, and 10 percent better in precision than the basic implementation. A very unexpected result is that although the advanced function is more complex, it is slightly faster. One reason for this could be that using the advanced function, a base probability is more likely to be found, so that there is less chance that the thresholds need to be relaxed. Thus, adding some complexity to calculating the base probability does not seem to result in any loss of total runtime in these examples.

Section 2.4 describes how the `GeneNet` algorithm divides the data into bins containing equals amounts of data. The data can also be divided such that the values used to determine bins have the same span. Table 7.5 shows a comparison of these two methods. As seen from the table, spacing data evenly into bins results in a 5 percent increase in recall, and a 17 percent increase in precision. This shows that when calculating the change in rising probabilities it is more important to have good data in the bins, than to have them span equal ranges. Another interesting note is that the runtime increases somewhat when data are more evenly divided. This is because there are more entries in the CD and therefore more entries in each PCD and the runtime is directly affected when calculating rising probabilities in the `Score` function by data compression.

Section 2.4 describes how time series data are discretized into a small number of bins.

Table 7.4. Recall, precision, and runtime results varying the base probability method.

	Recall	Precision	Total runtime
Basic FindBaseProb	0.68	0.7	58.7
Advanced FindBaseProb	0.68	0.79	55.97

Table 7.5. Binning Method.

	Recall	Precision	Total Runtime
Equal data per bin	0.67	0.77	54.12
Equal spacing of bins	0.63	0.61	49.86

These bins determine how sparsely the data are divided. With fewer bins, there are more data per bin and the probabilities of a species increasing in expression can be more easily calculated, but there are fewer probabilities. With more bins, the data are more sparsely distributed. However, potentially more information as to the ability of a species to increase in expression can be calculated. Figure 7.6 shows how the `GeneNet` algorithm performs given various bin sizes as well as the runtime at the various number of bins. As the figure shows, precision increases dramatically between bins two and four, and recall

Number of Bins	Recall	Precision	Total Runtime
2	0.63	0.55	99.52
3	0.69	0.68	70.03
4	0.67	0.77	55.02
5	0.66	0.77	51.78
6	0.6	0.75	52.35

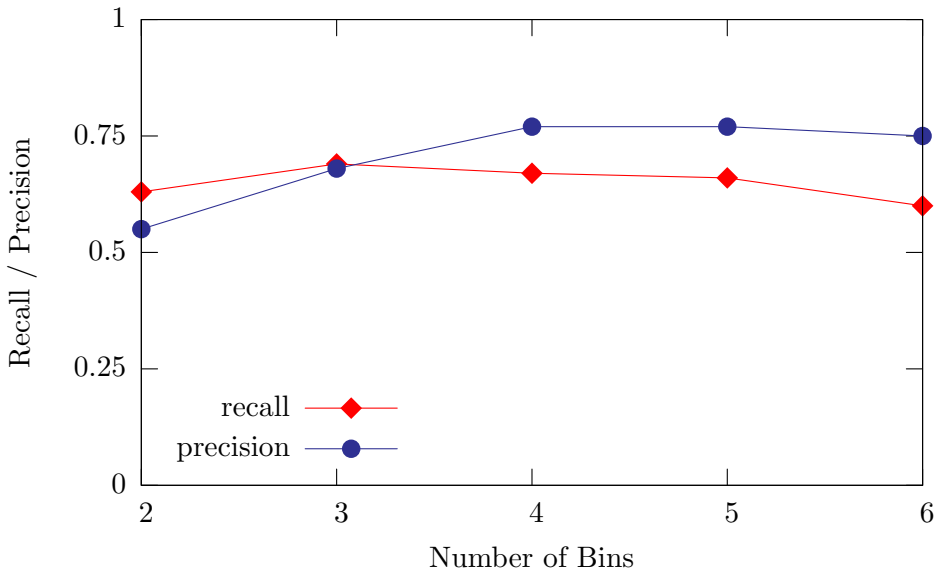


Figure 7.6. Varying the number of bins.

increases only slightly between bins two and three. Four bins gives the best recall and precision. Larger than four bins, both recall and precision start dropping, likely due to the amount of data per bin not being statistically significant.

7.3 Varying Parameters

This section describes how varying parameters in the `GeneNet` algorithm affects the ability of the algorithm to learn genetic networks. Specifically, the parameters are the T_i threshold which is used to score the background knowledge, the T_n threshold which determines the minimum number of initial influence vectors, the T_j threshold which determines the largest influence vector size, and the T_m threshold, which determines how close in score two influence vectors need to be in order to be considered for merging.

The threshold T_i is the value representing the score of an influence vector containing no influences. When no background knowledge is provided, this value is used in the `CreateInfluenceVectorSet` algorithm described in Section 4.1 to determine if an influence vector should be passed onto the next round. In this case, it is also used in the `Filter` function described in Section 4.2 to remove influence vectors that may have been added with relaxed thresholds, but that did not combine to form an influence vector with a high enough score. Figure 7.7 shows the differences in recall and precision when this threshold is varied as well as the runtimes at the various threshold values. A value of 0 represents no filtering and a value of 1 represents total filtering. As seen from the figure, a threshold value of 0.5 seems to give the best results for both recall and precision. Any lower than this value and precision suffers, and any higher recall suffers. As one would expect, as more and more filtering occurs, the runtime decreases as fewer influence vectors get passed on to the combine and compete algorithms.

The threshold T_n is the value representing the minimum number of influence vectors that are found in the `CreateInfluenceVectorSet` algorithm described in Section 4.1. The larger this value, the more likely that marginal influence vectors pass this round and may combine to form larger sets of influences. Figure 7.8 shows a comparison using different T_n values as well as the runtimes of the different threshold values. If the threshold value is larger than three for the four-gene networks then the algorithm is not able to find that number of initial influence vectors and instead yields the maximal value for these networks. As seen from the figure, there is only a very minimal change in recall for the values listed. Precision goes down slowly as more influence vectors are allowed to pass

	Recall	Precision	Total Runtime
0.0	0.65	0.47	250.81
0.10	0.69	0.54	130.75
0.20	0.71	0.58	96.04
0.30	0.7	0.6	75.84
0.40	0.72	0.68	64.15
0.50	0.67	0.77	54.91
0.60	0.62	0.84	51.46
0.70	0.53	0.83	48.76
0.80	0.36	0.81	45.1
0.90	0.22	0.72	44.01
0.99	0.17	0.73	41.25

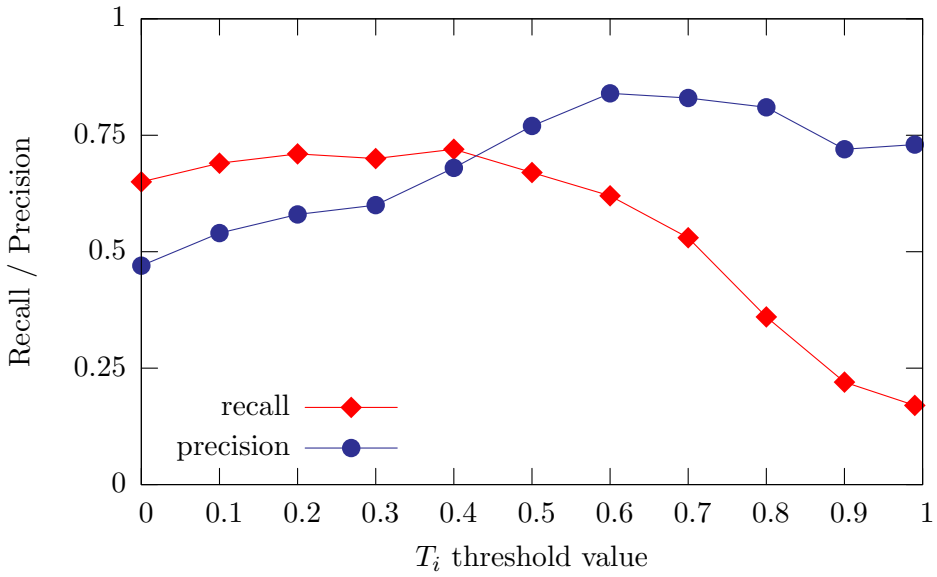


Figure 7.7. Score used for the background filter, using the T_i threshold.

the first round. This seems to suggest that the majority of influencing species are found by the `CreateInfluenceVectorSet` algorithm, and forcing more through only results in incorrect influence vectors being merged with correct influence vectors, possibly due to coorelation effects, and thereby reducing precision. From these data the default value could be either 0 or 2, and we selected to use 2. As one would expect, allowing more influence vectors through the first round results in longer runtimes.

The T_j threshold is used to determine the size of the largest influence vector created in the `CombineInfluenceVectors` algorithm described in Section 4.2. It also is used to limit the complexity of that algorithm as described in Section 4.4. A T_j of one indicates that

	Recall	Precision	Total Runtime
0	0.67	0.77	53.46
1	0.67	0.77	54.07
2	0.67	0.77	53.11
3	0.67	0.77	54.29
4	0.67	0.76	56.76
5	0.67	0.75	59.22
6	0.67	0.74	62.46
7	0.67	0.71	63.56
8	0.67	0.69	68.61

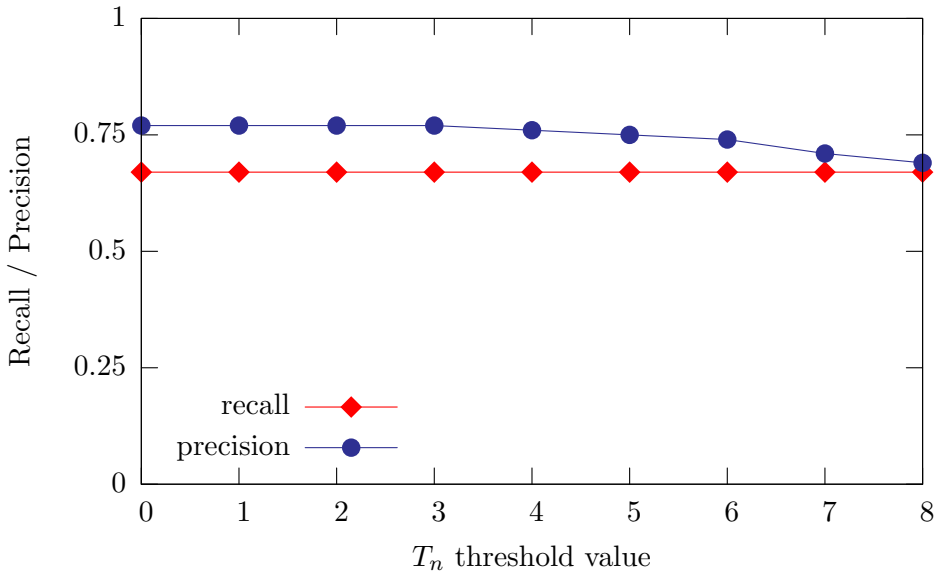


Figure 7.8. Minimum initial influences using the T_n threshold.

no influence vectors should be merged. A T_j of four indicates that influence vectors can be merged up to size four. Figure 7.9 shows the differences in recall and precision using various T_j values as well as the runtime for the various thresholds. As shown in the figure, there is only a slight change is recall, and a slight decrease in precision as the threshold is increased. This figure would seem to suggest that a T_j of one should be used. However, a T_j of two is more suitable due to the number of influence vectors of size one, compared to the number of influence vectors having a size larger than one in our data set. One would expect that increasing the T_j threshold would increase recall in a more dramatic way. However, as there are very few influence vectors with multiple influencing species in the data set this is not the case in this figure. Another reason for using a T_j of two would

	Recall	Precision	Total Runtime
1	0.67	0.81	47.56
2	0.67	0.77	53.03
3	0.67	0.76	56.89
4	0.67	0.76	56.86

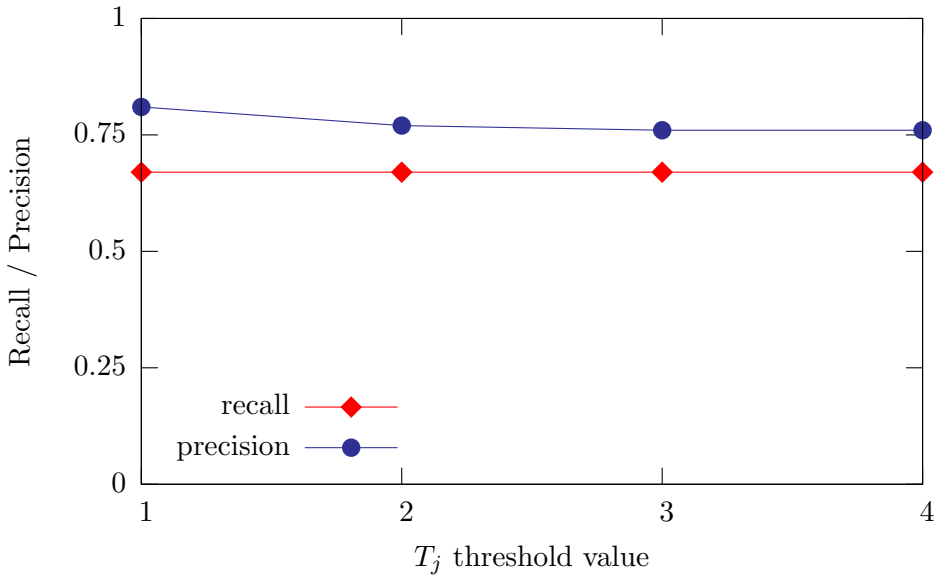


Figure 7.9. Maximum size of an influence vector using the T_j threshold.

be that this is the only way for an influence vector containing multiple influences in it to be suggested as an experiment that a researcher should perform, and thus the only way to learn any influence vectors of size two. As one would expect, increasing the maximum size of an influence vector increases the total runtime of the algorithm.

The T_m threshold determines how close in score two influence vectors need to be in order to be considered for combination in the `CombineInfluenceVectors` algorithm described in Section 4.2. This threshold is used to disallow a low scoring influence vector to be combined with a higher scoring influence vector as the lower scoring influence vector may simply be correlated with the higher scoring one. Figure 7.10 shows the differences in recall and precision as this threshold is varied as well as the runtimes for the various thresholds. From these data, a T_m value of 0 is clearly the best. As expected, at larger threshold values the runtimes are longer as more influence vectors are scored when considering if two influence vectors should be merged.

	Recall	Precision	Total Runtime
0	0.67	0.78	52.61
0.1	0.68	0.67	66.57
0.2	0.67	0.62	79.28
0.3	0.68	0.6	86.13
0.4	0.69	0.58	89.18
0.5	0.69	0.58	91.18

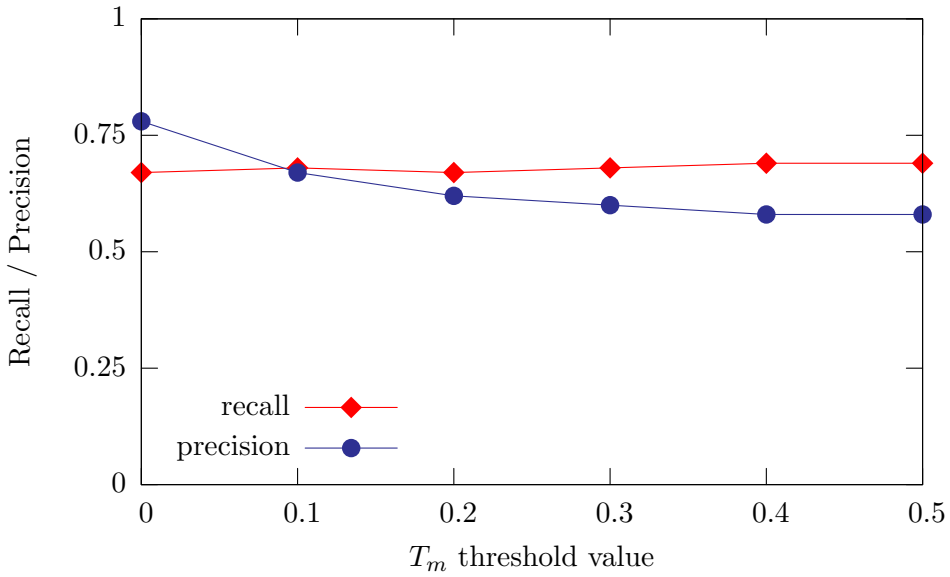


Figure 7.10. Merging influence vectors using the T_m threshold.

7.4 Varying Time Series Data Composition

This section describes how varying the composition of the time series data affects recall and precision. In particular, it is necessary to determine the ideal number of times that a time series experiment should be run, the number of data points per time series experiment, the time step between data points, and the duration of a time series data experiment. Since these values are all connected to each other, this section presents several plots varying groups of these parameters.

Figure 7.11 shows the effects on recall and precision given an experimental setup with 20 experiments and collecting data every 400 seconds as well as the runtimes of the `GeneNet` algorithm. The first point represents collecting 10 data points every 400 seconds. This results in a total runtime of an experiment of 3600 seconds. The second point represents collecting twice as much data, in twice the time and so on up to 10

	Recall	Precision	Total Runtime
10	0.41	0.5	7.09
20	0.55	0.59	18.11
30	0.61	0.67	29.83
40	0.63	0.69	40.97
50	0.66	0.77	50.85
60	0.69	0.77	64.19
70	0.75	0.81	72.86
80	0.72	0.79	84.23
90	0.73	0.83	96.27
100	0.75	0.87	106.84

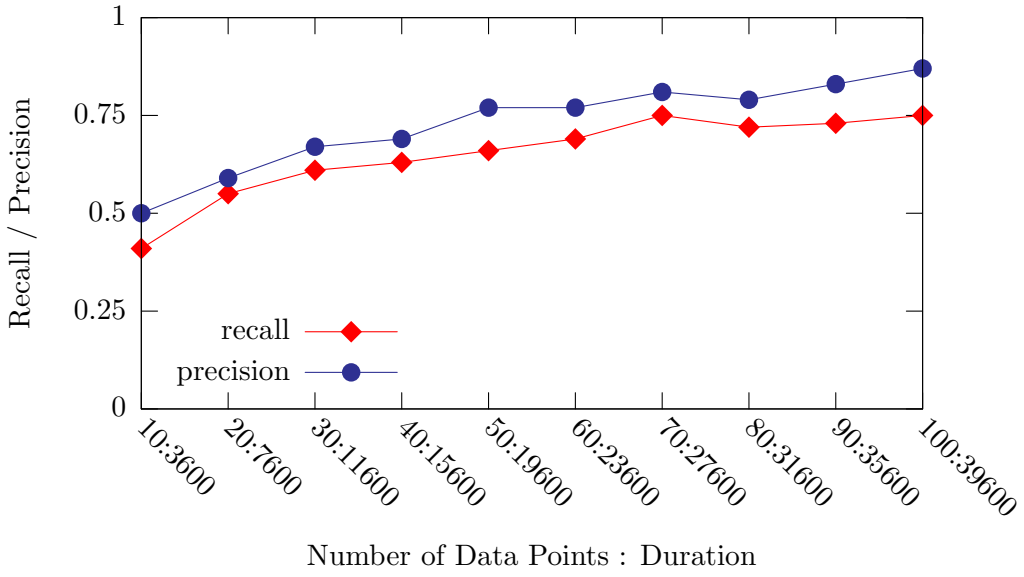


Figure 7.11. Varying the duration of the experiments given 20 experiments and a time step of 400.

times the data and a time series run of 39600 seconds. As seen from the figure, recall and precision increase fairly steadily up to 50 data points, and then both precision and recall level off. As there is not much improvement past 50 data points, and collecting less data is easier to do, 50 data points per experiment is chosen. Also, as one would expect, adding more data increases the runtime of the algorithm, though in a fairly linear fashion.

Figure 7.12 shows recall and precision of the `GeneNet` algorithm given 5 experiments up to 30 experiments each with 50 data points per experiment, a time step of 400 seconds and duration of 19600 seconds as well as the runtimes of the `GeneNet` algorithm. As seen

	Recall	Precision	Total Runtime
5	0.44	0.46	11.66
10	0.57	0.62	25.29
15	0.61	0.68	38.24
20	0.68	0.79	54.55
25	0.72	0.8	73.27
30	0.71	0.79	89.76

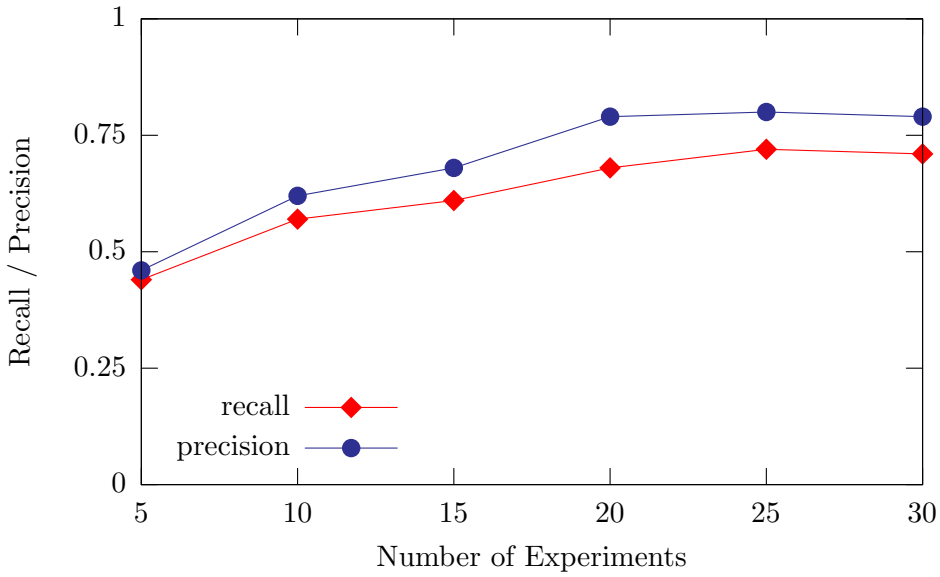


Figure 7.12. Varying the number of experiments given 50 data points per experiment and a time step of 400.

from the data, recall and precision increase steadily up to 20 experiments. Between 20 and 25 experiments precision exhibits a small rise and recall a large one at which point both precision and recall level off. Both 20 and 25 experiments seem to be suggested by this curve as the optimal number of experiments, but as performing less experiments is beneficial to the researcher 20 experiments is selected.

Figure 7.13 shows the effects of varying the number of experiments and time step for a fixed experiment duration of 19600 seconds as well as the runtime for the various points. Experiment 10 in the figure contains 10 experiments at a time step of 200. Experiment 20 contains 20 experiments each with a time step of 400. Experiment 50 contains 400 data points per experiment at a time step of 1000. In other words, each group of experiments includes 1000 data points. As seen from the figure, there is little variation between the points, and the best results in recall and precision appear to be for 30 experiments with

	Recall	Precision	Total Runtime
10	0.62	0.71	50.01
20	0.66	0.77	52.56
30	0.71	0.78	54.09
40	0.68	0.78	51.28
50	0.69	0.76	49.95

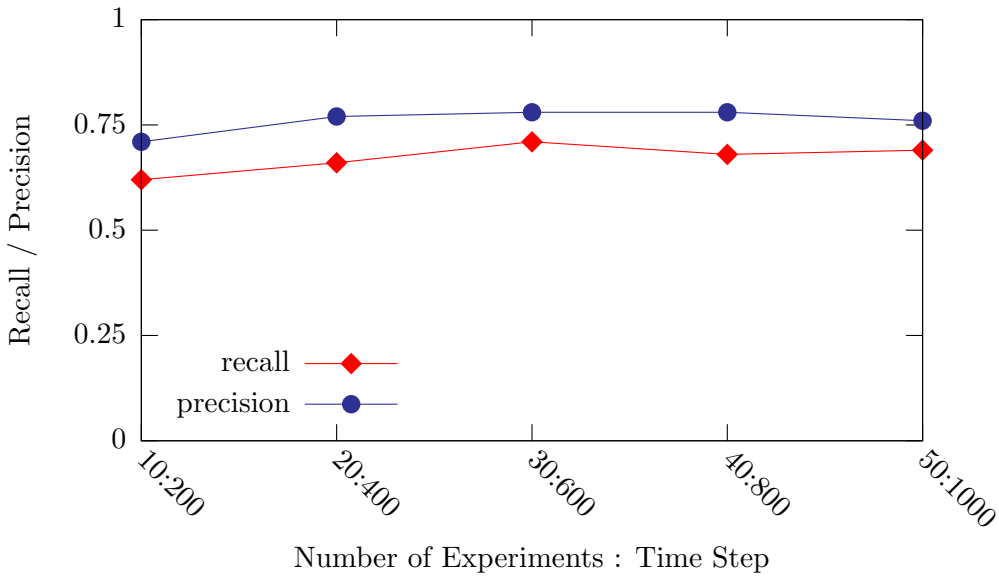


Figure 7.13. Varying the number of experiments and time step given 1000 data points and a duration of 19600.

a time step of 600 seconds.

Figure 7.14 shows the effect of varying the number of experiments and experiment duration using a fixed time step of 400 seconds and 1000 data points. Again, there is little variance between the different points. 20 experiments with a duration of 19600 seconds contains the maximum precision, whereas 30 experiments with a duration of 12,933 seconds contains the maximum recall.

Given all the results in this section, we decided to use 1000 data points distributed amongst 20 experiments, with a time step of 400 seconds, and duration of 19,600 seconds. These results indicate that redistributing the data points amongst somewhat different number of experiments, time steps, or duration has minimal effect. Of course, increasing the number of data points improves the results, but it results in a substantial increase in laboratory time. Therefore, 1000 data points seems to be a good compromise.

	Recall	Precision	Total Runtime
10	0.65	0.73	49.81
20	0.66	0.77	52.64
30	0.7	0.74	58.07
40	0.69	0.75	57.91
50	0.67	0.75	54.93

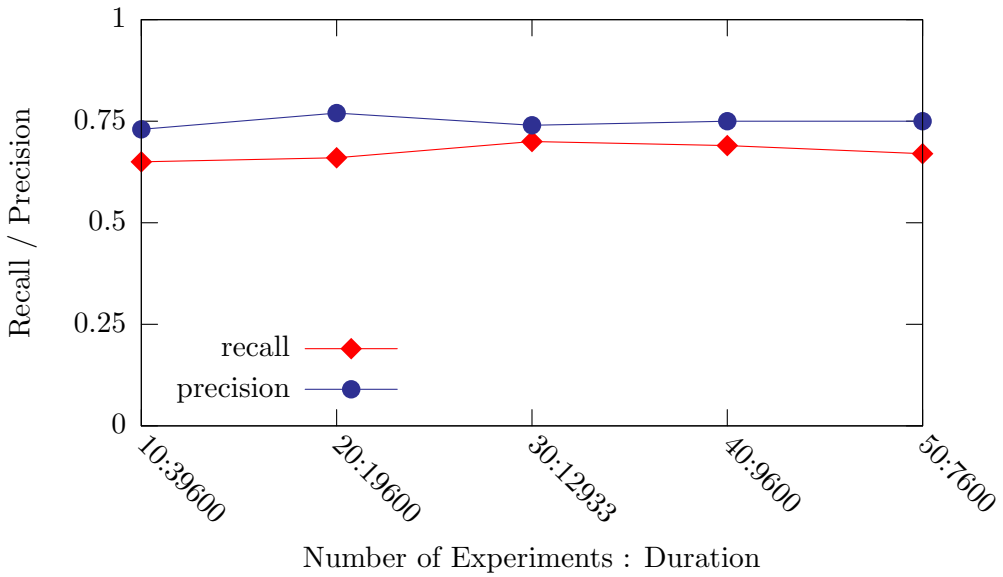


Figure 7.14. Varying the number of experiments and duration.

7.5 Comparisons

This section compares the `GeneNet` algorithm against Yu's DBN tool [117]. As described in Section 1.2.7, Yu's DBN tool first generates a DBN to determine the initial structure of the network. It then builds a *cumulative distribution function* to determine a score for each species that may potentially have influence over this gene. To compare the two tools, we provided both tools with the data described in Table 7.2 and allowed them to learn a network. Given the 68 networks used in this section, the `GeneNet` algorithm performs better in recall than Yu's DBN tool in 56 of the 68 cases, ties in 7 cases, and loses in 5 cases. A scatter plot shows these results in Figure 7.15. The `GeneNet` algorithm performs better in precision than Yu's DBN tool in 17 cases, ties in 13, and loses in 17. Note that groups of points with the same value are shown in a offset fashion. Also note that in 21 of the networks Yu's DBN tool reports no arcs for the entire network. The scatter plot for these results is shown in Figure 7.16 with the 21 networks containing no

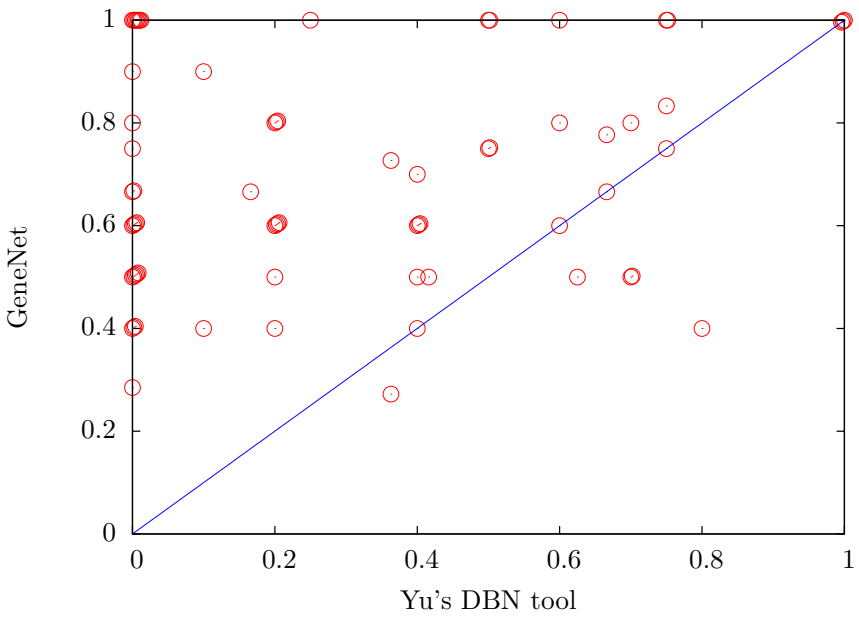


Figure 7.15. Recall scatter plot comparing results from the `GeneNet` algorithm to Yu's DBN tool for 68 genetic networks. The `GeneNet` algorithm wins in 56 and ties 7 of the 68 cases.

arcs removed from the graph. The runtime efficiency of the algorithm presented in this paper is also much better than Yu's DBN tool. Table 7.6 shows a runtime comparison with speedups of about an order of magnitude. Figure 7.17 shows the recall comparisons by example type, Figure 7.18 shows the precision comparisons by example type, and Figure 7.19 shows the runtime comparisons by example type. Table 7.7 and Table 7.8 show the total number of arcs, the number correct, the number of reported arcs, recall, and precision for each example.

These results show that the `GeneNet` algorithm significantly outperforms Yu's DBN tool in recall and runtime while performing nearly as well in precision. As Table 7.7 and Table 7.8 show, the small networks containing high feedback tend to be more difficult for Yu's DBN tool to learn when compared to the `GeneNet` algorithm, and for the networks containing many unconnected node, the `GeneNet` algorithm tends to add more connections than Yu's DBN tool. By increasing the T_i parameter, the number of arcs reported by the GeneNet algorithm is reduced. Table 7.9, Figure 7.20, and Figure 7.21 show the comparison of the GeneNet algorithm to Yu's DBN tool using a T_i value of 0.6. While

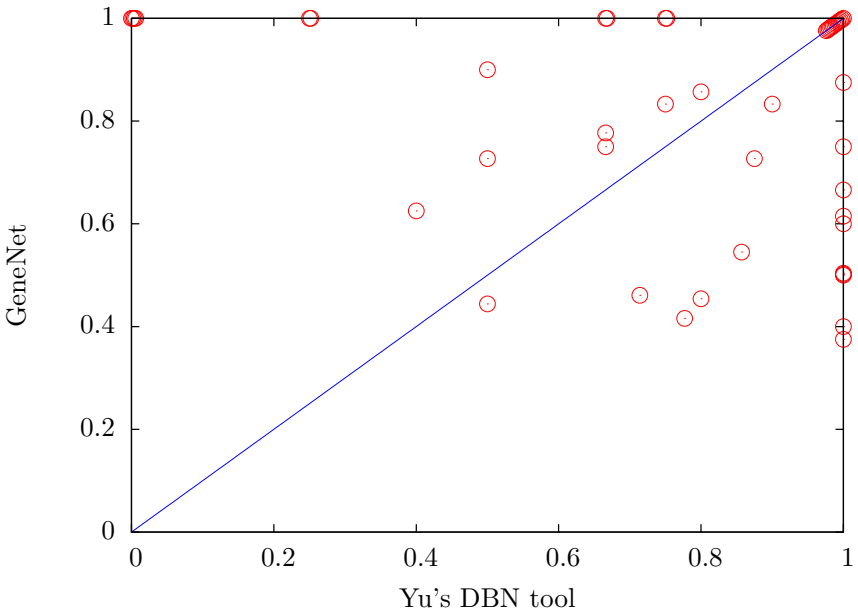


Figure 7.16. Precision scatter plot comparing results from the `GeneNet` algorithm to Yu's DBN tool for 47 genetic networks. The `GeneNet` algorithm wins in 17 and ties in 13 of the 47 cases. Note that groups of points with the same value are shown in an offset fashion. Note that there are an additional 21 networks in which Yu's DBN tool reports no arcs which are not shown in the graph.

Table 7.6. Recall, Precision, Runtime and Speedup comparisons.

	Yu's DBN tool			GeneNet			
Name	Recall	Prec.	Avg. Time	Recall	Prec.	Avg. Time	Speedup
Guet	0.22	0.74	0.14	0.69	0.92	0.03	4.67
Connected	0.31	0.67	4.38	0.68	0.69	1.24	3.53
Yu	0.57	0.92	112.97	0.63	0.61	4.02	28.10
Total	0.32	0.79	117.48	0.67	0.77	5.29	22.21

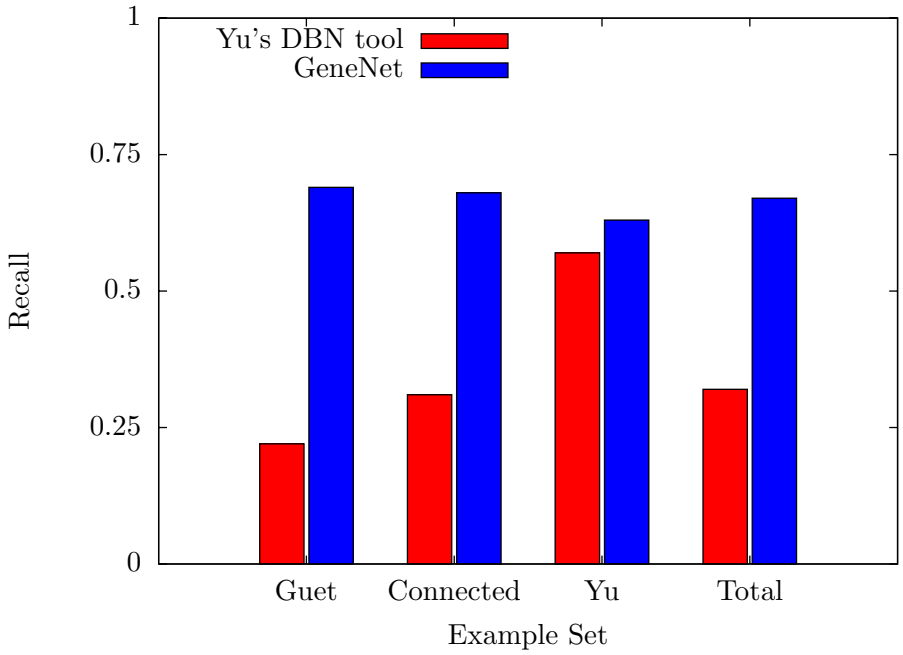


Figure 7.17. Recall comparisons by example type from Table 7.6.

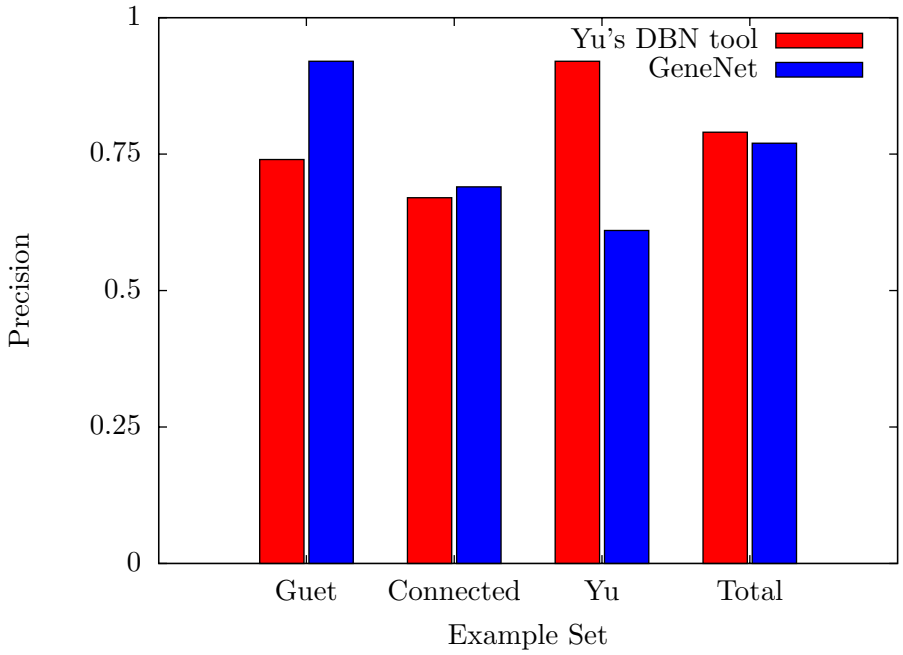


Figure 7.18. Precision comparisons by example type from Table 7.6.

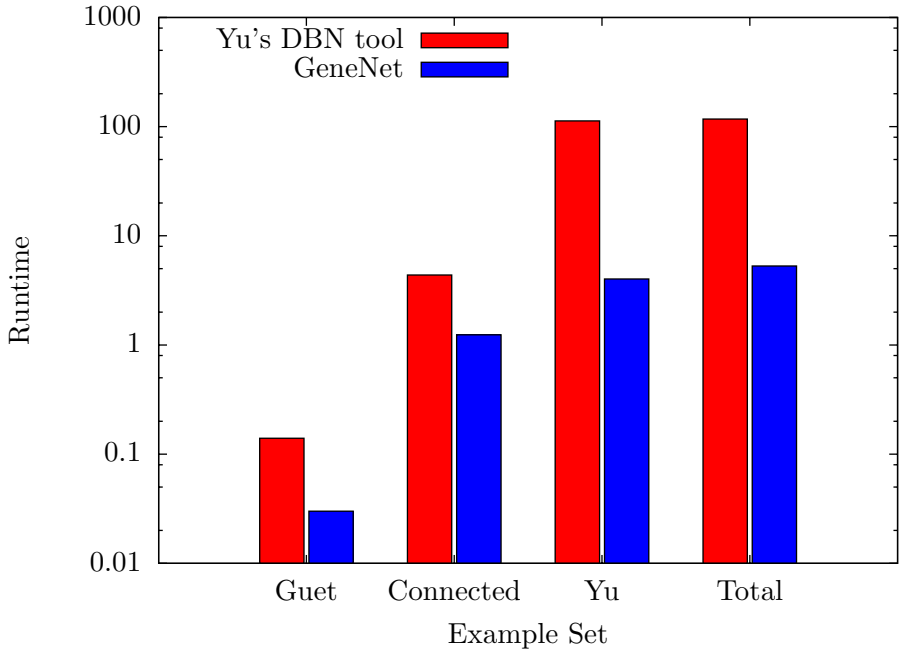


Figure 7.19. Runtime comparisons by example type from Table 7.6.

Table 7.7. Learning method evaluations for individual networks. Note that a '$*$' indicates that no arcs are reported for a given network and no precision score is calculated. For each method, the table shows the number of correct arcs reported, the total number of arcs reported, the recall and the precision for each network.

		GeneNet				Yu's DBN tool			
Network	Arcs	Corr.	Rept.	Recall	Prec.	Corr.	Rept.	Recall	Prec.
4-gene 01	4	4	4	1	1	0	1	0	0
4-gene 02	4	3	3	0.75	1	0	0	0	*
4-gene 03	4	4	4	1	1	0	0	0	*
4-gene 04	5	2	2	0.4	1	0	0	0	*
4-gene 05	4	4	4	1	1	4	4	1	1
4-gene 06	4	4	5	1	0.8	0	0	0	*
4-gene 07	4	4	4	1	1	0	0	0	*
4-gene 08	5	4	4	0.8	1	3	3	0.6	1
4-gene 09	5	3	3	0.6	1	1	1	0.2	1
4-gene 10	5	2	3	0.4	0.66	0	0	0	*
4-gene 11	5	2	4	0.4	0.5	2	2	0.4	1
4-gene 12	6	4	4	0.66	1	0	0	0	*
4-gene 13	4	4	4	1	1	3	3	0.75	1
4-gene 14	4	4	4	1	1	4	4	1	1
4-gene 15	4	4	4	1	1	2	2	0.5	1
4-gene 16	5	3	3	0.6	1	2	2	0.4	1
4-gene 17	4	4	4	1	1	0	0	0	*
4-gene 18	4	4	4	1	1	0	0	0	*
4-gene 19	4	4	4	1	1	2	3	0.5	0.66
4-gene 20	5	3	4	0.6	0.75	1	1	0.2	1
4-gene 21	5	4	4	0.8	1	0	1	0	0
4-gene 22	5	3	4	0.6	0.75	2	3	0.4	0.66
4-gene 23	5	3	3	0.6	1	0	4	0	0
4-gene 24	6	3	3	0.5	1	0	1	0	0
4-gene 25	4	4	4	1	1	4	4	1	1
4-gene 26	4	3	3	0.75	1	2	3	0.5	0.66
4-gene 27	4	4	4	1	1	0	0	0	*
4-gene 28	5	3	3	0.6	1	0	0	0	*
4-gene 29	4	3	3	0.75	1	3	4	0.75	0.75
4-gene 30	4	4	4	1	1	1	1	0.25	1
4-gene 31	4	4	4	1	1	3	4	0.75	0.75
4-gene 32	5	4	4	0.8	1	1	4	0.2	0.25
4-gene 33	5	3	3	0.6	1	1	1	0.2	1
4-gene 34	5	2	5	0.4	0.4	1	1	0.2	1

Table 7.8. Learning method evaluations for individual networks. Note that a '$*$' indicates that no arcs are reported for a given network and no precision score is calculated. For each method, the table shows the number of correct arcs reported, the total number of arcs reported, the recall and the precision for each network.

		GeneNet				Yu's DBN tool			
Network	Arcs	Corr.	Rept.	Recall	Prec.	Corr.	Rept.	Recall	Prec.
4-gene 35	5	3	3	0.6	1	0	0	0	*
4-gene 36	6	4	4	0.66	1	0	0	0	*
4-gene 37	5	3	3	0.6	1	1	4	0.2	0.25
4-gene 38	5	2	3	0.4	0.66	4	4	0.8	1
4-gene 39	5	4	4	0.8	1	1	1	0.2	1
4-gene 40	6	3	3	0.5	1	0	0	0	*
4-gene 41	5	3	3	0.6	1	0	0	0	*
4-gene 42	5	2	3	0.4	0.66	0	0	0	*
4-gene 43	5	3	3	0.6	1	3	3	0.6	1
4-gene 44	6	4	4	0.66	1	1	1	0.16	1
4-gene 45	6	3	3	0.5	1	0	0	0	*
4-gene 46	6	3	4	0.5	0.75	0	0	0	*
4-gene 47	6	3	4	0.5	0.75	0	0	0	*
4-gene 48	7	2	2	0.28	1	0	0	0	*
10-gene 01	10	8	11	0.8	0.72	2	4	0.2	0.5
10-gene 02	10	7	9	0.7	0.77	4	6	0.4	0.66
10-gene 03	10	6	7	0.6	0.85	4	5	0.4	0.8
10-gene 04	10	5	11	0.5	0.45	4	5	0.4	0.8
10-gene 05	10	5	8	0.5	0.62	2	5	0.2	0.4
10-gene 06	10	9	10	0.9	0.9	0	0	0	*
10-gene 07	10	10	12	1	0.83	6	8	0.6	0.75
10-gene 08	10	5	12	0.5	0.41	7	9	0.7	0.77
10-gene 09	10	4	9	0.4	0.44	1	2	0.1	0.5
10-gene 10	10	9	10	0.9	0.9	1	2	0.1	0.5
20-gene 01	12	10	12	0.83	0.83	9	10	0.75	0.9
20-gene 02	8	6	10	0.75	0.6	4	4	0.5	1
20-gene 03	10	8	11	0.8	0.72	7	8	0.7	0.87
20-gene 04	9	7	8	0.77	0.87	6	6	0.66	1
20-gene 05	12	6	13	0.5	0.46	5	7	0.41	0.71
20-gene 06	9	6	11	0.66	0.54	6	7	0.66	0.85
20-gene 07	11	8	13	0.72	0.61	4	4	0.36	1
20-gene 08	8	4	8	0.5	0.5	5	5	0.62	1
20-gene 09	11	3	8	0.27	0.37	4	4	0.36	1
20-gene 10	10	5	10	0.5	0.5	7	7	0.7	1

Table 7.9. Recall, Precision, Runtime and Speedup comparisons using a T_i value of 0.6.

	Yu's DBN tool			GeneNet			
Name	Recall	Prec.	Avg. Time	Recall	Prec.	Avg. Time	Speedup
Guet	0.22	0.74	0.14	0.64	0.94	0.03	4.67
Connected	0.31	0.67	4.38	0.57	0.69	1.10	3.98
Yu	0.57	0.92	112.97	0.63	0.80	3.89	29.04
Total	0.32	0.79	117.48	0.62	0.84	5.03	23.38

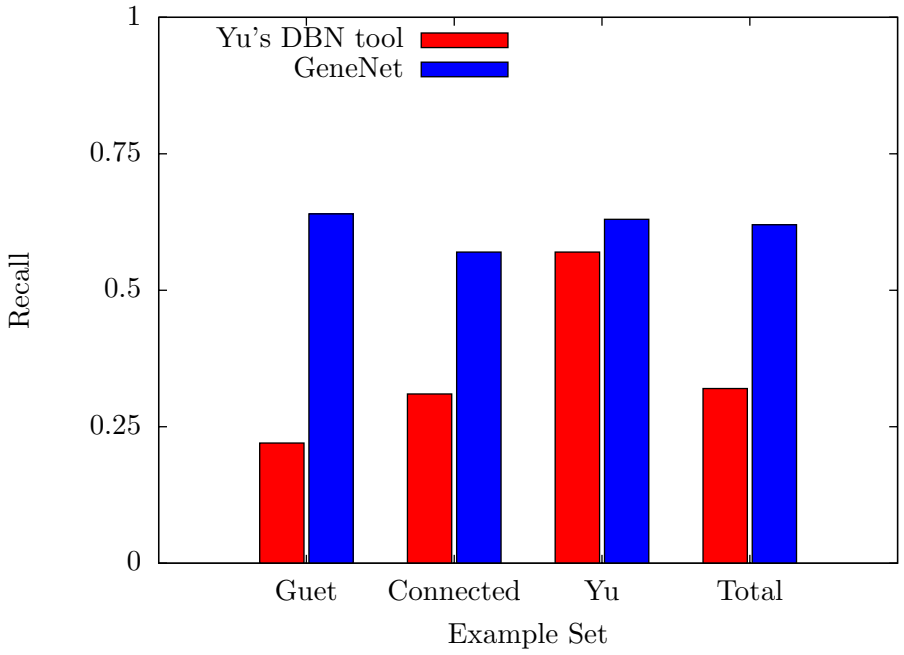


Figure 7.20. Recall comparisons by example type from Table 7.9 with a T_i value of 0.6.

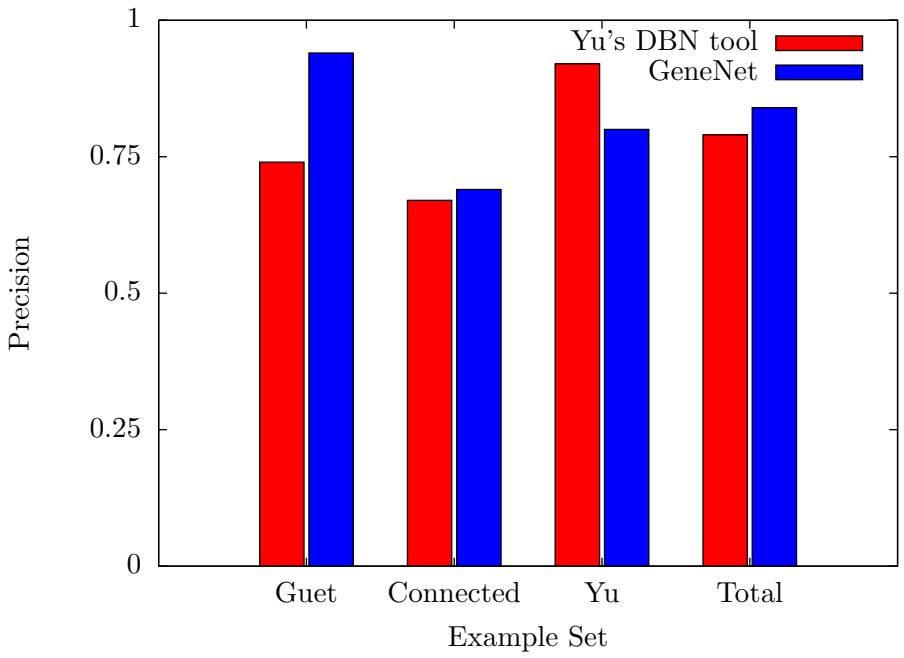


Figure 7.21. Precision comparisons by example type from Table 7.9 with a T_i value of 0.6.

for this value of T_i the recall results drop by 5 percent, they are still 30 percent better than those for Yu's DBN tool. The runtime results for the GeneNet algorithm are still over 20 times faster than Yu's DBN tool. Finally, in this case, the precision results are now 5 percent better than those for Yu's DBN tool.

7.6 Error Analysis

Although the `GeneNet` algorithm performs quite well, it is important to better understand when it fails to produce the correct influence vector. Table 7.10 shows an analysis performed on a default run of `GeneNet` for the 68 networks used throughout this chapter. Each entry represents how far the correct influence vector makes it during the execution of the `GeneNet` algorithm.

The first column represents species that are not regulated in any way. In other words, the influence vectors contain only 'n' entries. This influence vector is represented by the background knowledge influence vector. As the table shows, only Case 6 applies as this is the case in the `GeneNet` algorithm where background knowledge can get filtered. 23 of the influence vectors are filtered at this point and 98 times the correct answer appears.

Column two in Table 7.10 represents species that are regulated by exactly one other species. In other words, influence vectors with a size of one. Five of these influence vectors are filtered due a score less than background knowledge in Case 1. Cases 2, 3 and 6 do not remove these influence vectors. Case 4 represents that the influence vector is merged into a larger set of which there are 12. There are 31 influence vectors filtered due to a score less than background knowledge in Case 5, and 23 lost a competition in

Table 7.10. Cases where the correct influence vector failed.

Case	No Influences	Single Influences	Multiple Influences	Subsets
1		5		32
2			22	4
3			2	0
4		12		5
5		31	1	8
6	23			
7		23	0	2
Correct	98	241	1	21
Total	121	312	26	72

Case 7. Finally, the correct answer is found in 241 out of 312 cases.

The third column in Table 7.10 is for species with multiple influences and whose influence vectors are also considered fully by the `GeneNet` algorithm. That means that the merged influence vector is actually built and competed in some way. As the table shows, 22 did not have scores close enough together to be combined, two had a combined score less than the subsets, one is filtered due to background knowledge, and only one of these is correctly selected,

The fourth column in Table 7.10 is for species with multiple influences and whose influence vectors are never considered by the `GeneNet` algorithm in some way. Note that an influence vector of size three could represent a count of 3 under Case 1 as there are possibly 3 subsets of the influence vector. A total of 32 subsets are removed due to a lower score than background knowledge in Case 1. There are four subsets not combined due to their scores being too far apart in Case 2. No correct influence vectors score lower than their subsets in Case 3. Five of the subsets are merged with an incorrect influence vector in Case 4. Eight subsets are removed because there score is lower than background knowledge in Case 5. Two subsets lose in competition for Case 7, and 21 subsets are selected by the `GeneNet` algorithm as the winner, representing a partially correct result.

CHAPTER 8

CONCLUSIONS

Learning genetic networks is a new field as only recently have high throughput methods become available to study and record the expression levels of the species involved in genetic network reactions. This dissertation describes a method in which these regulatory networks can be learned, modeled, and then simulated. New experiments can then be suggested and this cycle repeated to increase the ability of the learning algorithm to correctly learn a regulatory network. The results on our generated test cases yield very promising results. This chapter also describes several areas in which this work can be extended and improved.

8.1 Summary

There are three distinct processes presented in this dissertation. The main process involves learning genetic networks using the `GeneNet` algorithm. The second is creating simulatable models from these networks. The third is suggesting experiments based on the results of the `GeneNet` algorithm.

As this dissertation describes, the `GeneNet` algorithm can be reliably used to learn genetic networks. It does so by organizing the values that occur in the time series data into bins and then searching these data to calculate the change in probability of expression increase from which influences between species are inferred. It learns influence vectors which can represent tight regulatory behavior and regulatory control structures. Influence vectors are easily represented graphically using directed graph notations which visually show how species are connected.

This dissertation also describes how influence vectors can be turned into a low level genetic network model described in SBML. This model allows for stochastic simulation of the reactions that occur in a genetic network to more reliably produce time series simulations. These simulations can be performed to help evaluate how networks behave in scenarios other than those tested in the laboratory.

Also presented in this dissertation is a method for suggesting experiments to help better learn genetic network models. This allows a researcher to skip experiments that are unnecessary and focus instead on interesting experiments. This method involves using the `GeneNet` algorithm as a tool in understanding which influence vectors best represent the data, and which are potential contenders.

This dissertation evaluates how the learning method performs using different internal algorithms and binning methods. It also evaluates how a variety of thresholds and data compositions effect learning. The learning method shows significant improvements over Yu's DBN tool in recall and runtime, while doing nearly as well or better in precision. Finally, the dissertation provides an error analysis to evaluate the various steps of the `GeneNet` algorithm.

8.2 Future Work

This work has focused on a new way in which to learn genetic networks, but there are still areas in which to improve upon this method. This section describes several of these areas. The first few involve manipulating the data itself to help boost probability calculations. The next few extend what is learned by the `GeneNet` algorithm to include more types of influences. A way in which the output from the `GeneNet` algorithm can be postprocessed to remove unwanted edges is then described. This dissertation also presents a way in which the influence vector representation can be translated into a simulatable model, but this process can be further improved. Finally, this section describes how the `GeneNet` algorithm can be applied to learn actual biological genetic networks.

8.2.1 Modifying the Data

When performing a time series data experiment on a genetic network, there are several places where noise can be introduced. One is in the stochastic nature of the networks themselves, and others come from the ways in which the data are gathered. Noise can be reduced by applying a filter to the data. The effect that this may have on the ability of the `GeneNet` algorithm to correctly learn a genetic network is one area of future work.

Another way in which the data can be modified is by interpolation. Interpolation can be used to create intermediate data points, effectively adding extra data into the experiment. This also has an added effect of smoothing the data, which may help in learning networks as just described. As shown in Chapter 7, having more data seems to

help when learning genetic networks. However, the effect of simply adding intermediate data points has yet to be studied thoroughly.

A third area in which the data can be modified is in how it is simulated. Currently, every time series data experiment starts with every species at its lowest amount. This creates many runs in which the same behavior is exhibited, meaning that the same states are reached and the same behavior observed. If each time series data experiment is started in different states, then there is a good chance that less data would be needed to observe how species influence each other.

8.2.2 Interventions

Although the `GeneNet` algorithm has been designed to suggest experiments that include interventions (*i.e.,* mutational data) and use the data from these experiments, little research has been done on the effectiveness of using this type of data. The use of these data may significantly improve the results, or it may be the case that the way in which the `GeneNet` algorithm breaks the data up into bins effectively creates the probabilities that would be found in a mutational experiments from traditional data. Also, the way in which these data are weighted may have a significant impact on the results. For example, given 19 wild type experiments and 1 mutational experiment, the mutational experiment would only be 1/20th of the data. However, all of the data for the species that is mutated are going into the same bin which may only exist in a few data points and thereby outweigh all 19 wild type experiments.

8.2.3 Detecting Self-Influences

As biological systems need to regulate the production of proteins, there are many ways in which this can occur. This dissertation focuses on those pathways in which species in a genetic network interact to regulate each other, but this is not the only way in which a species can be regulated. A species may activate or repress itself. While our method does allow for cycles as opposed to traditional Bayesian approaches, it does not yet detect these types of self-activation or repression influences.

8.2.4 Different Influence at Different Levels

The `Score` function, described in Chapter 3, is used to evaluate an influence vector. This function assumes that, for an activating species, as the amount of the activator increases the production of the influenced species also increases. It also assumes that

as the amount of repressor increases, the ability of the repressed species to be produced decreases. However, in some genetic networks a species may both activate and repress another species. It may be that at a low concentration, a species activates the production of another species, but at a high concentration, it may function as a repressor. Currently, the `GeneNet` algorithm has been designed to interpret this type of behavior to mean that these species are not connected. However, extensions to the `GeneNet` algorithm could be developed to look for this type of influence.

8.2.5 Improvements in Learning Multiple Influences

Overall, the results indicate that the `GeneNet` algorithm performs well in finding genes with zero or one influences, but it does not perform as well when there are multiple influences. The `GeneNet` algorithm uses the `CombineInfluenceVectors` algorithm when evaluating an influence vector and its subsets. As shown in Section 7.6, there are many cases where this approach is not optimal. Determining either a better way in which to evaluate an influence vector and its' subsets or finding better threshold values for this algorithm would be beneficial.

One of the basic assumptions of the `CombineInfluenceVectors` algorithm is that these species influence other species independent of the levels of the other influencing species. However, it is possible that two species must first bind together before they can exhibit an influence on another species. In other words, only when both of the species are present does this type of influence occur. This would be much like how, in the phage λ decision circuit, species CI must first form a dimer before it can repress species CII, except that in this example there would be two different species binding together. The `GeneNet` algorithm does not detect this type of behavior, as the probabilities calculated in the `Score` function involve the probability of production increasing between all the bin assignments to the two species and not just where the two species are at a high concentration. The `GeneNet` algorithm could be modified to try and learn this type of influence.

8.2.6 Transitive Edges

Potentially, the influencing arcs learned by the `GeneNet` algorithm may be due purely to correlational effects between species. In some cases, these arcs may be *transitive edges*, which means that when multiple genes are connected by a path, 'shortcuts' may be found. An example of this in the phage λ decision circuit is the 3 gene repression path from Cro

to CI to CII. Because Cro represses the repressor of CII, it activates CII in an indirect manner. This is not limited to simple two connection networks, or just repression. Paths of any length containing repression or activation arcs can create transitive edges. Once a genetic network is learned, a graph traversal algorithm may potentially be used to remove these transitive edges or mark them for further experiments.

8.2.7 More Accurate Modeling

As described in Chapter 5, SBML models can be created from the influence vectors learned by the `GeneNet` algorithm for a genetic network. These models are particularly useful as they can be simulated under conditions that may not be feasible, either physically or due to funding, in a laboratory. These models contain many biological reactions that are governed by a set of rate laws. Currently all similar reactions are given the same rates. One area that would improve the ability of the model to correctly reflect the biological system would be to learn the rates that govern each individual biological reaction. It would also be very useful to compare the output of a simulated network to the output of a real network to determine how accurately the SBML models are at capturing the behavior of the real network.

The SBML models produced do not model all types of genetic networks as the translation process only focuses on a few of the reactions that occur in genetic networks. There are other reactions that may be needed to construct an accurate model. Some of these include reactions in which a molecule prevents repression by binding to the repressor molecule, or more complicated degradation reactions such as the CII and CIII degradation mechanisms in the phage λ decision network.

8.2.8 Improving the Experimental Suggestion Method

The `GeneNet` algorithm uses a greedy approach for suggesting an experiment as described in Section 6.3. This approach may not always be the most optimal. Some cases in the `AddContender` function may actually be more valuable than others, however they are currently all treated the same. Also, more sophisticated experiments may potentially be suggested.

8.2.9 Learning Biological Networks

Chapter 7 describes the amount of time series data needed, and the way in which they need to be collected, for the `GeneNet` algorithm to correctly learn genetic networks.

Currently there are limited data of this type to use for learning in the `GeneNet` algorithm. As the goal of our research is to learn genetic networks and not synthetic networks that are simulated by computer, we plan to apply our method to data obtained from real genetic networks as these data sets become available.

REFERENCES

[1] Aebersold, R., and Mann, M. Mass spectrometry-based proteomics. *Nature 422* (March 2003), 198–207.

[2] Akutsu, T., Miyano, S., and Kuhara, S. Identification of genetic networks from a small number of gene expression patterns under the boolean network model, 1999.

[3] Albert, R., and Othmer, H. G. The topology of the regulatory interactions predicts the expression pattern of the segment polarity genes in *drosophila melanogaster*. *Journal of Theoretical Biology 223* (2003), 1–18.

[4] Andersen, J. B., Sternberg, C., Poulsen, L. K., Bjørn, S. P., Givskov, M., and Molin, S. New unstable variants of green fluorescent protein for studies of transient gene expression in bacteria. *Applied and environmental microbiology 64*, 6 (June 1998), 2240–2246.

[5] Andrec, M., Kholodenko, B. N., Levy, R. M., and Sontag, E. Inference of signaling and gene regulatory networks by steady-state perturbation experiments: structure and accuracy. *Journal of Theoretical Biology 232*, 3 (Feb 7 2005), 427–41.

[6] Arkin, A. Signal processing by biochemical reaction networks. In *Self-Organized Biological Dynamics and Nonlinear Control*, J. Walleczek, Ed. Cambridge University Press, 2000, ch. 5.

[7] Arkin, A., Ross, J., and McAdams, H. Stochastic kinetic analysis of developmental pathway bifurcation in phage lambda-infected escherichia coli cells. *Genetics 149* (1998), 1633–1648.

[8] Baldi, P., and Hatfield, G. W. *DNA Microarrays and Gene Expression*. Cambridge University Press, 2002.

[9] Bar-Joseph, Z., Gerber, G. K., Lee, T. I., Rinaldi, N. J., Yoo, J. Y., Robert, F., Gordon, D. B., Fraenkel, E., Jaakkola, T. S., Young, R. A., and Gifford, D. K. Computational discovery of gene modules and regulatory networks. *Nature Biotechnology 21*, 11 (Nov 2003), 1337–42.

[10] Barker, N., Myers, C., and Kuwahara, H. Learning genetic regulatory network connectivity from time series data. *Advances in Applied Artifical Intelligence 4031* (2006), 962–971.

[11] Basso, K., Margolin, A. A., Stolovitzky, G., Klein, U., Dalla-Favera, R., and Califano, A. Reverse engineering of regulatory networks in human cells. *Nature Genetics 37*, 4 (Apr 2005), 382–90.

[12] Beal, M. J., Falciani, F., Ghahramani, Z., Rangel, C., and Wild, D. L. Bayesian approach to reconstructing genetic regulatory networks with hidden factors. *Bioinformatics 21*, 3 (Feb 1 2005), 349–56.

[13] Bernard, A., and Hartemink, A. J. Informative structure priors: joint learning of dynamic regulatory networks from multiple types of data. *Pacific Symposium on Biocomputing* (2005), 459–70.

[14] BioSim. World Wide Web, `http://www.async.ece.utah.edu/`.

[15] BioSPICE. World Wide Web, `http://www.biospice.org/`, 2007.

[16] Boulesteix, A. L., and Strimmer, K. Predicting transcription factor activities from combined analysis of microarray and chip data: a partial least squares approach. *Theoretical Biology and Medical Modelling 2* (Jun 24 2005), 23.

[17] Bridges, C. B., and Brehme, K. S. *The mutants of Drosophila melanogaster*. Carnegie Institution of Washington, 1944.

[18] Brown, P. A., and Botstein, D. Exploring the new world of the genome with DNA microarrays. *Nature Genetics 21*, suppl. (1999), 33–37.

[19] Bussemaker, H. J., Li, H., and Siggia, E. D. Regulatory element detection using correlation with expression. *Nature Genetics 27*, 2 (Feb 2001), 167–71.

[20] Chip-on-chip. World Wide Web, `http://www.chiponchip.org/`, 2007.

[21] de Jong, H. Modeling and simulation of genetic regulatory systems: A literature review. *Journal of Computational Biology 9*, 1 (2002), 67–103.

[22] de Jong, H., Page, M., Hernandez, C., and Geiselmann, J. Qualitative simulation of genetic regulatory networks: Method and application. In *Proc. 17th Int. Joint Conf. Artif. Intell. (IJCAI-01)* (2001), Morgan Kaufmann Publishers, Inc., pp. 67–73.

[23] Dilworth, R. P. *Dilworth Theorems: Selected Papers of Robert P. Dilworth*. Contemporary Mathematicians. Birkhauser, 1990, pp. 7–18.

[24] Eisen, M. B., Spellman, P. T., Browndagger, P. O., and Botstein, D. Cluster analysis and display of genome-wide expression patterns. *Proceedings of the National Academy of Sciences 95* (December 1998), 14863–14868.

[25] Elowitz, M. B., and Leibler, S. A synthetic oscillatory network of transcriptional regulators. *Nature 403* (Jan 2000), 335–338.

[26] Friedman, N., Linial, M., Nachman, I., and Pe'er, D. Using bayesian networks to analyze expression data. *Journal of Computational Biology 7*, 3–4 (2000), 601–620.

[27] Friedman, N., Murphy, K., and Russell, S. Learning the structure of dynamic probabilistic networks. In *Uncertainty in Artificial Intelligence Conference* (1998), pp. 139–147.

[28] Gao, F., Foat, B. C., and Bussemaker, H. J. Defining transcriptional networks through integrative modeling of mrna expression and transcription factor binding data. *BMC Bioinformatics 5* (Mar 18 2004), 31.

[29] Garvie, C. W., and Wolberger, C. Recognition of specific sequences. *Molecular Cell 8*, 5 (Nov 2001), 937–46.

[30] Gibson, M., and Bruck, J. Efficient exact stochastic simulation of chemical systems with many species and many channels. *The Journal of Physical Chemistry A 104* (2000), 1876–1889.

[31] Giladi, H., Gottesman, M., and Oppenheim, A. B. Integration host factor stimulates the phage lambda p_l promoter. *Journal of Molecular Biology 213* (1990), 109–121.

[32] Gillespie, D. T. Exact stochastic simulation of coupled chemical reactions. *The Journal of Physical Chemistry 81*, 25 (1977), 2340–2361.

[33] Goodwin, B. C. *Temporal Organization of Cells*. Academic Press, 1963.

[34] Gottesman, S., and Gottesman, M. Protein degradation in *E. Coli*: the *lon* mutation and bacteriophage lambda N and CII protein stability. *Cell 24* (1981), 225–233.

[35] Gotthelf, A., and Lennox, J. *The first biologist: Perspectives on aristotle Philosophical Issues in Aristotle's Biology*. Cambridge University Press, 1987.

[36] Gouzé, J.-L., and Sari, T. A class of piecewise linear differential equations arising in biological models. Tech. rep., INRIA Sophia-Antipolis, Sophia-Antipolis, 2001. Technical Report RR-4207.

[37] Guet, C. C., Elowitz, M. B., Hsing, W., and Leibler, S. Combinatorial synthesis of genetic networks. *Science 296* (2002), 1466–1470.

[38] Guthke, R., Moller, U., Hoffmann, M., Thies, F., and Topfer, S. Dynamic network reconstruction from gene expression data applied to immune response during bacterial infection. *Bioinformatics 21*, 8 (Apr 15 2005), 1626–34.

[39] Gygi, S. P., Corthalsdagger, G. L., ZhangDagger, Y., Rochon, Y., and Aebersold, R. Evaluation of two-dimensional gel electrophoresis-based proteome analysis technology. *Proceedings of the National Academy of Sciences 97*, 17 (August 2000), 9390–9395.

[40] Hall, D. A., Zhu, H., Zhu, X., Royce, T., Gerstein, M., and Snyder, M. Regulation of gene expression by a metabolic enzyme. *Science 306*, 5695 (Oct 15 2004), 482–4.

[41] Hartemink, A. *Principled Computational Methods for the Validation and Discovery of Genetic Regulatory Networks*. PhD thesis, MIT, 2001.

[42] Hartemink, A. J., Gifford, D. K., Jaakkola, T. S., and Young, R. A. Using graphical models and genomic expression data to statistically validate models of genetic regulatory networks. *Pacific Symposium on Biocomputing* (2001), 422–33.

[43] Hastie, T., Tibshirani, R., Eisen, M. B., Alizadeh, A., Levy, R., Staudt, L., Chan, W. C., Botstein, D., and Brown, P. 'Gene shaving' as a method for identifying distinct sets of genes with similar expression patterns. *Genome Biology 1*, 2 (2000).

[44] Heckerman, D. A tutorial on learning with bayesian networks. Tech. rep., Microsoft Research, Microsoft Corporation, One Microsoft Way, Redmond, WA 98052, November 1996.

[45] Heidtke, K. R., and Schulze-Kremer, S. Design and implementation of a qualitative simulation model of lambda phage infection. *Bioinformatics 14*, 1 (1998), 81–91.

[46] Herzenberg, L. A., Parks, D., Sahaf, B., Perez, O., Roederer, M., and Herzenberg, L. A. The history and future of the fluorescence activated cell sorter and flow cytometry: A view from stanford. *Clinical Chemistry 48* (2002), 1819–1827.

[47] Hoyt, A. M., Knight, D. M., Das, A., Miller, H. I., and Echols, H. Control of phage lambda development by stability and synthesis of CII protein: role of the viral cIII and host *hflA*, *himA* and *himD* genes. *Cell 31* (1982), 565–573.

[48] Huber, W., von Heydebreck, A., and Vingron, M. Error models for microarray intensities. In *Encyclopedia of Genomics, Proteomics and Bioinformatics*, M. Dunn, L. Jorde, P. Little, and S. Subramaniam, Eds. John Wiley & Sons, 2004.

[49] Husmeier, D. Sensitivity and specificity of inferring genetic regulatory interactions from microarray experiments with dynamic bayesian networks. *Bioinformatics 19*, 17 (Nov 22 2003), 2271–82.

[50] Huxley, H. A. *Aphorisms and reflections from the works of T. H. Huxley.* Macmillan and Co., Limited, 1907.

[51] Ideker, T. E., Thorsson, V., and Karp, R. M. Discovery of regulatory interactions through perturbation: Inference and experimental design, 2000.

[52] Kærn, M., Blake, W. J., and Collins, J. The engineering of gene regulatory networks. *Annual Review of Biomedical Engineering 5* (2003), 179–206.

[53] Kahn, P. From genome to proteome: Looking at cell's proteins. *Science 270* (1995), 369–370.

[54] Karp, P. D., Riley, M., Saier, M., Paulsen, I. T., Paley, S. M., and Pellegrini-Toole, A. The ecocyc and metacyc databases. *Nucleic Acids Research 28*, 1 (2000), 56–59.

[55] Kauffman, S. A. Metabolic stability and epigenesis in randomly constructed genetic nets. *Journal of Theoretical Biology 22* (1969), 437–467.

[56] Kauffman, S. A. The large-scale structure and dynamics of gene control circuits: An ensemble approach. *Journal of Theoretical Biology 44* (1974), 167–190.

[57] Kholodenko, B. N., Kiyatkin, A., Bruggeman, F. J., Sontag, E., Westerhoff, H. V., and Hoek, J. B. Untangling the wires: a strategy to trace functional interactions in signaling and gene networks. *Proceedings of the National Academy of Sciences 99*, 20 (Oct 1 2002), 12841–6.

[58] Kim, P. M., and Tidor, B. Limitations of quantitative gene regulation models: a case study. *Genome Research 13* (November 2003), 2391–5.

[59] Kornitzer, D., Altuvia, S., and Oppenheim, A. B. The activity of the CIII regulator of lambdoid bacteriophages resides within 24-amino acid protein domain. *Proceedings of the National Academy of Sciences 88* (1991), 5217–5221.

[60] Kourilsky, P. Lysogenization by bacteriophage lambda: I. multiple infection and the lysogenic response. *Molecular and General Genetics 122* (1973), 183–195.

[61] Kundaje, A., Antar, O., Jebara, T., and Leslie, C. Learning regulatory networks from sparsely sampled time series expression data. Tech. rep., 2002.

[62] Kuwahara, H., Myers, C., Barker, N., Samoilov, M., and Arkin, A. Asynchronous abstraction methodology for genetic regulatory networks. In *The Third International Workshop on Computational Methods in Systems Biology* (2005).

[63] Kuwahara, H., Myers, C., Barker, N., Samoilov, M., and Arkin, A. Automatic abstraction methodology for genetic regulatory networks. In *The 1st Annual Mountain West Biomedical Engineering Conference* (2005).

[64] Kuwahara, H., Myers, C., Barker, N., Samoilov, M., and Arkin, A. Automated abstraction methodology for genetic regulatory networks. *Transactions on Computational Systems Biology VI 4220* (2006), 150–175.

[65] Lähdesmäki, H., Shmulevich, I., and Yli-Harja, O. On learning gene regulatory networks under the boolean network model. *Machine Learning 52* (2003), 147–167.

[66] Lee, T. I., Rinaldi, N. J., Robert, F., Odom, D. T., Bar-Joseph, Z., Gerber, G. K., Hannett, N. M., Harbison, C. T., Thompson, C. M., Simon, I., Zeitlinger, J., Jennings, E. G., Murray, H. L., Gordon, D. B., Ren, B., Wyrick, J. J., Tagne, J. B., Volkert, T. L., Fraenkel, E., Gifford, D. K., and Young, R. A. Transcriptional regulatory networks in saccharomyces cerevisiae. *Science 298*, 5594 (Oct 25 2002), 799–804.

[67] Lee, T. I., and Young, R. A. Transcription of eukaryotic protein-coding genes. *Annual Review of Genetics 34* (2000), 77–137.

[68] Liang, S., Fuhrman, S., and Somogyi, R. REVEAL, a general reverse engineering algorithm for inference of geneticnetwork architectures. In *Pacific Symposium on Biocomputing* (1998), vol. 3, pp. 18–29.

[69] Lipschutz, R. J., Fodor, S. P. A., Gingeras, T. R., and Lockhart, D. J. High density synthetic oligonucleotide arrays. *Nature Genetics 21*, suppl. (1999), 20–24.

[70] Lockhart, D. J., and Winzeler, E. A. Genomics, gene expressino and dna arrays. *Nature 405* (June 2000), 827–836.

[71] Mann, M. Quantitative proteomics. *Nature Biotechnology 17* (1999), 954–955.

[72] Matsuno, H., Doi, A., Nagasaki, M., and Miyano, S. Hybrid petri net representation of gene regulatory network. In *Pacific Symposium on Biocomputing* (Singapore, 2000), R. B. Altman, A. K. Dunker, L. Hunter, and T. E. Klein, Eds., vol. 5, World Scientific Press, pp. 341–352.

[73] McAdams, H. H., and Arkin, A. Stochastic mechanisms in gene expression. *Proceedings of the National Academy of Sciences 94*, 3 (1997), 814–819.

[74] McAdams, H. H., and Arkin, A. Simulation of prokaryotic genetic circuits. *Annual Review of Biophysics & Biomolecular Structure 27* (1998), 199–224.

[75] McAdams, H. H., and Arkin, A. Genetic regulation at the nanomolar scale: it's a noisy business. *Trends in Genetics 15*, 2 (1999), 65–69.

[76] McAdams, H. H., and Shapiro, L. Circuit simulation of genetic networks. *Science 269*, 5224 (1995), 650–656.

[77] McCollum, J. M., and Kemp, W. A. Ess documentation. World Wide Web, `http://biocomp.ece.utk.edu/Documentation/ESS_Documentation.pdf`, 2005.

[78] Meyers, S., and Friedland, P. Knowledge-based simulation of genetic regulation in bacteriophage lambda. *Nucleic Acids Research 12*, 1 (1984), 1–9.

[79] Myers, G. Lecture at the University of Utah, April 2004.

[80] Nachman, I., Regev, A., and Friedman, N. Inferring quantitative models of regulatory networks from expression data. *Bioinformatics 20 Suppl 1* (Aug 4 2004), 148–1256.

[81] Ong, I. M., Glasner, J. D., and Page, D. Modelling regulatory pathways in e. coli from time series experssion profiles. *Bioinformatics 18* (2002), s241–s248.

[82] Ong, I. M., and Page, D. Inferring regulatory pathways in e. coli using dynamic bayesian networks. Tech. rep., University of Wisconsin-Madison, May 2001.

[83] Orphanides, G., and Reinberg, D. Unified theory of gene expression. *Cell 108*, 4 (Feb 22 2002), 439–51.

[84] Pandey, A., and Mann, M. Proteomics to study genes and genomes. *Nature 405* (2000), 837–846.

[85] Pe'er, D. Bayesian network analysis of signaling networks: a primer. *Science's Signal Transduction Knowledge Environment 2005*, 281 (Apr 26 2005), pl4.

[86] Pennisi, E. A low number wins the genesweep pool. *Science 300* (June 2003).

[87] Perez, O. D., and Nolan, G. P. Simultaneous measurement of multiple active kinase states using polychromatic flow cytometry. *Nature Biotechnology 20* (2002), 155–162.

[88] Pilpel, Y., Sudarsanam, P., and Church, G. M. Identifying regulatory networks by combinatorial analysis of promoter elements. *Nature Genetics 29*, 2 (Oct 2001), 153–9.

[89] Ptashne, M. *A Genetic Switch.* Cell Press & Blackwell Scientific Publishing, 1992.

[90] Rabilloud, T. Two-dimensional gel electrophoresis in proteomics: old, old fashioned, but it still climbs up the mountains. *Proteomics 2* (January 2002), 3–10.

[91] Reinitz, J., and Vaisnys, J. R. Theoretical and experimental analysis of the phage lambda genetic switch implies missing levels of cooperativity. *Journal of Theoretical Biology 145* (1990), 295–318.

[92] Ren, B., and Dynlacht, B. D. Use of chromatin immunoprecipitation assays in genome-wide location analysis of mammalian transcription factors. *Methods in Enzymology 376* (2004), 304–15.

[93] Rice, J. J., Tu, Y., and Stolovitzky, G. Reconstructing biological networks using conditional correlation analysis. *Bioinformatics 21*, 6 (Mar 2005), 765–73.

[94] Robyr, D., Kurdistani, S. K., and Grunstein, M. Analysis of genome-wide histone acetylation state and enzyme binding using microarrays. *Methods in Enzymology 376* (2004), 289–304.

[95] Rogers, S., and Girolami, M. Bayesian regression approach to the inference of regulatory networks from gene expression data. *Bioinformatics 21*, 14 (Jul 15 2005), 3131–7.

[96] Sachs, K., Perez, O., Pe'er, D., Lauffenburger, D., and Nolan, G. Causal protein-signaling networks derived from multiparameter single-cell data. *Science 308* (April 2005), 523–529.

[97] Schafer, J., and Strimmer, K. An empirical bayes approach to inferring large-scale gene association networks. *Bioinformatics 21*, 6 (Mar 2005), 754–64.

[98] Segal, E., Shapira, M., Regev, A., Pe'er, D., Botstein, D., Koller, D., and Friedman, N. Module networks: identifying regulatory modules and their condition-specific regulators from gene expression data. *Nature Genetics 34*, 2 (Jun 2003), 166–76.

[99] Shea, M. A., and Ackers, G. K. The or control system of bacteriophage lambda: a physical-chemical model for gene regulation. *Journal of Molecular Biology 181* (1985), 211–230.

[100] Simon, I., Barnett, J., Hannett, N., Harbison, C. T., Rinaldi, N. J., Volkert, T. L., Wyrick, J. J., Zeitlinger, J., Gifford, D. K., Jaakkola, T. S., and Young, R. A. Serial regulation of transcriptional regulators in the yeast cell cycle. *Cell 106* (September 2001), 697–708.

[101] Spellman, P. T., Sherlock, G., Zhang, M. Q., Iyer, V. R., Anders, K., Eisen, M. B., Brown, P. O., Botstein, D., and Futcher, B. Comprehensive identification of cell cycle-regulated genes of the yeast *saccharomyce cerevisiae* by microarray hybridization. *Molecular Biology of the Cell 9* (December 1998), 3272–3297.

[102] Stark, J., Brewer, D., Barenco, M., Tomescu, D., Callard, R., and Hubank, M. Reconstructing gene networks: what are the limits? *Biochemical Society Transactions 31(Pt 6)* (Dec 2003), 1519–25.

[103] Systems Biology Workbench Development Group. `http://www.sbw-sbml.org/`.

[104] Szustakowski, J. Initial sequencing and analysis of the human genome. *Nature 409* (February 2001), 860–921.

[105] Thieffry, D., and Thomas, R. Dynamical behaviour of biological networks: Ii. immunity control in bacteriophage lamabda. *Bulletin of Mathematical Biology 57*, 2 (1995), 277–297.

[106] Thomas, R. Regulatory metworks seen as asynchronous automata: A logical description. *Journal of Theoretical Biology 153* (1991), 1–23.

[107] Thomas, R., and Ari, R. D. *Biological Feedback*. CCR Press, 1990.

[108] Torniuk, S., and Hofmann, K. Microarray probe selection strategies. *Briefings in Bioinformatics 2*, 4 (Dec 2001), 329–340.

[109] Turing, A. The chemical basis of morphogenesis. *Philosophical Transactions of The Royal Society of London Series B. 273* (1951), 37–72.

[110] Tyson, J., and Othmer, H. The dynamics of feedback control circuits in biochemical pathways. *Progress in Theoretical Biology 5* (1978), 1–62.

[111] Wang, W., Cherry, J. M., Nochomovitz, Y., Jolly, E., Botstein, D., and Li, H. Inference of combinatorial regulation in yeast transcriptional networks: a case study of sporulation. *Proceedings of the National Academy of Sciences 102*, 6 (Feb 8 2005), 1998–2003.

[112] Werhli, A. V., Grzegorczyk, M., and Husmeier, D. Comparative evaluation of reverse engineering gene regulatory networks with relevance networks, graphical gaussian models and bayesian networks. *Bioinformatics 22*, 20 (Oct 15 2006), 2523–31.

[113] Whalen, W., Ghosh, B., and Das, A. Nusa protein is necessary and sufficient *in vitro* for phage λ n gene product to suppress a ρ-independent terminator placed downstream of *nutL*. *Proceedings of the National Academy of Sciences 85* (1988), 2494–2498.

[114] Wille, A., Zimmermann, P., Vranova, E., Furholz, A., Laule, O., Bleuler, S., Hennig, L., Prelic, A., von P. Rohr, Thiele, L., Zitzler, E., Gruissem, W., and Buhlmann, P. Sparse graphical gaussian modeling of the isoprenoid gene network in arabidopsis thaliana. *Genome Biology 5*, 11 (2004), R92.

[115] Wolf, D. M., and Arkin, A. P. Fifteen minutes of *fim*: Control of type 1 pili expression in *e. coli*. *OMICS: A Journal of Integrative Biology 6*, 1 (2002), 91–114.

[116] Yeung, M. K., Tegner, J., and Collins, J. J. Reverse engineering gene networks using singular value decomposition and robust regression. *Proceedings of the National Academy of Sciences 99*, 9 (Apr 30 2002), 6163–8.

[117] Yu, J., Smith, V. A., Wang, P. P., Hartemink, A. J., and Jarvis, E. D. Advances to bayesian network inference for generating causal networks from observational biological data. *Bioinformatics 20* (December 2004), 3594–3603.

[118] Zak, D. E., Gonye, G. E., Schwaber, J. S., and 3rd, F. J. D. Importance of input perturbations and stochastic gene expression in the reverse engineering of genetic regulatory networks: insights from an identifiability analysis of an in silico network. *Genome Research 13*, 11 (Nov 2003), 2396–405.

[119] Zhu, H., and Snyder, M. Protein arrays and microarrays. *Current Opinion in Chemical Biology 5* (2001), 40–45.