BATCH MODE REINFORCEMENT LEARNING FOR CONTROLLING GENE REGULATORY NETWORKS AND MULTI-MODEL GENE EXPRESSION DATA ENRICHMENT FRAMEWORK

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

UTKU ŞİRİN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

JULY 2013

Approval of the thesis:

BATCH MODE REINFORCEMENT LEARNING FOR CONTROLLING GENE REGULATORY NETWORKS AND MULTI-MODEL GENE EXPRESSION DATA ENRICHMENT FRAMEWORK

submitted by **UTKU ŞİRİN** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. Adnan Yazıcı
Head of Department, **Computer Engineering** _____

Prof. Dr. Faruk Polat
Supervisor, **Computer Engineering Department, METU** _____

Examining Committee Members:

Prof. Dr. Volkan Atalay
Computer Engineering Department, METU _____

Prof. Dr. Faruk Polat
Computer Engineering Department, METU _____

Assoc. Prof. Dr. Tolga Can
Computer Engineering Department, METU _____

Assist. Prof. Dr. Öznur Taştan
Computer Engineering Department, Bilkent University _____

Assist. Prof. Dr. Mehmet Tan
Computer Engineering Department, TOBB ETÜ _____

Date: _____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: UTKU ŞİRİN

Signature :

ABSTRACT

BATCH MODE REINFORCEMENT LEARNING FOR CONTROLLING GENE REGULATORY NETWORKS AND MULTI-MODEL GENE EXPRESSION DATA ENRICHMENT FRAMEWORK

ŞİRİN, UTKU

M.S., Department of Computer Engineering

Supervisor : Prof. Dr. Faruk Polat

July 2013, 132 pages

Over the last decade, modeling and controlling gene regulation has received much attention. In this thesis, we have attempted to solve (i) controlling gene regulation systems and (ii) generating high quality artificial gene expression data problems. For controlling gene regulation systems, we have proposed three control solutions based on Batch Mode Reinforcement Learning (Batch RL) techniques. We have proposed one control solution for fully, and two control solutions for partially observable gene regulation systems. For controlling fully observable gene regulation systems, we have proposed a method producing approximate control policies directly from gene expression data without making use of any computational model. Results show that our proposed method is able to produce approximate control policies for gene regulation systems of several thousands of genes just in seconds without loosing significant performance; whereas existing studies get stuck even for several tens of genes. For controlling partially observable gene regulation systems, firstly, we have proposed a novel Batch RL framework for partially observable environments, Batch Mode TD(λ). Its idea is to produce approximate stochastic control policies mapping observations directly to actions probabilistically without estimating actual internal states of the regulation system. Results show that Batch Mode TD(λ) is able to produce successful stochastic policies for regulation systems of several thousands of genes in seconds; whereas existing studies cannot produce control solution for regulation systems of several tens of genes. To our best knowledge, Batch Mode TD(λ) is the first framework for solving non-Markovian decision tasks with limited number of experience tuples. For controlling partially observable gene regulation systems, secondly, we have proposed a method

to construct a Partially Observable Markov Decision Process (POMDP) directly from gene expression data. Our novel POMDP construction method calculates approximate observation-action values for each possible observation, and applies hidden state identification techniques to those approximate values for building the ultimate POMDP. Results show that our constructed POMDPs perform better than existing solutions in terms of both time requirements and solution quality. For generating high quality artificial gene expression data, we have proposed a novel multi-model gene expression data enrichment framework. We have combined four gene expression data generation models into one unified framework, and tried to benefit all of them concurrently. We have sampled from each generative models separately, pooled the generated samples, and output the best ones based on a multi-objective selection mechanism. Results show that our proposed multi-model gene expression data generation framework is able to produce high quality artificial samples from which inferred regulatory networks are better than the regulatory networks inferred from original datasets.

Keywords: Gene Regulatory Networks, Reinforcement Learning, Markov Decision Process, Partially Observable Markov Decision Process, POMDP Learning, TD(λ) algorithm, Control of GRN, Gene Expression Data, Data Enrichment

ÖZ

GEN AĞLARININ KISMİ GÖZLEMLENEBİLİR MARKOV SÜREÇLERİ İLE MODELLENEREK ETKİN OLARAK KONTROLÜ

ŞİRİN, UTKU

Yüksek Lisans, Bilgisayar Mühendisliği Bölümü

Tez Yöneticisi : Prof. Dr. Faruk Polat

Temmuz 2013 , 132 sayfa

Gen düzenleme sistemlerinin modellemesi ve kontrolü son on yılda çok dikkat çekmiştir. Bu tezde, (i) gen düzenleme sistemlerini kontrol etme ve (ii) yüksek kaliteli yapay gen ifade verisi üretme problemleri çözülmeye çalışılmıştır. Gen düzenleme sistemlerini kontrol etmek için, Yığın Modunda Pekiştirmeli Öğrenme (Batch RL) tekniklerine dayanan, üç kontrol çözümü önerilmiştir. Tam gözlemlenebilir gen düzenleme sistemleri için bir, kısmi gözlemlenebilir gen düzenleme sistemleri için iki kontrol çözümü önerilmiştir. Tam gözlemlenebilir düzenleme sistemlerinin kontrolü için, hiçbir berimsel yöntem kullanmadan, direkt olarak gen ifade verisinden yaklaşık kontrol politikaları üreten bir yöntem önerilmiştir. Sonuçlar gösteriyor ki, mevcut çalışmalar birkaç on genlik sistemler için çözüm üretemezken, bizim önerdiğimiz yöntem, sadece birkaç saniye içinde, kayda değer bir performans kaybı olmadan, binlerce genlik düzenleme sistemleri için yaklaşık kontrol politikaları üretebilmektedir. Kısmi gözlemlenebilir düzenleme sistemlerinin kontrolü için, ilk olarak, kısmi gözlemlenebilir ortamlar için yeni bir Batch RL taslağı, Yığın Modunda TD(λ), önerilmiştir. Temel fikir, gen düzenleme sisteminin gerçek iç dinamiklerini hesaplamadan, gözlemleri direkt olarak aksiyonlara olasılıksal olarak eşleyen yaklaşık kontrol politikaları üretmektir. Sonuçlar gösteriyor ki, mevcut çalışmalar birkaç on genlik sistemler için çözüm üretemezken, Yığın Modunda TD(λ), saniyeler içinde, binlerce genlik düzenleme sistemleri için başarılı olasılıksal kontrol politikaları üretebilmektedir. Bildiğimiz kadarı ile, Yığın Modunda TD(λ), kısıtlı deneyim bilgisi ile Markov özelliği göstermeyen karar problemlerini çözmek için önerilen ilk taslaktır. Kısmi gözlemlenebilir düzenleme sistemlerinin kontrolü için, ikinci olarak, direkt gen ifade verisinden Kısmi Gözlemlenebilir Markov Karar Süreci (POMDP) kuran bir yöntem önerilmiştir. Önerdiğimiz yeni POMDP kurma yöntemi, yaklaşık olarak

gözlem-aksiyon değerleri hesaplamakta, ve nihai POMDP'yi kurmak için, bu yaklaşık değerlere gizli durum belirleme yöntemlerini uygulamaktadır. Sonuçlar gösteriyor ki, önerdiğimiz POMDP kurma yöntemi, mevcut çalışmalardan hem zaman gerekliliği hem de çözüm kalitesi olarak daha iyi sonuçlar üretmektedir. Yüksek kaliteli yapay gen ifade verisi üretmek için, yeni bir çoklu-model gen ifade verisi zenginleştirme taslağı önerilmiştir. Dört gen ifade verisi üretme modeli tek bir taslakta birleştirilmiş, ve hepsinden aynı anda faydalanılmaya çalışılmıştır. Her üretici modelden veri alınıp, bu veriler birleştirilmiş ve en iyileri çoklu-amaçlı seçme mekanizması ile üretilmiştir. Sonuçlar, önerilen çoklu-model veri üretme sisteminin ürettiği yüksek kaliteli yapay gen ifade verilerinden çıkarsanan düzenleme ağlarının, orijinal gen ifade verilerinden çıkarsanan düzenleme ağlarından daha iyi olduğunu göstermektedir.

Anahtar Kelimeler: Gen Düzenleyici Ağlar, Pekiştirmeli Öğrenme, Markov Karar Süreci, Kısmi Gözlemlenebilir Markov Karar Süreci, POMDP Öğrenme, TD(λ) algoritması, GDA'ların Kontrolü, Gen İfade Verisi, Veri Zenginleştirme

To my bad luck

ACKNOWLEDGMENTS

I would like to thank my supervisor Prof. Faruk Polat. He has always welcomed me and my endless questions genially. He has been always there for my personal problems, as well. My academic perspective has been influenced greatly from his exclusive big picture view. I would like to thank Prof. Reda Alhajj for his collaboration and generous feedbacks. It was fun to collaborate with him.

I would like to thank Dr. Utku Erdoğdu for his valuable support and friendship. It was much harder for me to finish my thesis without his generous helps. I would like to thank Assist. Prof. Mehmet Tan for his helpful materials.

I would like to thank our chair Prof. Adnan Yazıcı, and vice chair Prof. Pınar Karagöz for welcoming me warmly to the department.

I would like to thank my father Hayrettin, mother Hatice, and sister Duygu for their constant supports. I have always been motivated by their supports. I would like to thank my girlfriend Selen for his lovely support and deep understanding.

Finally, I would like to thank residents of the office A-310 for their warm welcome and for the nice environment to work.

This work is supported by the Scientific and Technological Research Council of Turkey (TÜBİTAK) under Grant No. 110E179 and TÜBİTAK-BİDEB MS scholarship (2210) program.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ALGORITHMS

ALGORITHMS

LIST OF ABBREVIATIONS

MDP	Markov Decision Process
POMDP	Partially Observable Markov Decision Process
RL	Reinforcement Learning
Batch RL	Batch Mode Reinforcement Learning
FQI	Fitted Q Iteration
LSFQI	Least-Squares Fitted Q Iteration
GRN	Gene Regulatory Network
LSFTDI	Least-Squares Fitted TD(λ) Iteration
PBN	Probabilistic Boolean Network
BN	Boolean Network
ODE	Ordinary Differential Equation
HIMM	Hierarchical Markov Model
GA	Genetic Algorithm
TSNI	Time Series Network Identification

CHAPTER 1

INTRODUCTION

Genes are meaningful and functional molecular units of DNAs or RNAs as shown in Figure 1.1. A Gene regulation system is a system composed of a set of interacting genes that are regulating each other. Many cellular activities are accrued through the interactions of genes that are regulating each other. The basic idea for modeling a gene regulation system is to represent the system as a Gene Regulatory Network (GRN), where each node corresponds to a gene, and each edge corresponds to an interaction between genes. Hence, GRNs model gene regulation in terms of multivariate interactions among the genes. Figure 1.2 shows an example GRN which is a sub-network

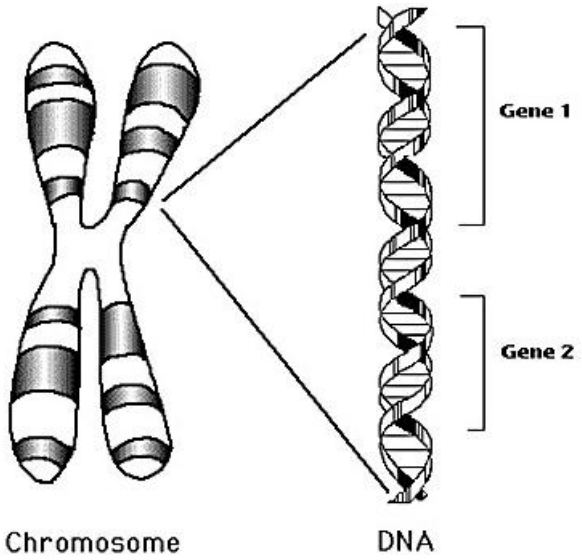

Figure 1.1: Genes

extracted from yeast cell cycle pathway obtained from KEGG [72]. There are different approaches for modeling gene regulation such as Bayesian networks, Boolean networks, and ordinary differential equations [31]. The idea is to understand the mathematical relationships between the genes. One of the major goals of modeling gene regulation, on the other hand, is to predict the behavior of the cellular system and to control it for developing therapies or drugs that would intervene the system dynamics. By designing intervention strategies, gene regulation systems can be controlled and prevented from

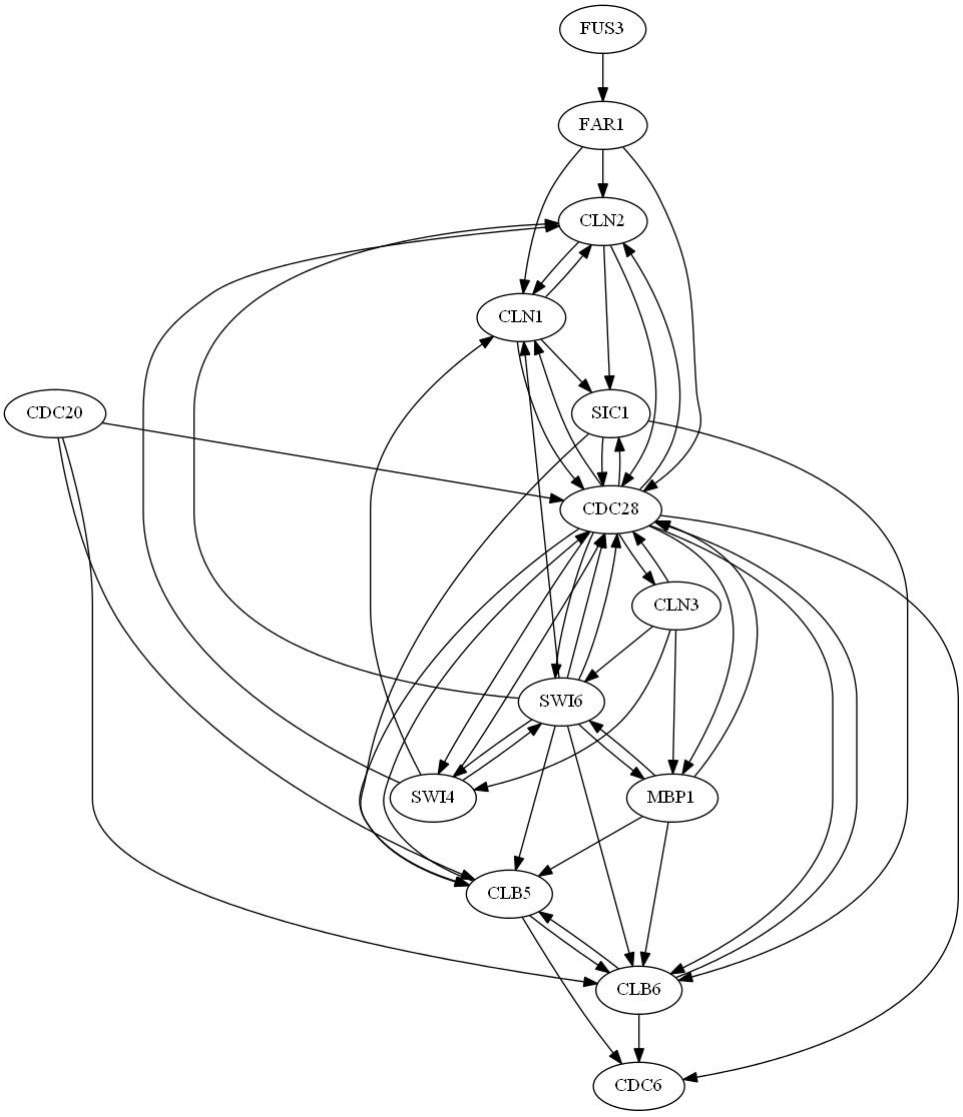

Figure 1.2: An example Gene Regulatory Network extracted from yeast cell cycle pathway

falling into diseased absorbing states.

In this thesis, we have attempted to solve two major problems, controlling gene regulation systems and generating high quality artificial gene expression samples. Controlling a gene regulation system means to control the state of the system by intervening a specific set of genes, namely *action genes*. To illustrate, it may be the case that a laboratory specialist tries to keep deactivated a specific set of genes, which we name as *target genes*, by suppressing the specified action genes. Figure 1.3 shows a subnetwork of *E. coli* network which we automatically extracted by the tool GeneNetWeaver (GNW) [58]. Blue arrows indicate up regulating, red arrows indicate down regulating effects. Genes *cueR* and *modA* are selected as action genes, and gene *modB* is selected as target gene. The goal of the control problem is to control expression value of *modB* by externally activating *cueR* and deactivating *modA*. Observe that the action applied to gene *cueR* up regulates expression value of *cueR*, whereas the action applied to gene *modA* down regulates expression value of *modA*. As a real life biological correspondence of control studies, gene regulation system of melanoma dataset presented in [5] is one of the good examples. It is reported that gene WNT5A has a significant effect on metastasizing of melanoma. Hence, a control policy for the regulation system of melanoma keeping deactivated WNT5A may considerably reduce the metastasis effect of melanoma [5, 12]. We have tried to provide control solutions for keeping WNT5A deactivated in Chapter 3, Chapter 4, and Chapter 5.

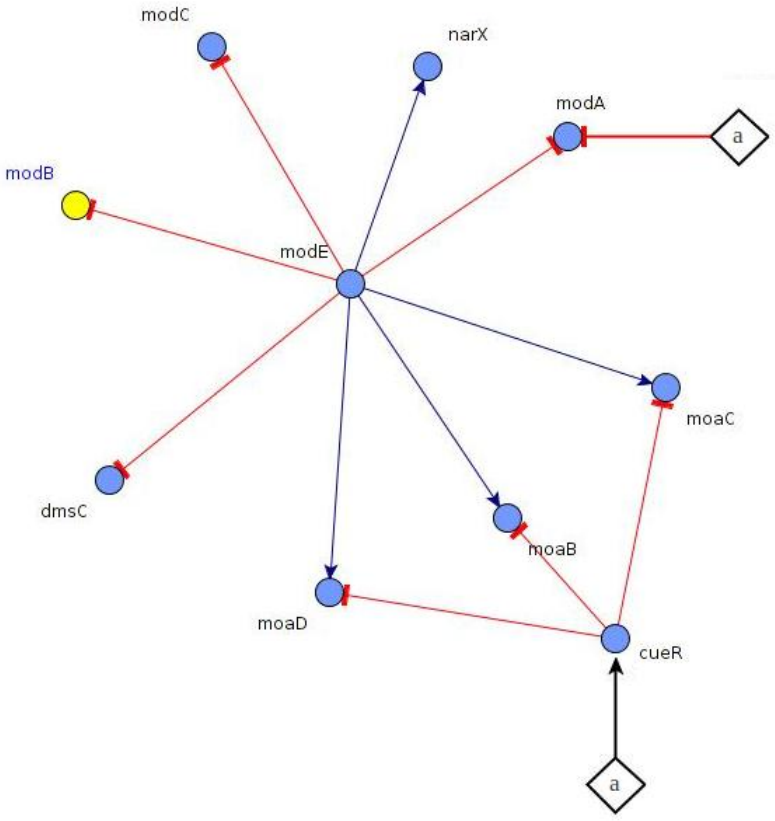

Figure 1.3: Controlling a GRN

We have proposed three solutions for controlling gene regulation systems, one solution for controlling fully observable gene regulation systems, and two solutions for controlling partially observable gene regulation systems. Fully observable gene regulation system means to be able to observe all genes in the regulation system. Observing a gene in a gene regulation system corresponds to be able to measure expression value of that gene during the experiments in the laboratory. For controlling fully observable gene regulation systems, we have proposed to obtain a control policy directly from available gene expression data without making use of any computational model. A control policy is a sequence of actions that should be applied to the gene regulation system so that, at the end, the state of the system will be one of the desirable states. Existing works for this control task firstly construct a computational model from gene expression data, mostly a Probabilistic Boolean Network (PBN), and then try to control the constructed computational model [12, 51, 22]. However, time series gene expression data can actually be interpreted as a series of experiences collected from the gene regulation system. A gene regulation system is not different than any dynamic system who has a transition dynamics between its states, and reacts to applied actions probabilistically, which perfectly fits to the framework of Markov Decision Processes (MDPs) explained in Chapter 2, Section 2.1. Hence, it is highly suitable to think the gene regulation system as an environment, and the specialist in the laboratory as an agent, in which agent applies actions to the selected genes in the regulation system, observes the state transitions in the system, and thereby gathers some experiences from the environment. Hence, in our work, we have treated available gene expression data as if they were generated by applying some actions to the actual gene regulation system, and converted the gene expression data into a series of experiences collected from the environment. Once we have done the conversion of gene expression samples into a series of experiences, we can directly use those experiences to obtain an approximate control policy based on Batch Mode Reinforcement Learning (Batch RL) techniques. Batch RL basically produces generalized and approximate control policies from limited number of available experiences. Thereby, since we are not dealing with any computational model, we are able to benefit from great reduction on time and space requirements. Results show that our proposed Batch RL based method is able to produce approximate policies for regulation systems of several thousands of genes just in seconds; whereas existing studies cannot solve the control problem even for several tens of genes. We have also measured the quality of our produced control policies with respect to the control policies generated by existing works. We have done the comparison by applying the produced policies to a constructed PBN, separately, and checking the probability distribution of the controlled PBNs. We expect to shift probability mass from undesirable states to desirable states. The results show that our generated approximate policies shift the probability mass almost as successfully as the ones generated by existing optimal solutions.

Partially observable gene regulation system, on the other hand, means to be able to observe only a set of genes in the regulation system. The main problem for controlling

partially observable gene regulation systems is that we do not know the actual internal states of the regulation system, but only a set of observations having a probabilistic relationship with the actual internal states. For solving the control problem of partially observable gene regulation systems, we have proposed two methods. The first method skips the internal state estimation phase, and produces approximate stochastic policies mapping probabilistically observations directly to actions. We have interpreted time series gene expression data produced as a sequence of observations from the gene regulation system we want to control, and obtained a generalized and approximate control policy directly from those observations. In order to do this, we benefit from the ideas of Jaakkola et al. (1994), Singh et al. (1994), and Sutton and Barto (1998) (check Section 5.8 and 7.11 for Sutton and Barto (1998)'s book). We have combined Batch RL techniques with Sutton's TD(λ) algorithm (1988) [64], and proposed a novel Batch RL framework for non-Markovian decision tasks, Batch Mode TD(λ). Batch Mode TD(λ) assigns an approximate observation-action value for each possible observation-action pair, and produces stochastic policies based on the ratio of those approximate observation-action values. Since Batch Mode TD(λ) does not deal with any computational model, and skips the internal state estimation of the partially observable environment, it can produce stochastic policies for regulation systems of several thousands of genes in seconds; whereas existing studies cannot solve the control problem even for several tens of genes. To our best knowledge, Batch Mode TD(λ) is the first solution for non-Markovian decision tasks with limited number of available samples. Note that partially observable environment and non-Markovian environment refer to the same concept, and are used interchangeably.

In our second solution for controlling partially observable gene regulation systems, unlike our previous solution Batch Mode TD(λ), we identify the actual internal states of the regulation system we want to control. We have calculated an approximate observation-action value for each possible observation-action pair based on Batch RL methods as we have done in our previous control solutions. Then, we have applied hidden state identification techniques to those approximate observation-action values to identify the actual internal states and their Bayesian relationships with the observations [47, 45, 48, 46]. The idea is that observations having similar observation-action values should be produced from similar internal state dynamics. With respect to the identified internal states, and their identified Bayesian relationships with the observations, we have constructed a Partially Observable Markov Decision Process (POMDP). Results show that our constructed POMDPs are more successful than the previous studies in terms of both solution quality and time requirements.

Here, it is important to note that in all of our control solutions we have assumed the available gene expression data is *times series*. Though the number of time-series datasets are relatively less with respect to the number of steady-state datasets, we believe our proposed control solutions will encourage to produce time series real life gene expression data, and control the gene regulation systems producing those time

series gene expression data.

Since all our proposed control solutions obtain control policies directly from gene expression data, the number of samples in gene expression datasets are one of the determining factors for our solutions. However, gene expression datasets are generally composed of small number of samples, which restricts not only our control solutions, but also the whole genome research since most of the genome research is driven by gene expression data. Hence, in the second part of this thesis, we have proposed a novel multi-model gene expression data generation framework in order to solve the problem of generating high quality artificial gene expression data. We aimed to mitigate low number of samples problem of gene expression datasets. The idea is, although there are many computational models for gene regulation, none of them is able to capture the biological relationships comprehensively. Hence, instead of trying to improve a single computational model, we have combined different computational models into one framework. Thereby, we aimed to benefit from different computational models concurrently, and provide a robust framework for generating high quality artificial gene expression data. We have used four generative models, Probabilistic Boolean Networks (PBNs), Ordinary Differential Equations (ODEs), Hierarchical Markov Model (HIMM), and Genetic Algorithm (GA). In order to generate k samples from our multi-model framework, we firstly sample from each generative model separately and pool them. The pool of the generated $4k$ samples constitutes a rich set of gene expression samples. We evaluate each of the $4k$ samples with respect to three well-defined metrics and rank them multi-objectively. At the last step, we output the top k samples as the best artificial gene expression samples. Results show that our proposed multi-model framework produces samples that are very close to the original gene expression data, and always carry new information with respect to the generated sample sets. Our multi-model framework works always better than any single model it constitutes, and captures biological relationships that cannot be captured by original gene expression datasets.

CHAPTER 2

BACKGROUND

2.1 Markov Decision Processes

A Markov Decision Process (MDP) is a framework composed of a 4-tuple (S, A, T, R), where S is the finite set of states; A is the finite set of actions; $T(s'|s, a)$ is the transition function which defines the probability of observing state s' by firing action a at state s; and $R(s, a)$ is the expected immediate reward received in state s firing action a. The transition function defines the dynamics of the environment, and the reward function specifies the rewards with respect to the state and action configurations [2]. MDPs define discrete time stochastic control processes flowing in a finite set of states with respect to the applied actions by the agents in the environment. An agent can be anything that can act in the environment and observe state of the environment such as a robot or a specialist in a laboratory environment. Figure 2.1 presents a simple MDP with five states and two actions. If the action fired by the agent is a_2 and state of the system is s_1, for example, then the system in Figure 2.1 gets into state s_5 with probability 0.7, and into state s_3 with probability 0.3. Note that probability values should sum up to 1.0 for each state action pairs. Note also that, in Figure 2.1, state s_3 is an absorbing state, in which the agent cannot escape once it gets in.

The ultimate goal of an MDP is to find the optimal control policy π^*, which is a mapping of states to actions, such that for a state s, its associated action a will result in getting the maximum future rewards after passing through state s. The actions producing maximum future rewards are named as optimal actions, and the function mapping states to their optimal actions is named as the optimal control policy, $\pi^* : S \rightarrow A$. The optimal policy can actually be found by searching in policy space in a brute-force manner. For each state, there are $|A|$ actions to apply, and the total number of possible policies are $|A|^{|S|}$. We run each possible policy for large number of steps, compare the sums of the obtained rewards, and output the policy producing the maximum sum of rewards. However, brute-force search in policy space requires exponential search time in terms of the number of states, which would fail even for MDPs with small number of states and actions. For an MDP with 30 states and 2 actions, for example, there are more than 1 billion possible policies, which is

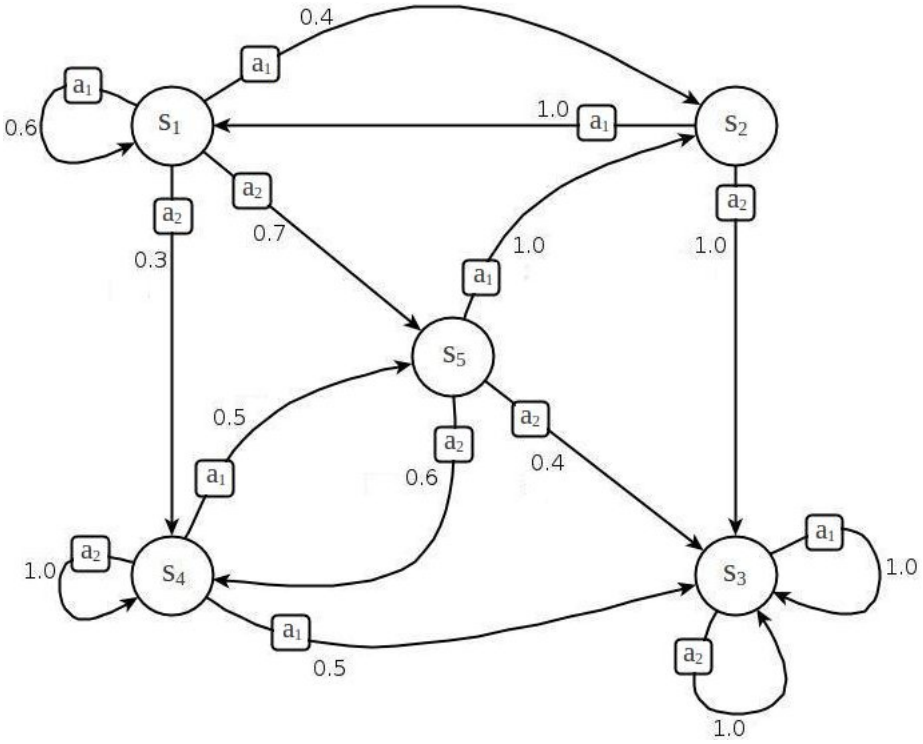

Figure 2.1: A simple Markov Decision Process

highly infeasible to search in [65].

The optimal control policy of an MDP, however, can also be found by applying the Dynamic Programming algorithm, Value Iteration, shown in Algorithm 1. Value Iteration tries to maximize future rewards in terms of expected *return* values, where the return, R_t, is a specific function of rewards. A simple return function can be just summation of the rewards obtained after passing through state s, and applying action a as Equation 2.1 shows [65].

$$R_t = r_{t+1} + r_{t+2} + ... + r_N \tag{2.1}$$

where N is the last step agent takes. Value Iteration keeps an intermediate data structure, namely the state-action function Q; and for each state-action pair, it accumulates expected return values in that Q data structure. By accumulating expected return values iteratively in the Q function, Value Iteration converges to find the optimal state-action function Q^*, which further provides to find the optimal policy π^*. The relationship between the expected return values, $E\{R_t|s_t = s, a_t = a\}$, and the state-action function Q is shown in Equations 2.2 to 2.6 [65].

$$Q(s,a) = E\{R_t | s_t = s, a_t = a\} \tag{2.2}$$

$$= E\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a\} \tag{2.3}$$

$$= E\{r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | s_t = s, a_t = a\} \tag{2.4}$$

$$= r_{t+1} + \sum_{s'} T(s'|s,a)\gamma(E\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | s_t = s, a_t = a\}) \tag{2.5}$$

$$= r_{t+1} + \gamma \sum_{s'} T(s'|s,a) \max_{a'} Q(s,a') \tag{2.6}$$

Note that since Value Iteration is expected to converge, the discount factor γ where $0 \leq \gamma \leq 1$, is introduced for finding the expected return. Equation 2.1 and Equation 2.3 are same except the the discount factor γ. As γ gets close to 1, agent gives more importance to future rewards. As γ gets close to 0, agent gives more importance to immediate rewards. Observe that Equation 2.6 is exactly the same equation Value Iteration applies to update its Q function in Line 7 of Algorithm 1. This equation is called as the Bellman optimality equation. In order to find the optimal policy π^*, Value Iteration iterates over the Bellman equation, and provably converges to find the optimal state-action function Q^* for sufficiently large number of iterations [2, 65]. Once Q^* function is found, it easy to find the the optimal policy π^* based on Equation in Line 11 of Algorithm 1.

Algorithm 1: Value Iteration

```
1  Input: discount factor γ, stopping criteria ε
2  Q ← 0, q ← ε
3  while |q − Q| ≥ ε do
4  |   q ← Q
5  |   for s ∈ S do
6  |   |   for a ∈ A do
7  |   |   |   Q(s,a) ← r(s,a) + γ Σ_{s'} T(s'|s,a) max_{a'} Q(s,a')
8  |   |   end
9  |   end
10 end
11 π*(s) = arg max_a Q*(s,a)
12 Output: π*
```

The complexity of Value Iteration is $|A| \times |S|^2$, which is linear in terms of number of actions, and quadratic in terms of number of states [42]. Note that $\max_{a'} Q(s,a')$ stands for the value of state s. It is found by setting the value of the maximizing action as Equation 2.7 shows [65].

$$V(s) = \max_{a'} Q(s, a') \tag{2.7}$$

2.2 Reinforcement Learning

Reinforcement Learning (RL) is a framework to find the optimal state-action function, Q^*, without using the model of the environment, which is $T(s'|s, a)$, the transition function, and $R(s, a)$, the reward function [65]. Agent in the environment does not know anything about how the environment will react to its actions, and successively interacts with the environment. The idea is to find the optimal state-action function Q^* based on the reward and state values observed during the interaction with the environment [32]. Figure 2.2 depicts the RL framework.

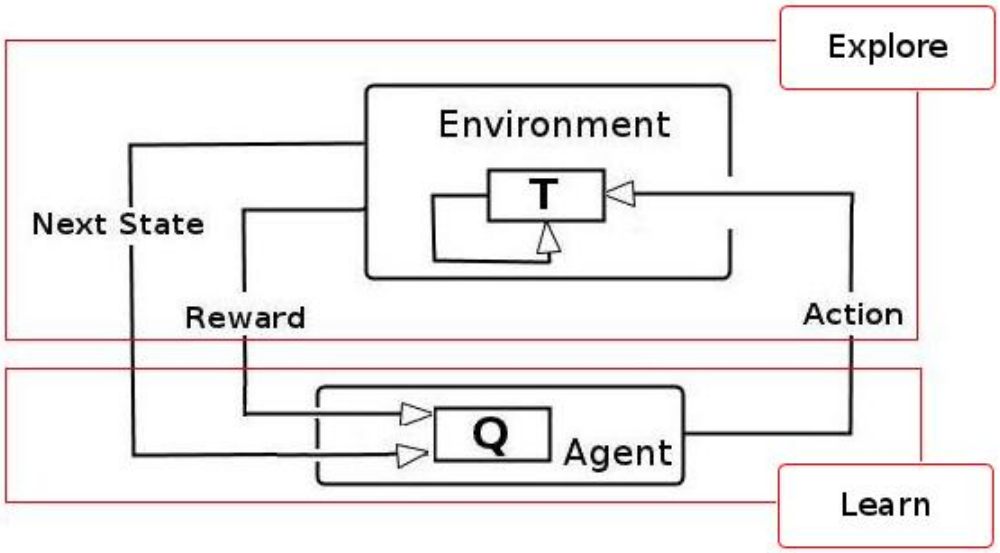

Figure 2.2: Reinforcement Learning framework

Just like Value Iteration, Q-Learning tries to find the optimal policy π^* by maximizing the expected return values. Again, there is a state-action function Q which keeps the expected return values for each state-action pairs. Since agent does not know the transition function $T(s'|s, a)$ of the environment in RL setting, it cannot calculate return values by expectation over all possible next states as in Equation 2.2 to 2.6. Instead, RL algorithms maintain return values by functions that depend only on observed future rewards. One simple return function is to take the average of the obtained rewards after experiencing a specific state-action pair shown in Equation 2.8.

$$R_t = \frac{r_{t+1} + r_{t+2} + ... + r_N}{N - t} \tag{2.8}$$

where N is the last step agent takes. Then, the relationship between the Q function

and the return values R_t is as shown in Equations 2.9 to 2.15 [65].

$$
\begin{aligned}
Q_N(s,a) &= E\{R_t | s_t = s, a_t = a\} && (2.9)\\
&= \frac{r_{t+1} + r_{t+2} + ... + r_N}{N-t} && (2.10)\\
&= \frac{r_{t+1} + r_{t+2} + ... + r_{N-1}}{N-t} + \frac{r_N}{N-t} && (2.11)\\
&= \frac{(N-t-1)Q_{N-1}}{N-t} + \frac{r_N}{N-t} && (2.12)\\
&= Q_{N-1} - \frac{Q_{N-1}}{N-t} + \frac{r_N}{N-t} && (2.13)\\
&= Q_{N-1} + \frac{1}{N-t}[r_N - Q_{N-1}] && (2.14)\\
&= Q_{N-1} + \alpha[r_N - Q_{N-1}] && (2.15)
\end{aligned}
$$

where Q_N is the evaluation of the Q function upto the N^{th} step with respect to the observed state s and applied action a at time t. Equations 2.9 to 2.15 show how expected return values are accumulated in the Q function incrementally. Equation 2.15 forms the basic update equation for RL algorithms. The general form is $Q_{new} = Q_{old} + LearningRate[Target - Q_{old}]$. This is actually a simple learning equation in which $[Target - Q_{old}]$ stands for the error we have in the old Q estimation and $LearningRate$ determines how much we will get close to the $Target$ value at each iteration. With respect to how we assigned the $Target$ and $LearningRate$ components, the RL algorithm shapes itself. For example, Monte-Carlo value estimation algorithm sets the $Target$ as the discounted sum of rewards, $\sum_{l=t+1}^{T} \gamma^{l-t} r(s_l, a_l)$; Q-Learning algorithm sets the $Target$ as the immediate reward plus the expected next state value, $r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a')$; and Sarsa algorithm sets the $Target$ as immediate reward plus the next state-action value, $r_{t+1} + \gamma Q(s_{t+1}, a_{t+1})$ [65].

Algorithm 2 presents Watkin's Q-Learning algorithm (1989) [69]. Observe that Q-Learning has a lot common with Value Iteration. Both algorithms uses the same data structure, namely the Q function, and iteratively updates it to find the optimal state-action Q^* and the optimal policy π^*. The only and most important difference, however, is that while Value Iteration applies *full* backup for updating the Q function, Q-Learning applies *sample* backup. Value Iteration takes all possible next states into account for updating the Q function as Equation in Line 7 of Algorithm 1 and 2.6 show. Q-Learning, on the other hand, considers only the value of the next state $\max_{a'} Q(s', a')$ for updating the Q function since in RL setting agent is assumed to know nothing about the transition function $T(s'|s, a)$ of the environment. However, as it is the case for Value Iteration, Q-Learning provably converges to produce the optimal state-action function Q^* as well. What is done is to start with a randomly selected state as shown in Line 5 and get into the inner update loop for updating the Q. At each iteration, Q-Learning firstly decide on the action that the agent should

take, applies it to the environment, observes the returned reward and next state values, and lastly updates its Q function based on the equation in Line 14. The algorithm continues by setting the current state s as the next state s'.

One important point about Q-Learning is the action-selection mechanism. Most of time, actions are selected as the ones maximizing the state-action function for current state s, i.e., $\arg\max_a Q(s,a)$. However, this way has always the risk of falling into an absorbing state due to an intermediate local maximum. In order to avoid from getting stuck to a local maximum, at some points, agent selects its action randomly instead of selecting the maximizing ones. One of the action selection mechanisms is ϵ-greedy method. ϵ-greedy defines a small probability value ϵ. At each iteration, agent guesses a random floating number between $[0,1]$, μ value in Line 7 of Algorithm 2. If μ is less than the ϵ, agent selects its action randomly. Otherwise, agent follows its usual way to select an action maximizing its state-action function for the current state. ϵ-greedy action selection mechanism is shown between Lines $7-12$ of Algorithm 2.

Algorithm 2: Q-Learning

1 **Input:** discount factor γ, learning rate α, ϵ, stopping criteria ε,
2 episode number E
3 $Q \leftarrow 0, q \leftarrow \varepsilon$
4 **while** $|q - Q| \geq \varepsilon$ **do**
5 $iter \leftarrow 1, q \leftarrow Q, s \leftarrow randomStateSelection$
6 **while** $iter \leq E$ **do**
7 $\mu \leftarrow randomFloat[0,1]$ // ϵ-greedy action selection
8 **if** $\mu < \epsilon$ **then**
9 $a \leftarrow randomActionSelection$
10 **else**
11 $a \leftarrow \arg\max_a Q(s,a)$
12 **end**
13 Take action a, observe $R(s,a)$ and s'
14 $Q(s,a) = Q(s,a) + \alpha(r(s,a) + \gamma \max_{a'} Q(s',a') - Q(s,a))$
15 $s \leftarrow s'$
16 $iter \leftarrow iter + 1$
17 **end**
18 **end**
19 $\pi^*(s) = \arg\max_a Q^*(s,a)$
20 **Output:** π^*

2.3 Batch Mode Reinforcement Learning

Batch Mode Reinforcement Learning (Batch RL) is an extension of classical RL. In Batch RL, the learner directly takes several number of experience tuples that are already collected from the environment. The experience tuples can be collected arbitrarily, even randomly in the exploration stages of the learner [38]. The main idea is to use these limitedly available experience tuples in batch to obtain an approximate and generalized state-action function. Hence, unlike classical RL employing exploration and learning phases consecutively as shown in Figure 2.2, Batch RL clearly separates the exploration and learning phases as Figure 2.3 shows [38, 8, 21]. Learner collects a set of experience tuples without updating its Q function, and then updates the Q function in batch. Note that an experience tuple is a 4-tuple (s, a, s', r), where s is the current state, a is the current action, s' is the next state and r is the immediate reward. Experience tuples are the formal expressions of one step experiences gathered from the environment that the agent interacts with.

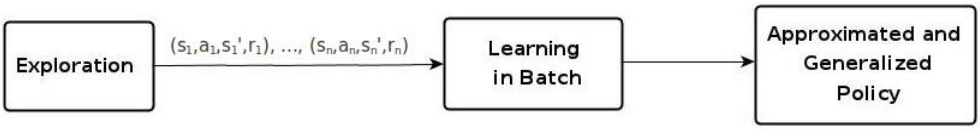

Figure 2.3: Batch RL framework

There are several Batch RL algorithms. The Experience Replay algorithm presented in [41] assumes the learner experiences again and again what it had experienced before. Hence, it can only know the state-action values of the state action pairs that it had visited before, and have limited application. The Kernel-Based RL in [50] introduces kernel functions to get an approximate and generalized state-action function. Based on the knowledge of the values for experienced state-actions, it finds the values for unexperienced state-actions by applying some kernel functions. Taking the average of the state-action values of the most similar three experience tuples would be an example of a kernel-function application. The Fitted Q Iteration (FQI) algorithm presented in [21], on the other hand, converts the Batch RL problem into a supervised learning problem. The main idea is that it is actually a supervised learning problem to find an approximate and generalized state-action function mapping all possible state action pairs into their state-action values. Since there are different supervised learning algorithms, FQI provides different ways to solve Batch RL problem for different problem domains.

2.3.1 Least-Squares Fitted Q Iteration

In this thesis, we have extensively used one of the Batch RL methods, Least-Squares Fitted Q Iteration (LSFQI) algorithm presented in [8, 21]. All of our proposed solutions for controlling gene regulatory networks, somehow, are making use of LSFQI algorithm.

LSFQI employs parametric approximation to obtain an approximate and generalized state-action function $\hat{Q}$. $\hat{Q}$ is parameterized by an n-dimensional vector Θ, where every parameter vector Θ corresponds to a compact representation of the approximate state-action function $\hat{Q}$ as Equation 2.16 shows.

$$\hat{Q}(s,a) = [F(\Theta)](s,a) \tag{2.16}$$

The calculation of $[F(\Theta)](s,a)$, on the other hand, is done by utilization of feature values as Equation 2.17 shows.

$$[F(\Theta)](s,a) = [\phi_1(s,a), ..., \phi_n(s,a)]^T \cdot [\Theta_1, ..., \Theta_n] \tag{2.17}$$

where ϕ_i stands for a single feature specific for state s, and the action a, Θ_i stands for the parameter corresponding to the i^{th} feature, and Θ is the parameter vector of Θ_i's. LSFQI algorithm, iteratively train the parameter vector Θ with respect to the defined feature values and calculated state-action values for each experience tuples by using least-squares linear regression. Note that the number of parameters in LSFQI is same as the number of features defined. The overall algorithm is shown in Algorithm 3.

Algorithm 3: Least-Squares Fitted Q Iteration

Input: discount factor γ,
  experience tuples $\{(s_i, a_i, r_i, s'_i) | i = 1, ..., N\}$
$j \leftarrow 0$, $\Theta_j \leftarrow 0$, $\Theta_{j+1} \leftarrow \epsilon$
while $|\Theta_{j+1} - \Theta_j| \geq \epsilon$ **do**
  for $i = 1...N$ **do**
    $T_i \leftarrow r_i + \gamma \max_{a'}[F(\Theta_j)](s'_i, a')$
  end
  $\Theta_{j+1} \leftarrow \Theta^*$,
    where $\Theta^* \in \arg\min_{\Theta} \sum_{i=1}^{N} (T_i - [F(\Theta)](s_i, a_i))^2$
  $j \leftarrow j + 1$
end
Output: Θ_{j+1}

We begin with an initial parameter vector Θ_0. At each iteration, LSFQI firstly assigns a target state-action value T_i for each experience tuple. This is achieved by summing immediate reward r_i and the discounted future reward $\gamma \max_{a'}[F(\Theta_j)](s'_i, a')$ for each experience tuple. Note that this equation is same as Equation in Line 14 of Algorithm 2 with learning rate (α) as 1. Then, those target values and available feature values are used to train the parameter vector Θ by using least squares linear regression, which provides the next parameter vector Θ_{j+1}. The algorithm continues with the next iteration by using the same feature values but the refined parameter vector. Once

the parameter vector is converged, the algorithm outputs the parameter vector which can be used to find the approximate state-action function based on the Equation 2.17. Then, it is easy to find the approximate infinite horizon control policy by taking the minimizing action for each state as Equation 2.18 shows.

$$\pi(s) = \arg\max_{a'}[F(\Theta_{j+1})](s, a') \tag{2.18}$$

2.4 Partially Observable Markov Decision Processes

A Partially Observable Markov Decision Process (POMDP) is a 6-tuple (S, O, A, T, Ω, R), where S is the finite set of states; O is the finite set of observations; A is the finite set of actions; $T(s'|s, a)$ is the transition function which defines the probability of observing state s' by firing action a at state s; $\Omega(o|s)$ is the observation function which defines the probability of observing observation o at state s; and $R(s, a)$ is the immediate reward received in state s firing action a [9]. The transition function and the reward function are same as the ones in Markov Decision Process (MDP). The Observation function, on the other hand, determines the Bayesian relationship between the actual internal states of the system, and the observations that the agent in the environment observes.

POMDPs are actually generalizations of Markov Decision Processes (MDPs). MDP framework models fully observable environments. That is, an agent moving in the state space of an MDP framework always gets complete state information about where it is in the state space. In POMDP framework, however, agent never knows where it is in the state space precisely. Agent only gets some partial state information returned by the environment that is probabilistically related to the actual internal state dynamics of the environment, which we name as observations. Figure 2.4 depicts a POMDP with three states, three observations and two actions. Agent can only observe the observations with respect to the probabilities it is assigned by the observation function Ω. To illustrate, once an agent applies action a_2 at state s_1 in the POMDP of Figure 2.4, even if it makes a transition from the s_1 to s_2 with probability 1.0, the agent is able to observe only the observations that can be produced from the s_2 which are o_2 and o_3. Enclosed in different boxes in Figure 2.4, in a POMDP setting, internal dynamics of the environment including state transitions are unobservable to the agent.

In a POMDP framework, the aim is to find the optimal control policy maximizing expected discounted sum of rewards just as in the MDP framework. Here, the basic challenge is that the environment is not Markovian in terms of the observations. That is, the environment may seem unchanged after an action application, though its internal state has actually changed. For example, making transition from s_1 to s_2 by applying action a_2 in the POMDP of Figure 2.4 may result in observing the same

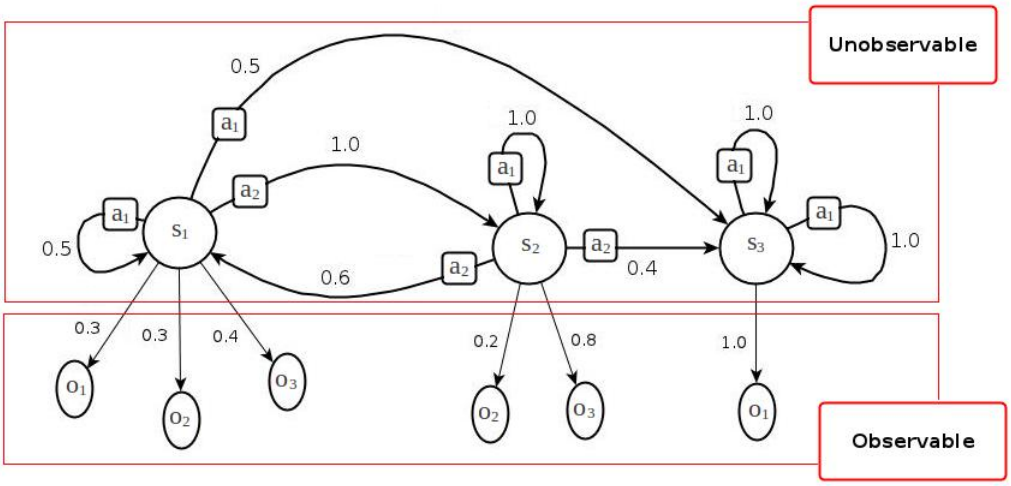

Figure 2.4: Partially Observable Markov Decision Process

observation o_2, although the agent makes a transition from s_1 to s_2. Therefore, we cannot keep a Q function mapping state-action pairs into their expected reward values, and obtain a policy based on the states in the POMDP, which is done by the Value Iteration shown in Algorithm 1. What can be done, however, is that agent can try to build a control policy not over the state-action pairs, but over *belief state* & action pairs. Belief state is a probability distribution over the set of states representing the belief of the agent on where it is in the state space. For the POMDP in Figure 2.4, an example belief state can be [0.2 0.8 0.0] implying that the agent is in s_1 with probability 0.2, in s_2 with probability 0.8, and certainly not in s_3 due to 0.0 probability. In order to find an optimal control policy in a POMDP setting, we actually solve the MDP whose state space is the space of the belief states of the POMDP. That means, we have an MDP with continuous state space. Hence, instead of keeping a value in a Q as in MDP setting, we keep an n dimensional vector, a hyperplane for each belief state & action pairs, where n is the number of states. Because we multiply belief state vector element by element with the reward values of each state separately. That means the hyperplanes for belief state & action pairs are linear with respect to the belief states [9]. Therefore, instead of keeping a discrete Q function, we keep a set of hyperplanes corresponding for each belief state & action pairs. Figure 2.5 presents an example for a two state, two actions belief state POMDP [57, 9]. The x-axis shows the probability value of state 1 in the belief state, and the y-axis shows the corresponding belief state & action value for each action in the POMDP framework. Here, the value of doing action 1 in state 1 is 3, in state 2 is 1. The value of doing action 2 in state 1 is 1, in state 2 is 2. If probability of state 1 is zero, i.e., the agent is in state 2, the value of doing action 1 is 1. If probability value of state 1 is 1, i.e., the agent is in state 1, the value of doing action 1 is 3. That's why the red line presenting the value of applying action 1 in Figure 2.5 goes from 1 to 3. For the belief states between [0 1] and [1 0], the value of doing action 1 is just the expectation of the reward values with respect to the belief state as Equation 2.19 and the lines in Figure 2.5 shows. The belief state &

action values shown in Figure 2.5 corresponds to horizon 1 values, indeed. Hence, the optimal action for horizon 1 for the POMDP of Figure 2.19, is 2 for belief states having less than ~ 0.32 probability value for state 1, and is 1 for belief states having more than ~ 0.32 probability value for state 1. In order to calculate belief state & action values for horizon 2, we will apply the same procedure we have done for horizon 1, indeed. However, instead of using directly the $R(s, a)$ for calculating the hyperplanes, we will use the already calculated horizon 1 values [57, 9].

$$\rho(b, a) = \sum_{s} b(s) \cdot R(s, a) \tag{2.19}$$

where b is the belief state.

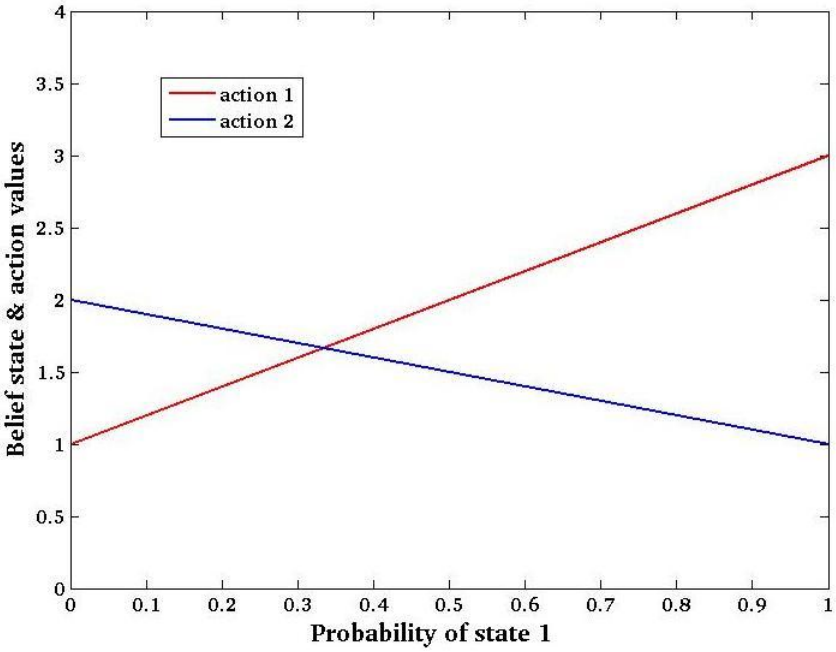

Figure 2.5: Belief state & action values

CHAPTER 3

CONTROLLING GENE REGULATORY NETWORKS: FULL OBSERVABILITY

3.1 Introduction

The control problem for GRNs is defined as controlling the state of the regulation system through interventions of a set of genes. That is, we apply a series of actions to some pre-selected set of genes, and expect the regulation system not to fall into undesirable states. There are several studies in the area of controlling GRNs. They all model gene regulation as a Probabilistic Boolean Network (PBN) and attempt to identify the best intervention strategy over the constructed PBN [60]. The study described in [12] tries to find an optimal finite horizon intervention strategy for PBNs so that at the ultimate horizon, the system achieves the highest probability of being in a desirable state. The study in [51] finds optimal infinite horizon intervention strategy by formulating the control problem as a Markov Decision Process (MDP) and then solving the problem with the help of the Value Iteration algorithm. They showed that the optimal policy can shift the probability mass from undesirable states to desirable ones in the constructed PBN. The study in [22, 23] applies Reinforcement Learning (RL) techniques to obtain an approximate control policy of the constructed PBN. Note that an infinite horizon control policy corresponds to a control policy mapping states into their optimal actions without depending on the time step that the action applied. To illustrate, a finite horizon policy assigns optimal actions to the states with respect to the time steps that the actions are applied. An optimal action found by a finite horizon policy may not be the optimal action for an infinite horizon policy since it may result in getting into bad states in future. Similarly, an optimal action found by an infinite horizon policy may not be the optimal action for a finite horizon policy since it may result in getting into bad states unnecessarily in the time window of the specified finite horizon [32].

The basic and most important problem with the existing solutions for controlling GRNs is that none of them can solve the control problem for systems with more than several tens of genes due to their exponential time and space requirements. The finite horizon study of [12] builds a finite horizon MDP, and the infinite horizon study of [51] builds an

infinite horizon MDP over the PBN. Both require exponential time and space in terms of the number of genes. A 30-gene system, for example, requires dealing with more than 1 billion states which is highly infeasible even to keep in the memory. Although the study in [22] proposes an approximate algorithm that runs in polynomial time, it again requires exponential space since they explicitly keep a state-action function Q. Besides, their approximate solution still requires to construct a PBN, which already takes $O(d^k \times n^{k+1})$ time and space, where n is the number of genes, k is the maximum number of predictor genes and d is the discretization level of gene expression samples [60]. The study in [22] reports that the construction of the PBN for their 10-gene system takes more than 3 days for $k = 3$ and $d = 2$. Moreover, for the same 10-gene system, it requires almost one hour for their proposed algorithm to produce sufficiently good policies, which is a long time compared to our method.

In this chapter, we propose a novel method to control GRNs making use of Batch Mode Reinforcement Learning (Batch RL) approach. Batch RL provides approximate infinite horizon control policies without requiring the model of the environment. That is, it tries to obtain a generalized control policy based on limitedly available experience tuples collected from the environment. Our idea is that time series gene expression data can actually be interpreted as a sequence of experience tuples collected from the environment. Each sample represents the state of the system, and successive state transitions demonstrate system dynamics. Therefore, instead of modeling gene regulation as a PBN and obtaining a control policy over the constructed PBN, we directly use the available gene expression samples to obtain an approximate control policy for gene regulation using Batch RL. Using this method, we are able to benefit from the great reduction on both time and space since we get rid of the most time consuming phase, inferring a PBN out of gene expression data and running it for control. Figure 3.1 depicts the two alternative solution methods, where the flow in the box summarizes our proposal. The results show that our method can find policies for regulatory systems of several thousands of genes just in several seconds. Interestingly, the results also show that our approximate control policies have almost the same solution quality compared to that of the existing PBN-based optimal solutions. This means our method is not only much faster than the previous solution alternatives but also provides almost the same solution quality.

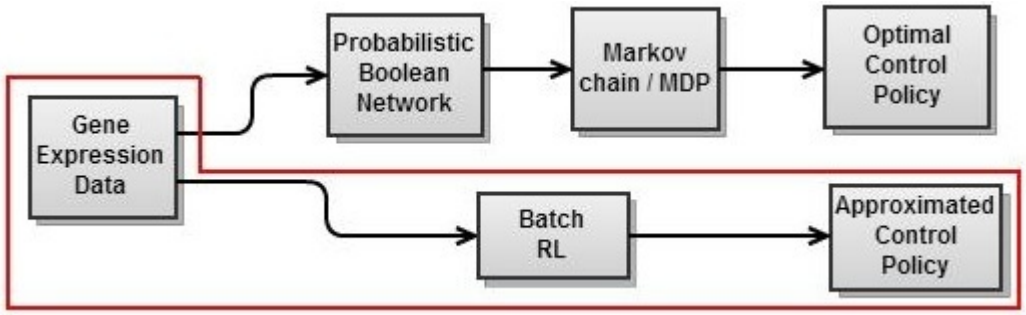

Figure 3.1: Flowchart of control solutions

The rest of this chapter is organized as follows. Section 3.2 describes our proposed method for controlling GRNs. Section 3.3 shows the experimental evaluations of our method and Section 3.4 concludes with a discussion.

3.2 Batch RL for Controlling GRNs

This section describes our proposed method for solving the GRN control problem. We have used the LSFQI algorithm explained in Section 2.3.1. Figure 3.2 shows the block diagram of our proposed method. First, we convert gene expression data, which is sampled from the gene regulation system that we want to control, into a series of experience tuples. Then, we calculate feature values for each experience tuple to use them in the LSFQI algorithm. Note that as Equation 2.17 shows, feature values depend only on the current state and action values of each experience tuples. Therefore, they do not change over the iterations of LSFQI shown in Algorithm 3. Hence, we calculate them once for each available experience tuples beforehand, and let LSFQI use them. Lastly, we invoke the LSFQI algorithm given in Algorithm 3, and obtain the approximate control policy. Thereby, we obtain a control policy for a gene regulation system directly from the gene expression data without making use of any computational model. In the following subsections, we will describe how we convert the gene expression samples into experience tuples and what features we used with the Batch RL algorithm.

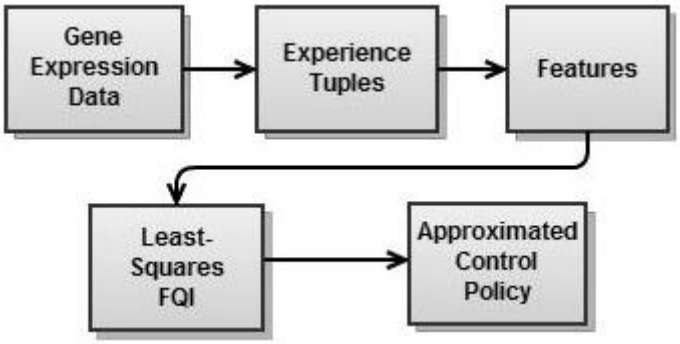

Figure 3.2: Batch RL for controlling GRNs

3.2.1 Experience Tuples

This section describes how we converted gene expression data into a series of experience tuples, which is one of the most critical steps of our solution. An experience tuple is a 4-tuple (s, a, s', c), where s is the current state, a is the current action, s' is the next state and c is the immediate cost. It represents one-step state transition in the environment. Here, we explain how each of the four elements of each experience tuple is obtained from the gene expression data. Note that instead of associating reward

values for the desirable states, in our method we have associated cost values for undesirable states as presumed by previous studies [12, 51]. Instead of maximizing reward, this time we will try to minimize the cost [65].

States: As all previous studies for controlling GRNs, the state of a GRN is defined by the discretized form of the gene expression sample itself [12, 51, 22]. Hence, the i^{th} and $(i+1)^{th}$ gene expression samples constitute the current state s and the next state s' values for the i^{th} experience tuple. Similar to most of the previous studies, we have used binary discretization [12, 51, 22]. Note that there are 2^n possible states if the number of genes is n.

Actions: The action semantics for a gene regulation system is mostly implemented through reversing the value of a specific gene or a set of genes, i.e., changing its value from 0 to 1 or 1 to 0 [12, 51, 22]. Those reversed genes are named as *input genes* and should be specified in the context of the control problem. If the value of an input gene is reversed, the action is assumed as 1, if it is left as it is the action is assumed to be 0. Hence, there are 2^k distinct actions given k input genes. In order to obtain the action values from the gene expression samples, we have checked the absolute value of the difference between the values of the input genes in the successive gene expression samples. For a regulation system of six genes, for example, let the gene expression sample at time t be 101001 and at time $t+1$ be 011000. If the input genes are the 2^{nd} and 5^{th} genes, the action value, i.e., a in the experience tuple, at time t is 10 in binary representation since the 2^{nd} gene has changed its value while the 5^{th} gene has not.

Costs: The only remaining values to be extracted from the gene expression samples is the cost values, i.e., c in the experience tuples. Costs are associated with the goal of the control problem. The goal can be defined as having the value of a specific gene as 0 as in [51], or as reaching to a specific basin of attractors in the state space as in [7]. If the state of the regulation system does not satisfy the goal, it is penalized by a constant value. Moreover, applying an action for each input gene also has a relatively small cost to realize it. So, the cost function can be defined as follows:

$$cost(s,a) = \begin{cases} 0 + n \times c & \text{if goal(s)} \\ \alpha + n \times c & \text{if } \neg \text{ goal(s)} \end{cases} \tag{3.1}$$

where α is the penalty of being in an undesirable state, n is the number of input genes whose action value is 1, and c is the cost of action to apply for each input gene.

3.2.2 Features

This section describes the features that are built from the experience tuples obtained from the gene expression samples. In our study we have defined three feature sets,

which we name as State Features, Gaussian Features, and Distance Features.

State Features: In the GRN domain, state values are composed of the discretized forms of gene expression samples, hence they provide a deep insight and rich information about the characteristics of the GRN. Based on this fact, we decided to use the current state values of the experience tuples, i.e., the discretized gene expression samples itself, directly as features, as the first feature set. That is, for each experience tuple, there are exactly as many features as the number of genes and feature values are equal to the binary discretized gene expression values, which can be formulated as below.

$$\phi_i(s) = \begin{cases} 0 & \text{if } s(i) == 0 \\ 1 & \text{if } s(i) == 1 \end{cases} \tag{3.2}$$

where $0 \leq i \leq n$, n is the number of genes and ϕ_i is the i^{th} feature in the feature vector and it is equal to the expression value of the i^{th} gene in the discretized gene expression sample. So, for a 6-gene regulation system, a state having binary value as 101011 has its feature vector same as 101011.

Gaussian Features: As the second feature set, we have used Gaussian kernel as proposed by [8]. Gaussian kernel function is shown in Equation 3.3.

$$\phi_i(s, s_i) = exp(-\frac{|s - s_i|^2}{2\sigma^2}) \tag{3.3}$$

where $0 \leq i \leq m$, m is the number of samples and ϕ_i is the i^{th} feature in the feature vector. For each experience tuple, we have applied Gaussian kernel to all the other experience tuples and set it as the feature ϕ_i. Hence, this time there are exactly as many features as the number of samples, and each has the value of the Gaussian kernel function applied to the current state s and the i^{th} state s_i.

Distance Features: As the third feature set, we have defined four features manually. Firstly, we have used the Euclidean distance between the state of the current experience tuple and the zero state, which is the state having 000...0 value in binary, as shown by Equation 3.4.

$$\phi_1(s) = \sqrt{s^2 + 0} \tag{3.4}$$

Secondly, we have used the Euclidean distance between the state of the current experience tuple and the one state, which is the state having 111...1 value in binary, as shown by Equation 3.5.

$$\phi_2(s) = \sqrt{s^2 + (2^n - 1)} \tag{3.5}$$

where n is the number of genes. Thirdly, we have taken the mean of the index values of the 1 bits in the binary state value of the current experience tuple. Lastly, we have taken the mean of the index values of the 0 bits in the binary state value of the current experience tuple. Note that for the third and forth feature values, we assumed the first bit has index value as 1. Through defining those four feature values, we aim to be able to define the characteristics of each experience tuple.

Note that as suggested in [8], we have used different parameter vectors for each possible actions, therefore the action does not affect the feature values in Equation 3.2 to 3.5.

3.3 Experimental Evaluation

This section describes the experimental evaluation of our proposed method for controlling gene regulation systems in terms of its solution quality. Firstly, we will compare solution qualities of our method and the previous work of [51] on controlling GRNs with respect to melanoma and yeast cell cycle microarray datasets [5, 63]. Secondly, we will validate our method on a large scale gene regulation system. Lastly, we present the time requirement of our method, and present how our method is efficient to solve control problems of several thousands of genes just in seconds.

3.3.1 Melanoma Application

This section describes the application of our method to melanoma dataset presented in [5]. Melanoma dataset is composed of 8067 genes and 31 samples. It is reported that the WNT5A gene is highly discriminating factor for metastasizing of melanoma, and deactivating the WNT5A significantly reduces the metastatic effect of WNT5A. Hence, a control strategy for keeping the WNT5A deactivated may mitigate the metastasis of melanoma [5, 12, 51, 22].

We have compared our method with the previous infinite horizon solution for the GRN control problem presented in [51] in terms of their solution qualities. Hence, our experimental settings are the same as that of [51]. We considered a seven-gene subset of the complete melanoma dataset, which are WNT5A, pirin, S100P, RET1, MART1, HADHB, and STC2, in order, i.e., WNT5A is the most significant bit and STC2 is the least significant bit in the state values. We have selected the 2^{nd} gene, pirin, as the input gene. Therefore, there are two possible actions defined in the control problem, reversing the value of pirin, $a = 1$, or not reversing it, $a = 0$. We also set the cost of applying an action as 1. The goal objective is to have WNT5A deactivated, its

expression value as 0. We set penalty of not satisfying the goal as 5. Hence, the cost formulation mentioned in Section 3.2.1 can be realized as follows:

$$cost(s,a) = \begin{cases} 0 & \text{if goal(s) and a} = 0 \\ 1 & \text{if goal(s) and a} = 1 \\ 5 & \text{if } \neg \text{ goal(s) and a} = 0 \\ 6 & \text{if } \neg \text{ goal(s) and a} = 1 \end{cases} \tag{3.6}$$

Note that for states $[0-63]$ WNT5A has the value of 0, and for the remaining states $[64-127]$ WNT5A is 1 since WNT5A is the most significant bit in the state values. Hence, the states $[0-63]$ are desirable while states $[64-127]$ are undesirable. We also set our discount factor in the Algorithm 3 as 0.9.

Based on these experimental settings, we obtained an approximate control policy from our proposed method and compared it with the optimal policy obtained by the method proposed in [51]. The following subsections show the results for the three features sets we defined in Section 3.2.2, State Features, Gaussian Features, and Distance Features, respectively.

3.3.1.1 State Feature Results

This section presents the results for our proposed method adapting Least-Squares Fitted Q Iteration (LSFQI) to control gene regulation based on the first feature set we defined, namely State Features. Figure 3.3 shows the average costs over the iterations of LSFQI when we run LSFQI with the control policy it produced over the iterations and without any control, i.e. choosing the action value as always 0. We see that average cost is always less with control policy than without control. We also see that average cost converges to its ultimate value at the 84^{th} iteration.

Figure 3.4 shows the ultimate cost values for each state after the convergence again with and without control policy. We see that cost values are less with control policy than without control. We also see that costs of the desirable states, the states between $0-63$ are generally less than the costs of the undesirable states, the states between $64-127$, which is also consistent with the definition of the problem and our solution. Note that, these cost values are approximate cost values, which is the reason for having negative cost for some states. Because, without approximation it is impossible to obtain a negative cost with the cost formulation we defined in Equation 3.6.

Figure 3.5, on the other hand, compares the two policies obtained from our LSFQI based method and the previous work of [51] on the same plot. We observe that the two policies are almost the same, which actually proves the well-behavior of our LSFQI based method. Because our method computes a control policy directly from gene expression data without making use of any computational method, such as PBN.

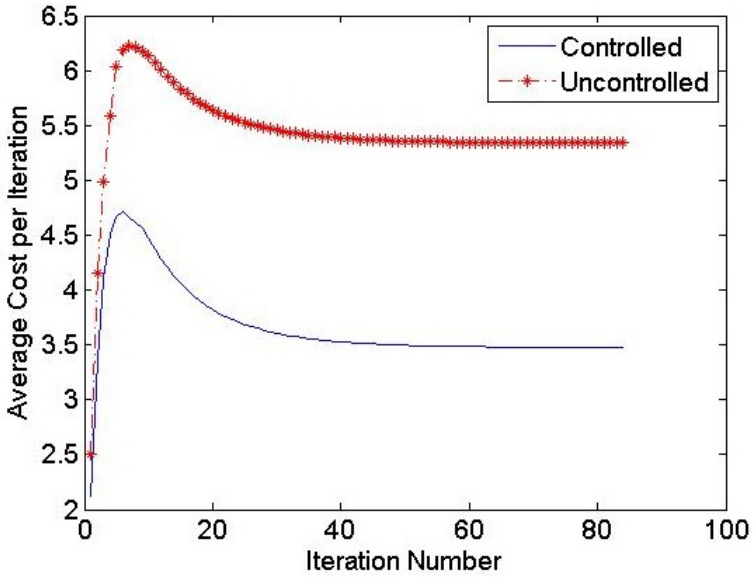

Figure 3.3: Average cost over the iterations of LSFQI for State Features

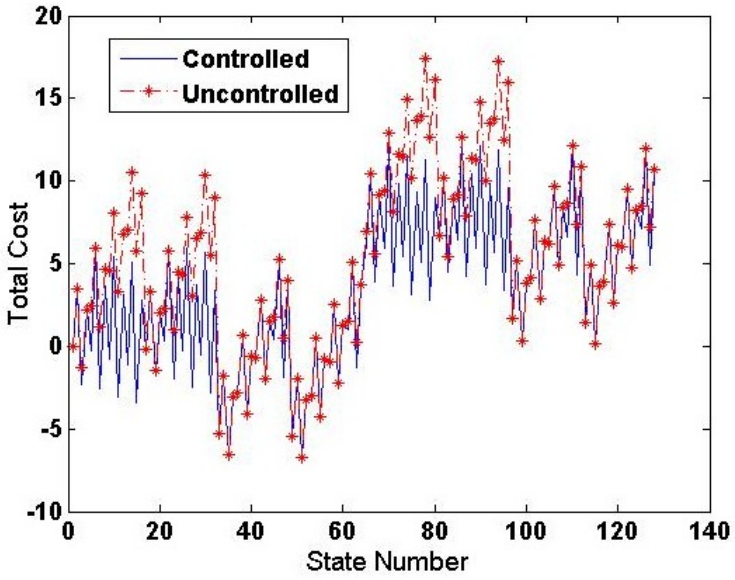

Figure 3.4: Value function after convergence for State Features

However, the policy our proposed method produced is almost same as the one produced by the PBN-based Value Iteration method of [51].

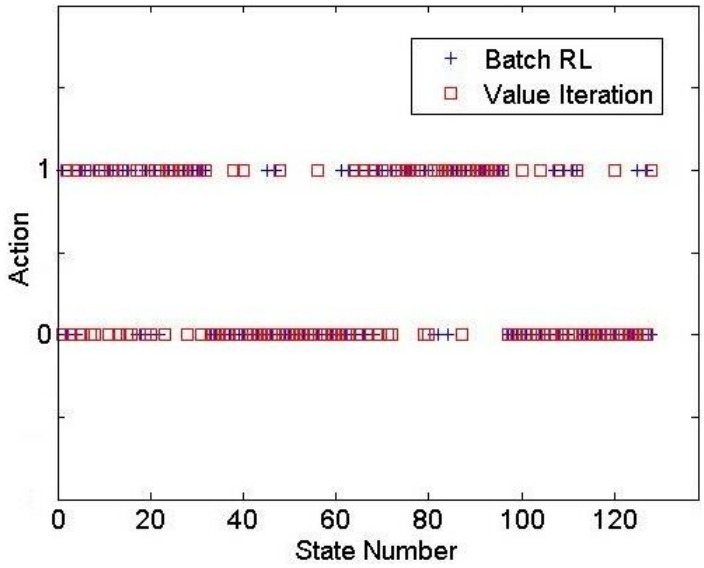

Figure 3.5: Policy comparison for State Features

Up to now, we have tried to present the transient behavior of our method throughout the learning process. Now, we will give the comparative results of our method and the previous work on controlling GRNs proposed by [51]. Remember that [51] models gene regulation as a PBN and then tries to obtain the infinite horizon optimal control policy of the constructed PBN. They formulate the control problem as an MDP and apply the Value Iteration algorithm. In our comparative experiment, we do the following. First, we have constructed a PBN from the seven-gene melanoma dataset. It is done based on the PBN construction algorithm presented in [60]. Then, we obtained a control policy from the method presented in [51] and from our proposed method separately. Lastly, we applied both of the policies to the same constructed PBN separately and checked the steady-state probability distributions of the controlled PBNs. Note that the method proposed by [51] used the constructed PBN to obtain a control policy. Whereas, our method did not use the constructed PBN, but obtained a control policy directly from the gene expression data as explained in Section 3.2. In fact, our aim is not to control the constructed PBN, but to control directly the gene regulation system that produced the real gene expression data. However, to be able to compare the quality of the two produced policies, we applied the two policies separately to the same constructed PBN and checked the steady-state probability distributions of the controlled PBNs, which is also the way the other studies, e.g., [51, 22, 13] follow.

Figure 3.6 shows the results of the probability distributions. In the left-hand side of Figure 3.6, we see that the steady-state probability distributions are shifted from undesirable states to the desirable states in the controlled PBNs with respect to the uncontrolled PBN. Note that uncontrolled PBN means to run the PBN without any

intervention, i.e., the action value is always 0. If we compare the two probability shifts provided by the policy of our Batch RL based method and provided by the policy of the existing Value Iteration based method, we see that our method is able to shift the probability mass better than the existing method. The probability values of being in the states 39, 53 and 55, which are among the desirable states, are significantly larger for the policy of our method while the other probability values are almost the same. In the right-hand side of Figure 3.6, we sum the probability values of the desirable states for both policies. As it is seen, we get 0.54 for the probability distribution of the uncontrolled PBN; 0.96 for the probability distribution of the PBN controlled by the existing method; and 0.98 for the probability distribution of the PBN controlled by our method.

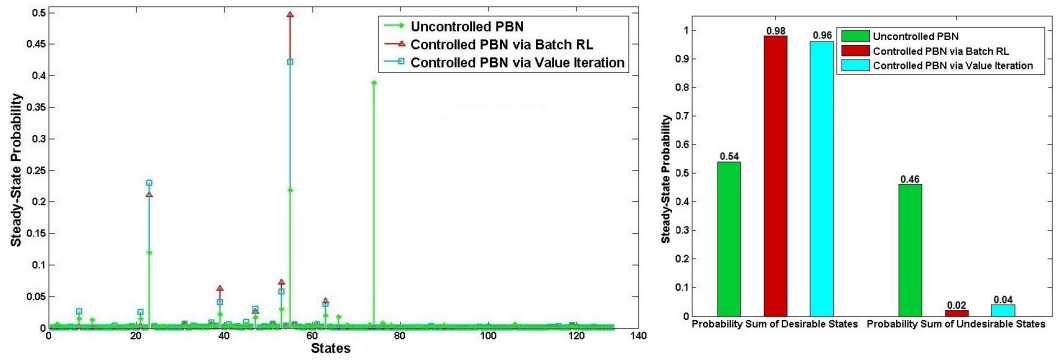

Figure 3.6: Steady-state probability distribution for State Features

If we compare the expected costs for the two control solutions, we see that our method is able to reduce the expected cost near to the optimal. The expected cost for a control solution is calculated as the dot product of the steady-state probability distribution of the controlled PBN and the cost function, which is given below (Equation 3.7).

$$E[C(\pi)] = \sum_{s=1}^{N} p(s) \cdot cost(s, \pi(s)) \tag{3.7}$$

where N is the number of states, $p(s)$ is the steady-state probability value of state s, π is the produced control policy, and *cost* is the cost function defined in Equation 3.6. Remember that uncontrolled PBN refers to the PBN without any intervention, i.e., $\pi(s) = 0$ for all the states of the uncontrolled PBN. Figure 3.7 shows the expected costs for uncontrolled and the two controlled PBNs. The expected cost for the uncontrolled PBN is 2.29, for the controlled PBN via Value Iteration is 0.21, and for the controlled PBN via Batch RL is 0.40. Since Value Iteration provably produces the optimal solution with the minimum possible expected cost of 0.21 [2], our method is able to produce a near-optimal solution producing expected cost of 0.40, by requiring much less time.

It is indeed a very interesting result. Because our method has nothing to do with

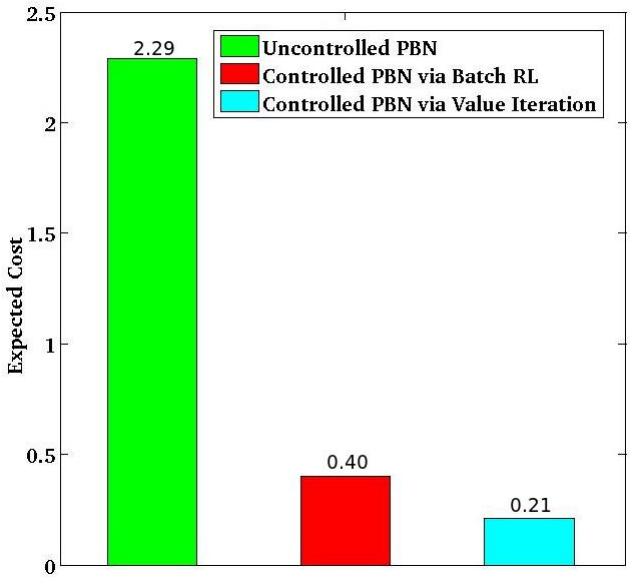

Figure 3.7: Expected costs for State Features

PBN, but its produced policy is almost the same as the optimal policy. Moreover it works almost as successful as the optimal policy obtained over the constructed PBN itself with the Value Iteration algorithm. Hence, it can be said that instead of building a gene regulation model such as PBN and dealing with complex details of the constructed computational model, a control problem can be solved by using directly the state transitions available in the gene expression data with almost the same solution quality. Moreover, since PBN is the most time consuming part of the available control solutions, it reduces the time requirements of the problem greatly as presented in detail in Section 3.3.4.

3.3.1.2 Gaussian Feature Results

This section presents the results for our proposed method adapting LSFQI to control gene regulation based on the second feature set we defined, namely Gaussian Features. We have done the same experiments with Section 3.3.1.1. Figure 3.8 shows the average costs over the iterations of LSFQI when we run LSFQI with the control policy it produced over the iterations and without any control, i.e. choosing the action value as always 0. We see that average cost is always less with control policy than without control. We also see that average cost converges to its ultimate value at the 7^{th} iteration.

Figure 3.9 shows the ultimate cost values for each state after the convergence again with and without control policy. We see that cost values are less with control policy than without control. We also see that costs of the desirable states, the states between $0 - 63$ are generally less than the costs of the undesirable states, the states between $64 - 127$, which is also consistent with the definition of the problem and our solution.

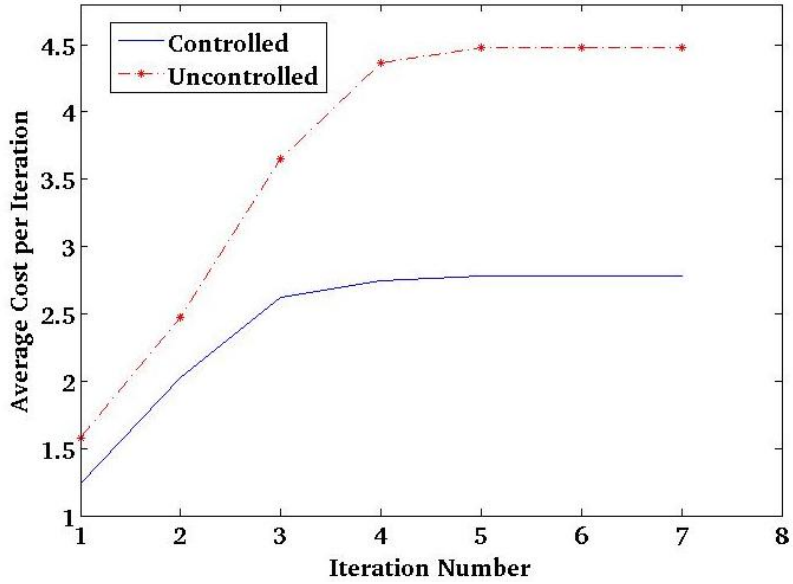

Figure 3.8: Average cost over the iterations of LSFQI for Gaussian Features

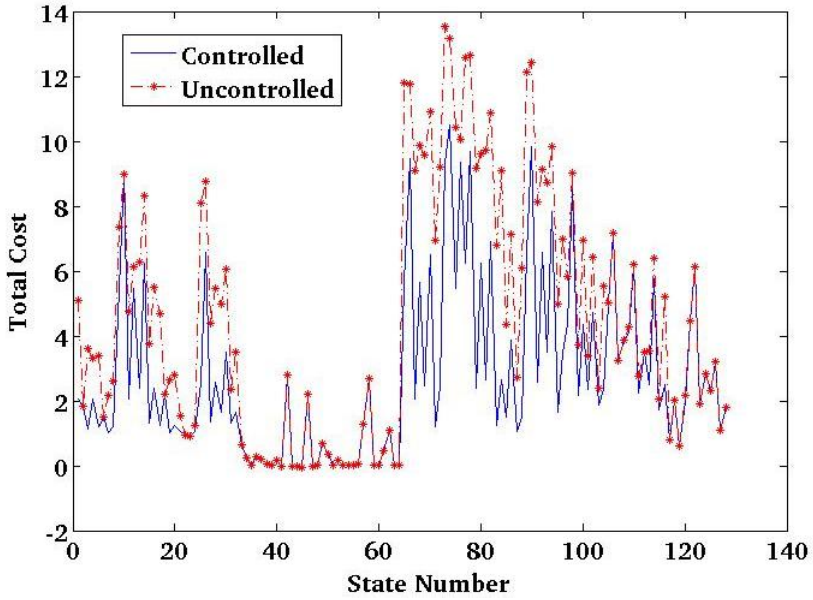

Figure 3.9: Value function after convergence for Gaussian Features

Figure 3.10, on the other hand, compares the two policies obtained from our LSFQI based method and the previous work of [51] on the same plot. We observe that the two policies are almost the same, which again proves the well-behavior of our LSFQI based method.

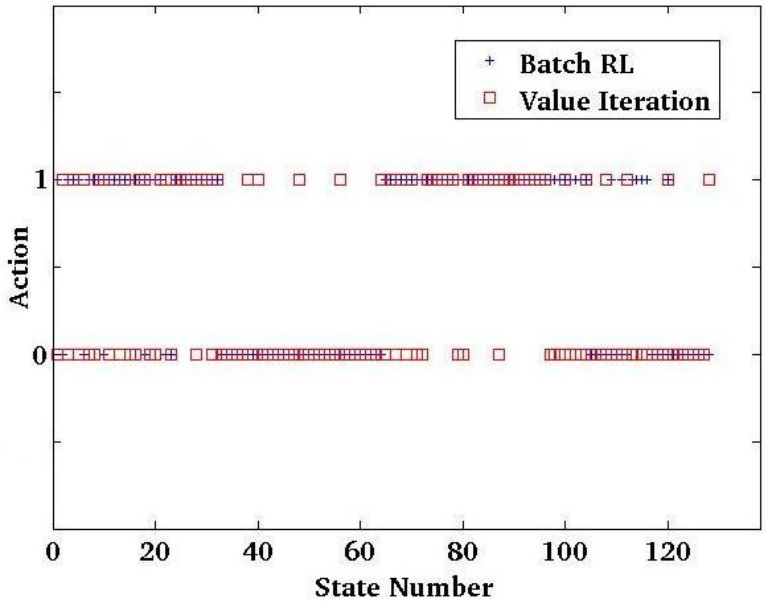

Figure 3.10: Policy comparison for Gaussian Features

As comparative results of our proposed LSFQI based method and previous work of [51], we have again checked the steady-state probability distributions of the controlled PBNs with the policy obtained by our LSFQI based method and with the policy obtained by the method of [51]. Remember that the method proposed by [51] used the constructed PBN to obtain a control policy. Whereas, our method did not use the constructed PBN, but obtained a control policy directly from the gene expression data as explained in Section 3.2. We applied the two policies separately to the same constructed PBN and checked the steady-state probability distributions of the controlled PBNs, as the other studies, e.g., [51, 22, 13] follow.

Figure 3.11 shows the results of the probability distributions. In the left-hand side of Figure 3.11, we see that the steady-state probability distributions are shifted from undesirable states to the desirable states in the controlled PBNs with respect to the uncontrolled PBN. If we compare the two probability shifts provided by the policy of our Batch RL based method and provided by the policy of the existing Value Iteration based method, we see that our method and the existing method of [51] shift the probability mass equally. The right-hand side of Figure 3.11 shows the sum of the probability values of the desirable states for both policies. As it is seen, we get 0.54 for the probability distribution of the uncontrolled PBN; 0.96 for the probability distribution of the PBN controlled by the existing method; and 0.96 for the probability distribution of the PBN controlled by our method.

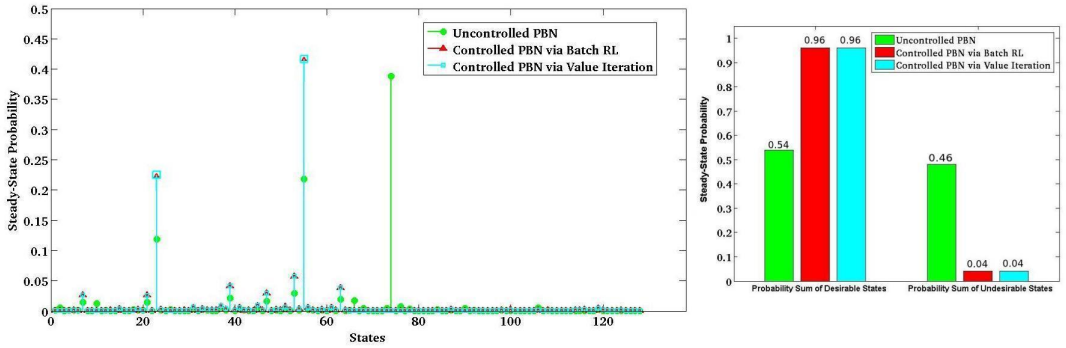

Figure 3.11: Steady-state probability distribution for Gaussian Features

If we compare the expected costs for the two control solutions, we see that our method is able to reduce the expected cost almost optimally. We have again used the expected cost function in Equation 3.7. Figure 3.12 shows the expected costs for uncontrolled and the two controlled PBNs. Remember that uncontrolled PBN refers to the PBN without any intervention, i.e., $\pi(s) = 0$ for all the states of the uncontrolled PBN. The expected cost for the uncontrolled PBN is 2.29, for the controlled PBN via Value Iteration is 0.21 and for the controlled PBN via Batch RL is 0.26. Since Value Iteration provably produces the optimal solution with the minimum possible expected cost of 0.21 [2], our method is able to produce almost optimal solution producing expected cost of 0.26, by requiring much less time.

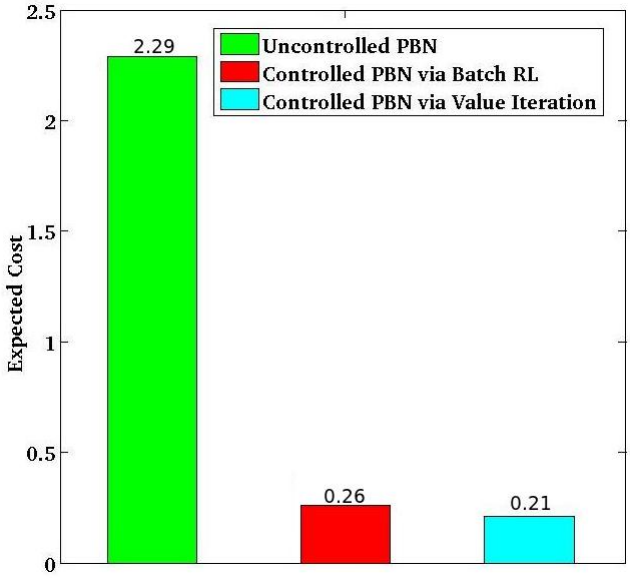

Figure 3.12: Expected costs for Gaussian Features

3.3.1.3 Distance Feature Results

This section presents the results for our proposed method adapting LSFQI to control gene regulation based on the third feature set we defined, namely Distance Features.

We have done the same experiments with Section 3.3.1.1 and 3.3.1.2. Figure 3.13 shows the average costs over the iterations of LSFQI when we run LSFQI with the control policy it produced over the iterations and without any control, i.e. choosing the action value as always 0. We see that average cost is always less with control policy than without control. We also see that average cost converges to its ultimate value at the 119^{th} iteration.

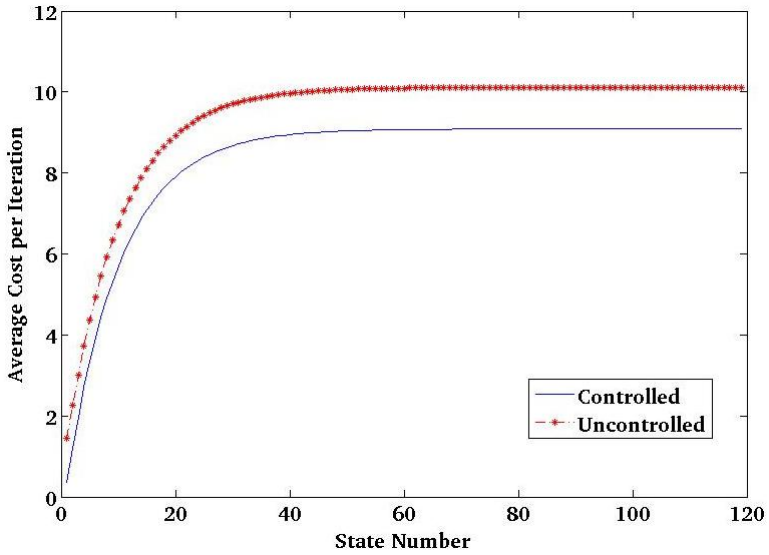

Figure 3.13: Average cost over the iterations of LSFQI for Distance Features

Figure 3.14 shows the ultimate cost values for each state after the convergence again with and without control policy. We see that cost values are less with control policy than without control. This time costs of the desirable states and undesirable states are similar, which we guess due to the approximation we had. Indeed, this shows our defined third feature set, Distance Features, are not successful enough to capture difference between experience tuples, which will be clearer in the experiments later in this section.

Figure 3.15, on the other hand, compares the two policies obtained from our LSFQI based method and the previous work of [51] on the same plot. We observe that the two policies are almost the same, which again proves the well-behavior of our LSFQI based method. However, we can also observe that there are more difference between the two policies in Figure 3.15 than the differences shown in Figure 3.5 and 3.10.

As comparative results of our proposed LSFQI based method and previous work of [51], we have again checked the steady-state probability distributions of the controlled PBNs with the policy obtained by our LSFQI based method and with the policy obtained by the method of [51]. Remember that the method proposed by [51] used the constructed PBN to obtain a control policy. Whereas, our method did not use the constructed PBN, but obtained a control policy directly from the gene expression data as explained in Section 3.2. We applied the two policies separately to the same constructed PBN and

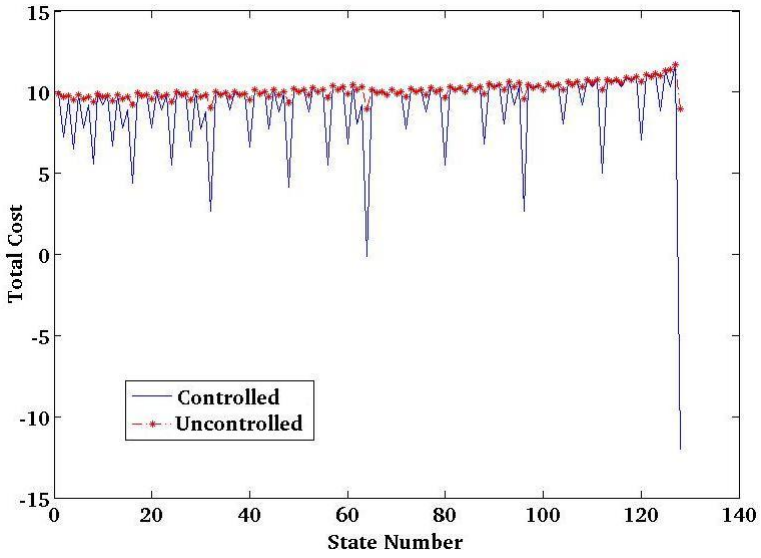

Figure 3.14: Value function after convergence for Distance Features

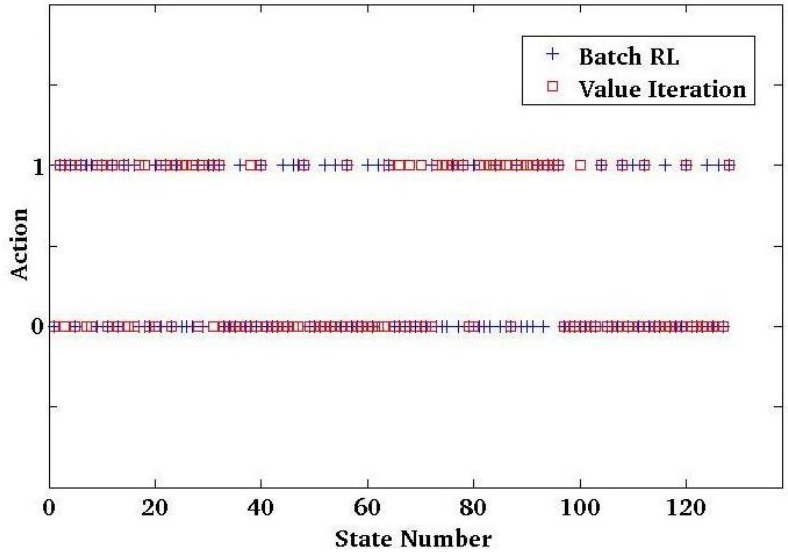

Figure 3.15: Policy comparison for Distance Features

checked the steady-state probability distributions of the controlled PBNs, as the other studies, e.g., [51, 22, 13] follow.

Figure 3.16 shows the results of the probability distributions. In the left-hand side of Figure 3.16, we see that the steady-state probability distributions are shifted from undesirable states to the desirable states in the controlled PBNs with respect to the uncontrolled PBN. If we compare the two probability shifts provided by the policy of our Batch RL based method and provided by the policy of the existing Value Iteration based method, we see that our method shift the probability mass worse than that of Value Iteration, but still performing reasonably well. The right-hand side of Figure 3.16 shows the sum of the probability values of the desirable states for both policies. As it is seen, we get 0.54 for the probability distribution of the uncontrolled PBN; 0.96 for the probability distribution of the PBN controlled by the existing method; and 0.90 for the probability distribution of the PBN controlled by our method. Hence, although our defined third feature set, Distance Features, performs reasonably well for shifting the probability mass, it is worse than our defined first and second features sets, State Features and Gaussian Features.

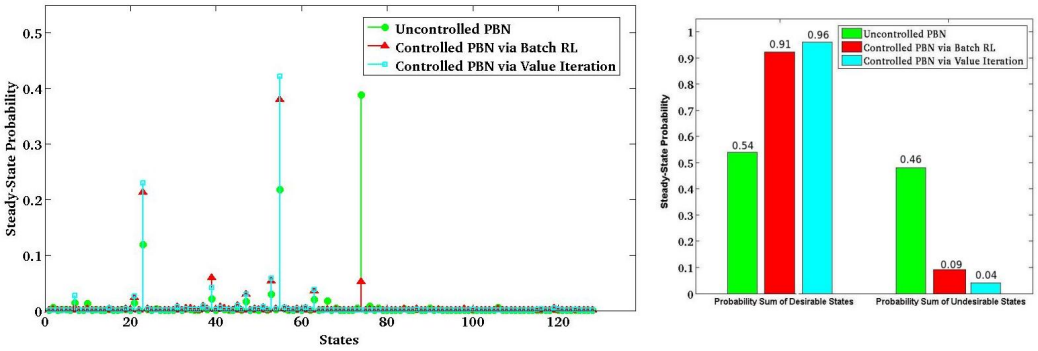

Figure 3.16: Steady-state probability distribution for Distance Features

If we compare the expected costs for the two control solutions, we see that our method is able to reduce the expected cost close to the optimal. We have again used the expected cost function in Equation 3.7. Figure 3.17 shows the expected costs for uncontrolled and the two controlled PBNs. Remember that uncontrolled PBN refers to the PBN without any intervention, i.e., $\pi(s) = 0$ for all the states of the uncontrolled PBN. The expected cost for the uncontrolled PBN is 2.29, for the controlled PBN via Value Iteration is 0.21 and for the controlled PBN via Batch RL is 0.51. Since Value Iteration provably produces the optimal solution with the minimum possible expected cost of 0.21 [2], our method is able to produce a solution that is close to the optimal solution having expected cost of 0.51, by requiring much less time. Note that this results also verifies our conclusion that our defined third feature set performs worse than the first and second feature sets we defined. When we compare with the existing work of [51], however, we can still say that our third feature set performs considerably well since it requires much less time than that of [51] and shifts the probability mass significantly.

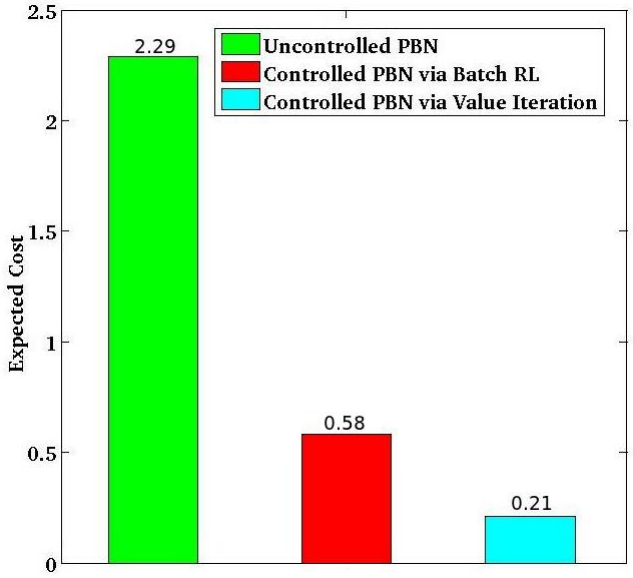

Figure 3.17: Expected costs for Distance Features

3.3.2 Yeast Application

This section describes the application of our method to yeast cell cycle dataset presented in [63]. Yeast cell cycle dataset is composed of 6178 genes and 77 samples. We have again compared our method with the previous infinite horizon solution for the GRN control problem presented in [51] in terms of their solution qualities.

We considered an ten-gene subset of the 25-gene yeast cell cycle dataset used by [3] and [67]. The 25-gene dataset is a subset of the complete yeast cell cycle dataset. The genes are MCM1, CLB5, ACE2, SWI6, MBP1, CTS1, STB1, SWI4, HTA1, and NDD1, in order, i.e., MCM1 is the most significant bit and NDD1 is the least significant bit in the state values. We have selected the 5^{th} gene, MBP1, as the input gene. We set the cost of applying an action as 1. The goal objective is to have MCM1 deactivated, its expression value as 0. We set penalty of not satisfying the goal as 5. Hence, the cost formulation is same as the one shown in Equation 3.6 Note that for states $[0-511]$ MCM1 has the value of 0, and for the remaining states $[512-1023]$ MCM1 is 1 since MCM1 is the most significant bit in the state values. Hence, the states $[0-511]$ are desirable while states $[512-1023]$ are undesirable. We also set our discount factor in the Algorithm 3 as 0.9.

Based on these experimental settings, we obtained an approximate control policy from our proposed method and compared it with the optimal policy obtained by the method proposed in [51]. The following subsections show the results for the three features sets we defined in Section 3.2.2, State Features, Gaussian Features, and Distance Features, respectively.

3.3.2.1 State Feature Results

This section presents the results for our proposed method adapting LSFQI to control gene regulation based on the first feature set we defined, namely State Features. Figure 3.18 shows the average costs over the iterations of LSFQI when we run LSFQI with the control policy it produced over the iterations and without any control, i.e. choosing the action value as always 0. We see that average cost is always less with control policy than without control. We also see that average cost converges to its ultimate value at the 34^{th} iteration.

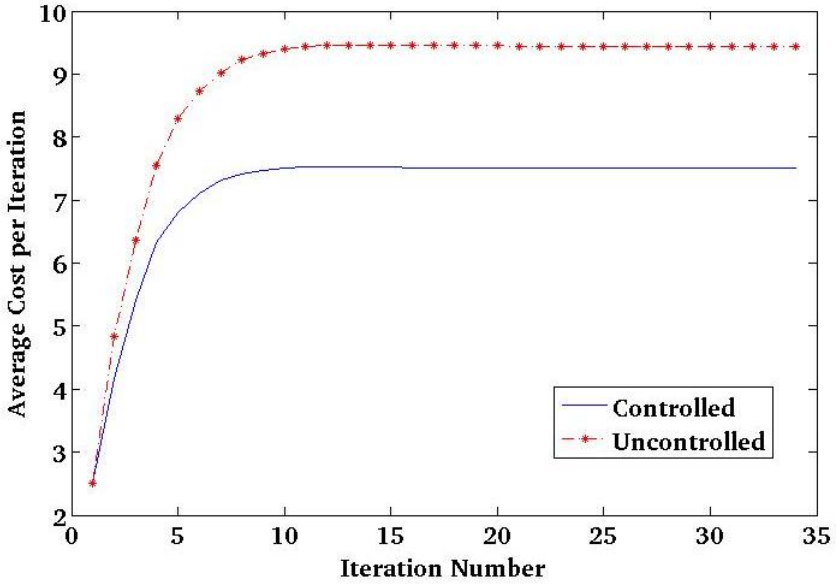

Figure 3.18: Average cost over the iterations of LSFQI for State Features

Figure 3.19 shows the ultimate cost values for each state after the convergence again with and without control policy. We see that cost values are less with control policy than without control. We also see that costs of the desirable states, the states between $0-511$ are generally less than the costs of the undesirable states, the states between $512-1024$, which is also consistent with the definition of the problem and our solution. Note that the negative values of the costs are again due to the approximation.

Figure 3.20, on the other hand, compares the two policies obtained from our LSFQI based method and the previous work of [51] on the same plot. We observe that, although the two policies have common results, they have also significant differences with respect to the results we obtained in Section 3.3.1. We see that for the 453 states of the all 1024 states, the policies have the same values, whereas for the rest 571 states, they have different values. Hence, we can say that for yeast cell cycle dataset our method is not as successful as it is for the melanoma dataset. The comparison between the qualities of the policies will be analyzed in detail later in this section.

As comparative results of our proposed LSFQI based method and previous work of [51],

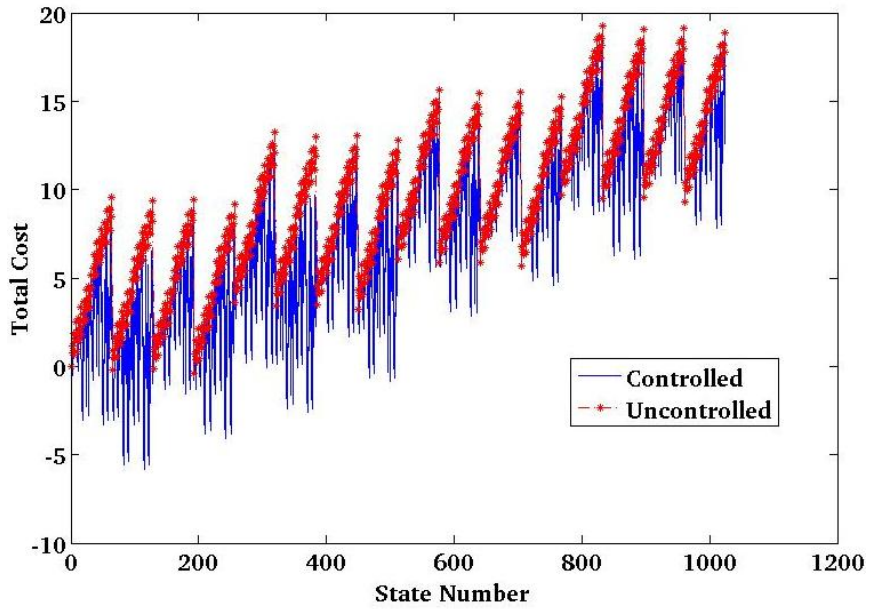

Figure 3.19: Value function after convergence for State Features

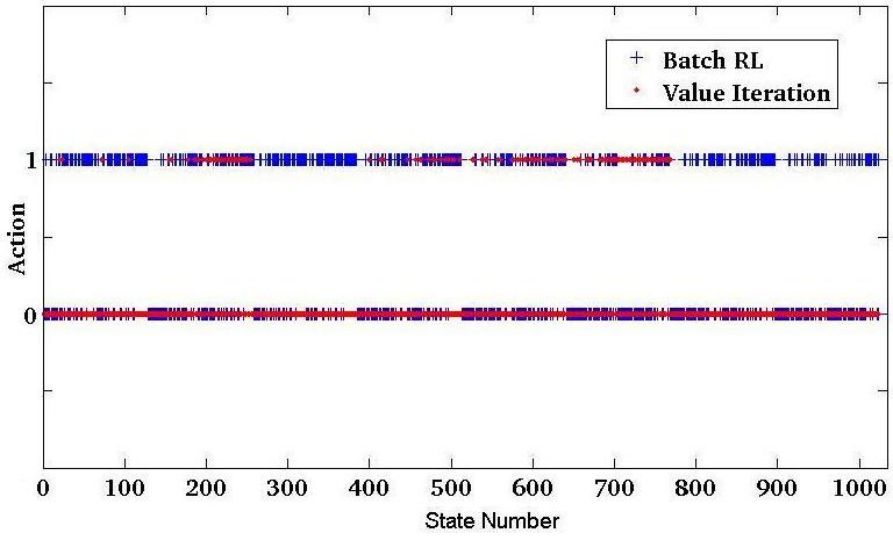

Figure 3.20: Policy comparison for State Features

we have again checked the steady-state probability distributions of the controlled PBNs with the policy obtained by our LSFQI based method and with the policy obtained by the method of [51]. Remember that the method proposed by [51] used the constructed PBN to obtain a control policy. Whereas, our method did not use the constructed PBN, but obtained a control policy directly from the gene expression data as explained in Section 3.2. We applied the two policies separately to the same constructed PBN and checked the steady-state probability distributions of the controlled PBNs, as the other studies, e.g., [51, 22, 13] follow.

Figure 3.21 shows the results of the probability distributions. In the left-hand side of Figure 3.21, we see that the steady-state probability distributions are shifted from undesirable states to the desirable states in the controlled PBNs with respect to the uncontrolled PBN. That shows both of the control policies are working correctly. If we compare the two probability shifts provided by the policy of our Batch RL based method and provided by the policy of the existing Value Iteration based method, we see that our method shifts considerable amount of probability mass to desirable states, but less than the probability shift of the optimal policy. The right-hand side of Figure 3.26 shows the sum of the probability values of the desirable states for both policies. As it is seen, we get 0.28 for the probability distribution of the uncontrolled PBN; 0.56 for the probability distribution of the PBN controlled by the existing method; and 0.41 for the probability distribution of the PBN controlled by our method. Since LSFQI basically applies an approximation over the gene expression data, our method highly depends on the gene expression data we are using. Because all of the features are derived from the data, which indeed limits our method in a sense. Therefore, it is reasonable for our method to have worse performance than the optimal policy. Indeed, if we count the great reduction on time requirements by our method, the performance of 0.41 probability shift is significantly good, which will be clearer by the experiments in Section 3.3.3 and 3.3.4.

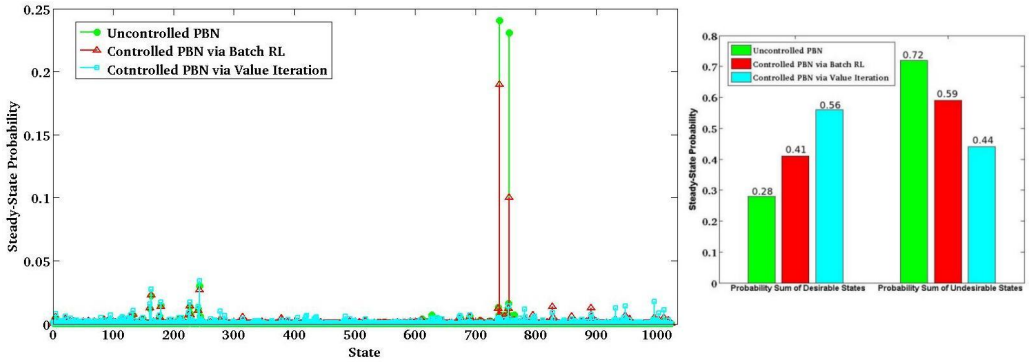

Figure 3.21: Steady-state probability distribution for State Features

Figure 3.22 shows the expected costs for uncontrolled and the two controlled PBNs. We have again used the expected cost function in Equation 3.7. Remember that uncontrolled PBN refers to the PBN without any intervention, i.e., $\pi(s) = 0$ for all the states of the uncontrolled PBN. The expected cost for the uncontrolled PBN is 3.58,

for the controlled PBN via Value Iteration is 2.28 and for the controlled PBN via Batch RL is 3.57. The cost of 2.28 is the minimum possible cost since it has produced by the Value Iteration, which is provably optimal. Our method, however, can reduce the cost value only 0.01 with respect to the uncontrolled PBN. The main reason for this is that our method produces policies applying more actions than the optimal policy. If we check Figure 3.20, it can be seen that there are much more 1 action's in the policy of our method than the optimal policy. Hence, our method cannot capture the fact that letting the PBN evolve itself both provides better probability shift and costs less. Yet, since the approximate policy produced by our LSFQI based method shifts probability mass reasonably well, it can still be said that our method is successful.

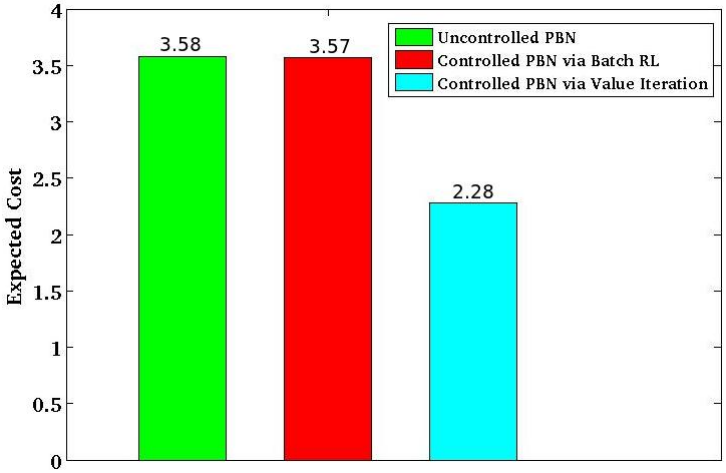

Figure 3.22: Expected costs for Distance Features

3.3.2.2 Gaussian Feature Results

This section presents the results for our proposed method adapting LSFQI to control gene regulation based on the second feature set we defined, namely Gaussian Features. We have done the same experiments with Section 3.3.2.1. Figure 3.23 shows the average costs over the iterations of LSFQI when we run LSFQI with the control policy it produced over the iterations and without any control, i.e. choosing the action value as always 0. We see that average cost is always less with control policy than without control. We also see that average cost converges to its ultimate value at the 114^{th} iteration.

Figure 3.24 shows the ultimate cost values for each state after the convergence again with and without control policy. We see that cost values are less with control policy than without control. We also see that, for the controlled case, costs of the desirable states, the states between $0 - 511$ are generally less than the costs of the undesirable states, the states between $512 - 1024$, which is also consistent with the definition of the problem and our solution.

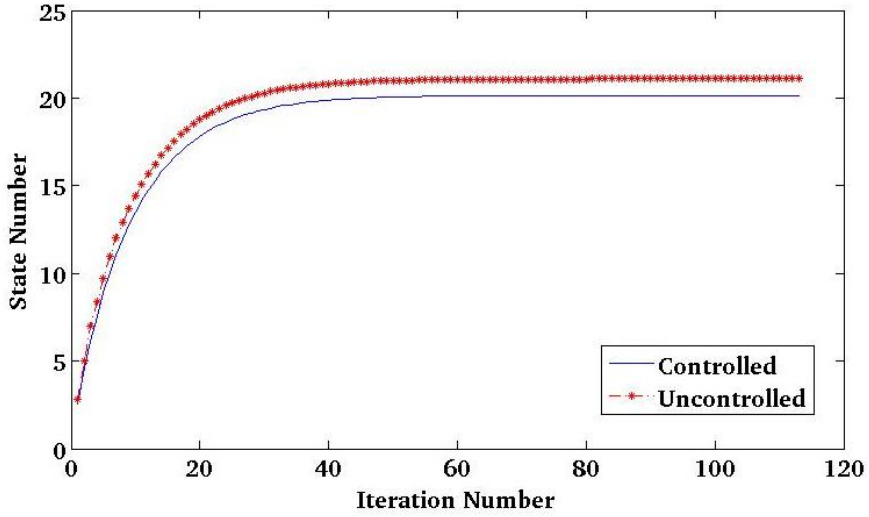

Figure 3.23: Average cost over the iterations of LSFQI for Gaussian Features

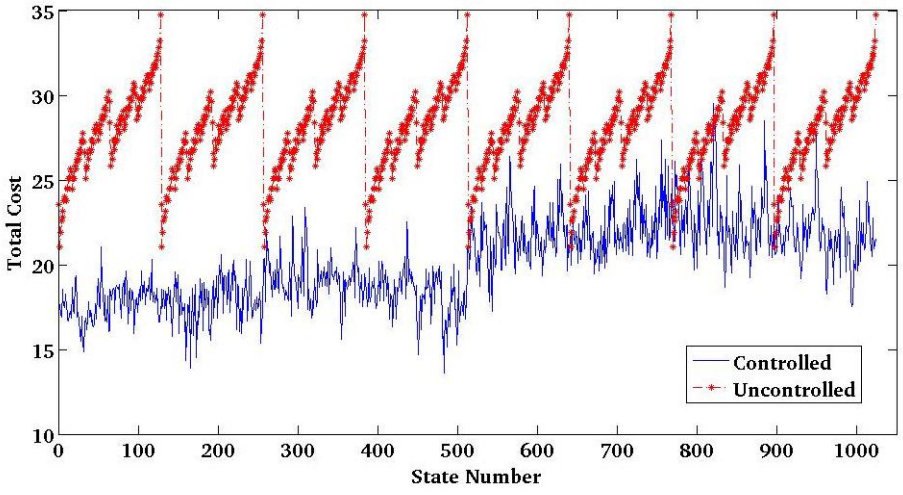

Figure 3.24: Value function after convergence for Gaussian Features

Figure 3.25, on the other hand, compares the two policies obtained from our LSFQI based method and the previous work of [51] on the same plot. Once again observe that there is significant difference between the obtained policies. While for 483 of the all 1024 states, the policy values are same, the rest of the 541 states policy values are different. Hence, we can again say that, for yeast cell cycle dataset, our method is not as successful as it is for melanoma dataset with Gaussian Features as well.

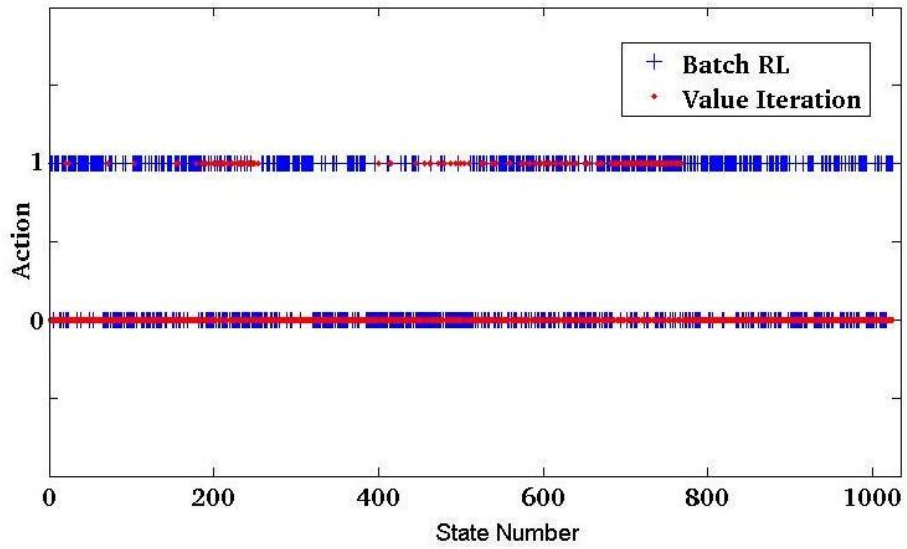

Figure 3.25: Policy comparison for Gaussian Features

As comparative results of our proposed LSFQI based method and previous work of [51], we have again checked the steady-state probability distributions of the controlled PBNs with the policy obtained by our LSFQI based method and with the policy obtained by the method of [51]. We have applied the two policies separately to the same constructed PBN and checked the steady-state probability distributions of the controlled PBNs, as the other studies, e.g., [51, 22, 13] follow.

Figure 3.26 shows the results of the probability distributions. In the left-hand side of Figure 3.26, we see that the steady-state probability distributions are shifted from undesirable states to the desirable states in the controlled PBNs with respect to the uncontrolled PBN. That shows both of the control policies are working correctly. If we compare the two probability shifts provided by the policy of our Batch RL based method and provided by the policy of the existing Value Iteration based method, we again see that our method performs worse than the Value Iteration. The right-hand side of Figure 3.26 shows the sum of the probability values of the desirable states for both policies. As it is seen, we get 0.28 for the probability distribution of the uncontrolled PBN; 0.56 for the probability distribution of the PBN controlled by the existing method; and 0.43 for the probability distribution of the PBN controlled by our method. Since State Features provide to shift the probability mass up to 0.41, we can say that Gaussian Features are more successful for yeast cell cycle dataset. Since our method is approximation of the optimal control policy, we can again say that the

performance of 0.43 probability shift is significantly good in consideration with the great reduction on time requirement of our method.

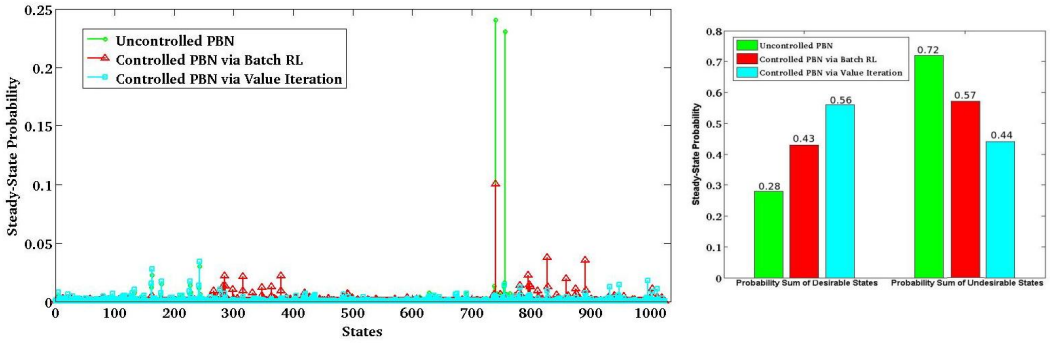

Figure 3.26: Steady-state probability distribution for Gaussian Features

Figure 3.27 shows the expected costs for uncontrolled and the two controlled PBNs. We have again used the expected cost function in Equation 3.7. The expected cost for the uncontrolled PBN is 3.58, for the controlled PBN via Value Iteration is 2.28 and for the controlled PBN via Batch RL is 3.44. This time, we can reduce the expected cost comparatively better than we have done with State Features. But still, we cannot get close to the optimal expected cost of 2.28. Again, we can say that our method applies unnecessary actions making the expected cost greater than the optimal expected cost. Yet, thanks to the considerable amount of probability shift shown in Figure 3.26, we can say that the approximate policy generated by our method is successful with Gaussian Features, and more successful than it is with State Features.

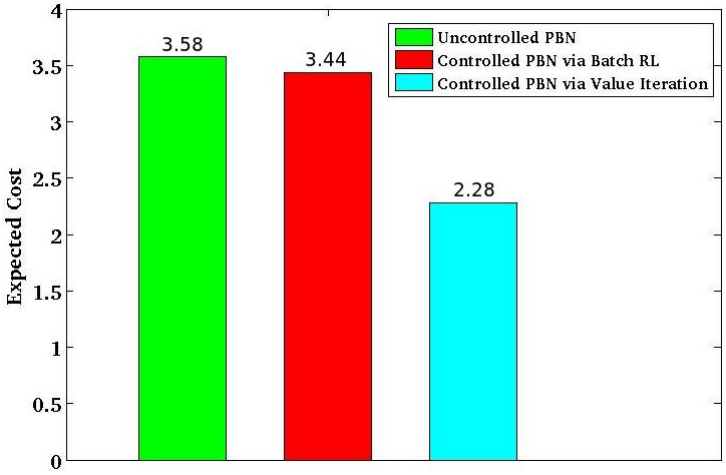

Figure 3.27: Expected costs for Gaussian Features

3.3.2.3 Distance Feature Results

This section presents the results for our proposed method adapting LSFQI to control gene regulation based on the third feature set we defined, namely Distance Features.

We have done the same experiments with Section 3.3.2.1 and 3.3.2.2. Figure 3.28 shows the average costs over the iterations of LSFQI when we run LSFQI with the control policy it produced over the iterations and without any control, i.e. choosing the action value as always 0. This time we see that there is not much difference between the controlled and uncontrolled iterations of LSFQI. That means, the policy obtained by our method with Distance Features includes many 0 actions, which has also been shown in Figure 3.30. Since the optimal policy of the value iteration shown in Figure 3.20, 3.25 and 3.30, has many 0 actions as well, it is reasonable to have many 0 actions in the produced policy, and to have close average cost values in the controlled and uncontrolled iterations of the LSFQI. We also see that average cost converges to its ultimate value at the 130^{th} iteration.

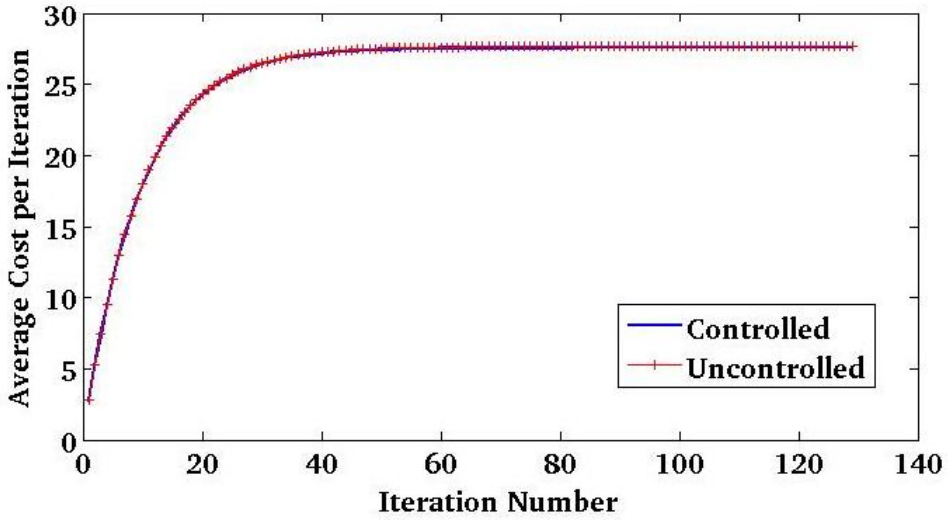

Figure 3.28: Average cost over the iterations of LSFQI for Distance Features

Figure 3.29 shows the ultimate cost values for each state after the convergence again with and without control policy. We observe that, although cost values of the control policy less than that of without control for some states, it is vice versa for some other states. For 589 states, the cost with the control policy is greater than the cost without control; and for the rest of the 435 states, the cost with the control policy is less than the cost without control. This is an undesired result. Although it is good to have a policy close to the optimal policy, our algorithm is expected to provide less cost values when its produced policy is used to calculate the costs. Hence, from the transient behavior of the learning process of our method with Distance Features, we can say that the approximation is not that much successful with Distance Features.

Figure 3.30, on the other hand, compares the two policies obtained from our LSFQI based method and the previous work of [51] on the same plot. We observe that the two policies has a lot in common with respect to the other features we have used in Section 3.3.2.1 and 3.3.2.2. For 877 states, the two policies has the same value, while for 147 states, they are different.

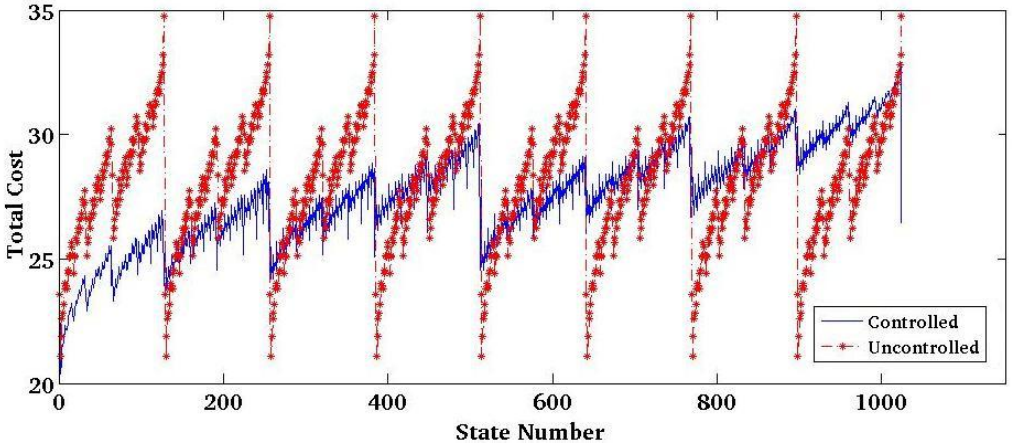

Figure 3.29: Value function after convergence for Distance Features

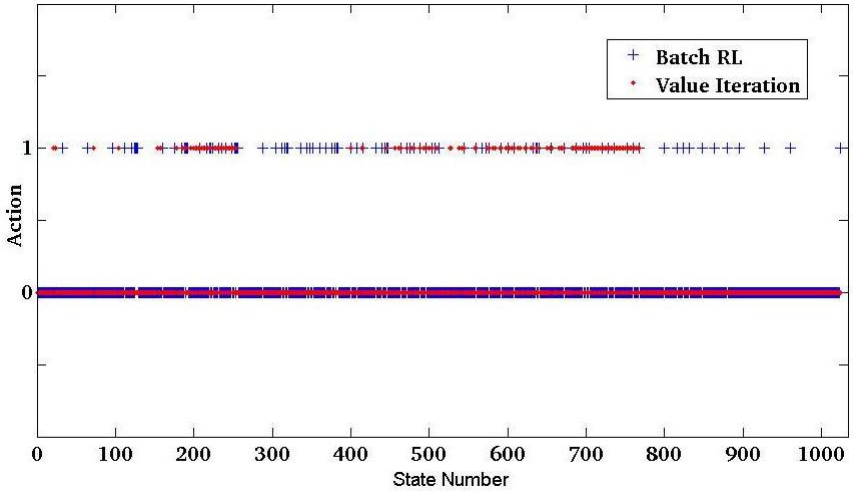

Figure 3.30: Policy comparison for Distance Features

As comparative results of our proposed LSFQI based method and previous work of [51], we have again checked the steady-state probability distributions of the controlled PBNs with the policy obtained by our LSFQI based method and with the policy obtained by the method of [51]. We have applied the two policies separately to the same constructed PBN and checked the steady-state probability distributions of the controlled PBNs, as the other studies, e.g., [51, 22, 13] follow.

Figure 3.31 shows the results of the probability distributions. In the left-hand side of Figure 3.31, we see that the steady-state probability distributions are shifted from undesirable states to the desirable states in the controlled PBNs with respect to the uncontrolled PBN, which shows both of the produces policies are working correctly. If we compare the two probability shifts provided by the policy of our Batch RL based method and provided by the policy of the existing Value Iteration based method, we observe that our method performs worse than the Value Iteration, although the produced policies shown in Figure 3.30 has a lot of common actions. The right-hand side of Figure 3.26 shows the sum of the probability values of the desirable states for both policies. As it is seen, we get 0.28 for the probability distribution of the uncontrolled PBN; 0.56 for the probability distribution of the PBN controlled by the existing method; and 0.38 for the probability distribution of the PBN controlled by our method. Since sum of the probability shifts were 0.41 and 0.43 for State and Gaussian Features, respectively, we can say that our method performs worse with Distance Features than State and Gaussian Features. This is an interesting result, indeed. Because the produced policy of Distance Features has the most common actions with the optimal policy with respect to the other defined features, while the shift of the probability mass is the least one. Therefore, we can conclude that, despite the commonality with the optimal policy, Distance Features cannot capture some critical actions making the PBN evolve to the optimal probability distribution having maximum probability mass in desirable states.

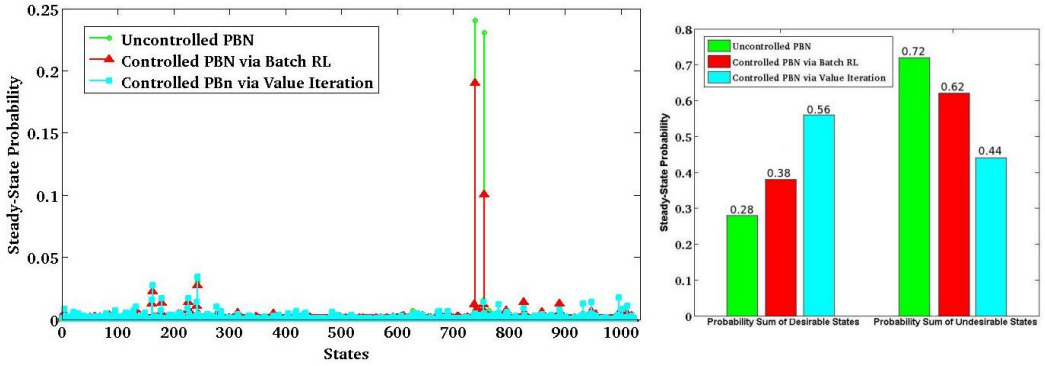

Figure 3.31: Steady-state probability distribution for Distance Features

Figure 3.32 shows the expected costs for uncontrolled and the two controlled PBNs. We have again used the expected cost function in Equation 3.7. The expected cost for the uncontrolled PBN is 3.58, for the controlled PBN via Value Iteration is 2.28

and for the controlled PBN via Batch RL is 3.17. Distance Features can reduce the expected cost comparatively better than the State and Gaussian Features. The main reason for this is to have less 1 actions in the produced policy of the Distance Features as Figure 3.30 shows. But still, we cannot get close to the optimal expected cost of 2.28. Since there is not much difference between the approximate policy produced by our method with Distance Features and the optimal policy, the main reason of high expected cost is to have high probability mass in the undesirable states of the converged PBN, which is shown by 3.31. Thanks to time requirement of our method, the result of our approximate policy produce by Distance Features is still acceptable. However, if it will be applied to yeast cell cycle regulation, utilization of State or Gaussian Features would be better.

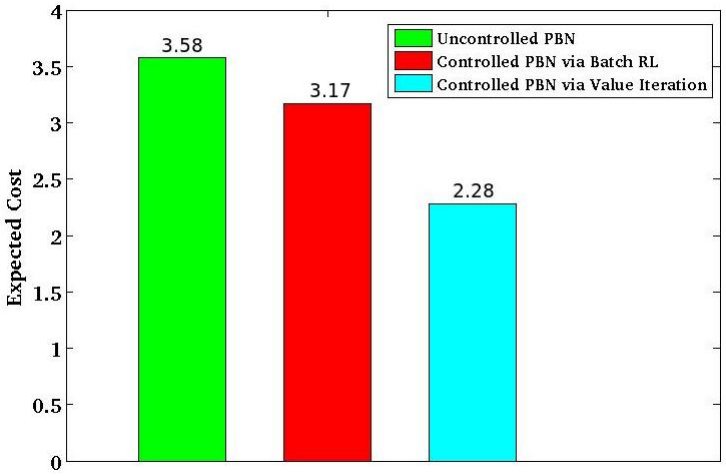

Figure 3.32: Expected costs for Distance Features

3.3.3 Large Scale Melanoma Application

This section describes the results of our method on a large scale gene regulation system. We have used a subset of the melanoma dataset presented in [5]. We combined the 10 interacting genes presented in Table 3 of [37] and the 22 highly weighted genes that form the major melanoma cluster presented in Figure 2b of [5]. The 10-gene set includes the genes pirin, WNT5A, S100, RET-1, MMP-3, PHO-C, MART-1, HADHB, synuclein and STC2. The 22-gene set includes the genes WNT5A, MART-1, pirin, HADHB, CD63, EDNRB, PGAM1, HXB, RXRA, ESTs, integrin b1, ESTs, syndecan4, tropomyosin1, AXL, EphA2, GAP43, PFKL, synuclein a, annexin A2, CD20 and RAB2. Since the four genes WNT5A, pirin, MART1 and HADHB, are available in both sets, we have 28 genes in total and 31 samples that the melanoma dataset already provides. Table 3.1 presents the list of genes with their descriptions. Our objective is again to have WNT5A deactivated, its expression value as 0, and use the same cost formulation shown in Equation 3.6. We have used State Features due to their best probability mass shift to desirable states in the experiments of melanoma dataset,

Section 3.3.1. We again set the WNT5A as the most significant bit and STC2 as the least significant bit. Hence, the desirable states are $[0 - 2^{27})$ since WNT5A has its expression value as 0, and undesirable states are $[2^{27} - 2^{28})$ since WNT5A has its expression value as 1. The input gene is also the 2^{nd} gene, pirin. The order of the genes in the state value representation is WNT5A, pirin, MART1, HADHB, CD63, EDNRB, PGAM1, HXB, RXRA, ESTs, integrin b1, ESTs, syndecan4, tropomyosin1, AXL, EphA2, GAP43, PFKL, synuclein a, annexin A2, CD20, RAB2, S100P, RET1, MMP3, PHOC, synuclein and STC2.

We have applied our proposed method to the extended 28-gene subset of the melanoma dataset. As in Section 3.3.1 and Section 3.3.2, we have again measured the quality of the policy produced by our method with respect to the shift of the probability mass from undesirable states to the desirable states in a controlled PBN. Hence, firstly we applied our proposed method to the 28-gene melanoma dataset and obtained an approximate control policy for the 28-gene regulation system. Then, we constructed the PBN of the 28-gene regulation system with the algorithm presented in [60]. We applied the policy produced by our method to the constructed PBN and checked the steady-state probability distribution of the controlled PBN with respect to the steady-state probability distribution of the uncontrolled PBN. Since it is almost impossible to plot the probability value of each possible 2^{28} states, here, we sum the probability values in the desirable states and in the undesirable states. Then, we compared the probability sums of the desirable states and the undesirable states in the controlled and uncontrolled PBNs. Figure 3.33 shows the results.

For the uncontrolled PBN, the probability sum of desirable states is 0.01 and the undesirable states is 0.99. For the controlled PBN, on the other hand, the probability sum of desirable states is 0.8 and the undesirable states is 0.2. This means, by applying the policy obtained by our method to the constructed PBN, we can significantly shift the probability mass from undesirable states to desirable states. It was 0.99 probability to be in one of the undesirable states in the uncontrolled PBN. However, this value reduces to 0.2 if we control the PBN with respect to the control policy produced by our method. Therefore, we can say that our method not only works for small regulatory systems, but also solves large scale control problems; this is a good justification for verifying its robustness and effectiveness.

Note that, 28-gene regulatory system may not seem as large enough since our method can easily produce control policies for regulation systems composed of several thousands of genes. However, here, we verify our method with respect to a constructed PBN as existing methods have done, and PBN construction algorithm limits our experiments due to its $O(d^k \times n^{k+1})$ time and space complexities, where n is the number of genes, k is the maximum number of predictor genes and d is the discretization level [60]. Actually, PBN construction algorithm does not work for regulation systems larger than 50 genes for $k = 3$ and $d = 2$ with our current hardware configuration, Intel i7 processor and 8-GB memory, especially due to its space requirement. Still, to the best

of our knowledge, it is the first study successfully producing a control solution for a gene regulation system with more than 15 genes.

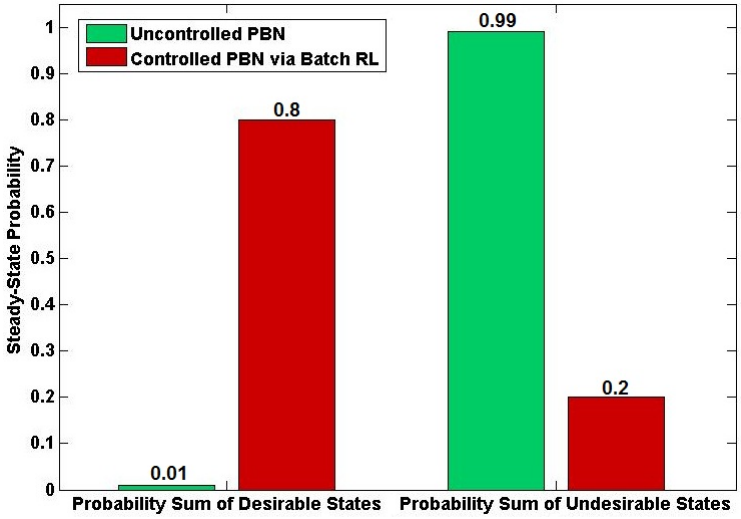

Figure 3.33: Large scale melanoma steady-state probability shift

3.3.4 Time Requirements

This section describes the time requirement of our proposed solution for controlling GRNs. We show how our method significantly reduces time requirements of the GRN control problem. We again used the gene expression data presented in [5], and again the same configurations explained in Section 3.3.3. This time, we gradually increased the number of genes in the dataset from 10 genes to 8067 genes, applied our method and checked the elapsed time to obtain a controlling policy. Figure 3.34 shows the results. As it is shown, time increases linearly with the number of genes in the dataset and the maximum required time to obtain a control policy for the complete gene regulation system of 8067 genes is just about 6 seconds [1]. It is a great improvement since existing PBN-based studies cannot solve control problems even for several tens of genes. Moreover, to our best knowledge, it is the first solution that can produce policies for regulation systems with several thousands of genes.

One important point here is that LSFQI solves a regression problem at each iteration, and regression problem scales at best quadratically in terms of the number of features. The reason for our method scales linearly with the number of genes, i.e., number of features, however, is that, the computation time for one iteration of the LSFQI algorithm is not large enough to capture the quadratic relationship. To illustrate, if the number of genes would increase to much larger numbers, such as 100.000, most probably, the quadratic behavior would show itself clearer. This issue will be clearer

[1] We have used MATLAB's pinv (Moore-Penrose pseudoinverse) function for solving the least-squares regression problem in the Least-Squares FQI algorithm

when we explained the time requirement of our first partial observability solution in Chapter 4, in its Section 4.6.3.

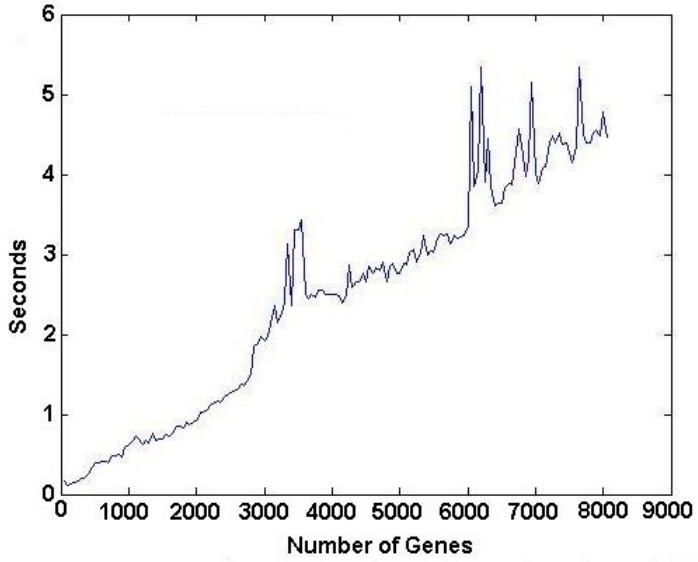

Figure 3.34: Execution time for our method

3.4 Discussion

In this chapter, we have proposed a novel method for controlling GRNs. Our algorithm makes use of Batch Mode Reinforcement Learning to produce a control policy directly from the gene expression data. The idea is to treat the time series gene expression data as a sequence of experience tuples and compute an approximate policy over those experience tuples without explicitly generating any computational model for gene regulation such as Probabilistic Boolean Network. The reported results show that our proposed method is successful in producing control policies with almost the same solution qualities compared to the previous control solutions. Moreover, it can solve control problems with several thousands of genes just in seconds, whereas existing methods cannot solve the control problem even for several tens of genes. To the best of our knowledge, it is the first study that can generate solutions for gene regulation systems with several thousands of genes.

Table3.1: List of genes for the extended 28-gene melanoma dataset

Clone ID	Title	Gene Description
324901	WNT5A	wingless-type MMTV integration site family, member 5A
234237	pirin	pirin
266361	MART1	Human melanoma antigen recognized by T-cells (MART-1) mRNA
108208	HADHB	hydroxyacyl-Coenzyme A dehydrogenase/ 3-ketoacyl-Coenzyme A thiolase/ enoyl-Coenzyme A hydratase (trifunctional protein), beta subunit
39781	CD63	CD63 antigen (melanoma 1 antigen)
49665	EDNRB	endothelin receptor type B
39159	PGAM1	Phosphoglycerate mutase 1 (brain)
23185	HXB	hexabrachion (tenascin C, cytotactin)
417218	RXRA	Retinoid X receptor, alpha
31251	ESTs	ESTs
343072	integrin, b1	integrin, beta 1 (fibronectin receptor, beta polypeptide, antigen CD29 includes MDF2, MSK12)
142076	ESTs	ESTs
128100	syndecan4	Syndecan 4 (amphiglycan, ryudocan)
376725	tropomyosin1	Tropomyosin alpha chain (skeletal muscle)
112500	AXL	AXL receptor tyrosine kinase
241530	EphA2	EphA2
44563	GAP43	growth associated protein 43
36950	PFKL	Phosphofructokinase (liver type)
812276	synuclein, a	synuclein, alpha (non A4 component of amyloid precursor)
51018	annexin A2	Annexin II (lipocortin II)
306013	CD20	CD20 antigen
418105	RAB2	RAB2, member RAS oncogene family
759948	S100P	S100 calcium-binding protein, beta (neural)
25485	RET1	reticulon 1
324700	MMP3	matrix metalloproteinase 3 (stromelysin 1, progelatinase)
43129	PHOC	phospholipase C, gamma 1 (formerly subtype 148)
40764	synuclein	synuclein, alpha (non A4 component of amyloid precursor)
130057	STC2	stanniocalcin 2

CHAPTER 4

CONTROLLING GENE REGULATORY NETWORKS: BATCH MODE TD(λ)

4.1 Introduction

There are two different ways of defining the GRN control task. The first one is to assume full observability in the environment implying that all of the genes in the gene expression data is observable [12, 51]. Our proposed LSFQI based method in Chapter 3 solves the control problem of fully observable GRNs, for example. The second one, on the other hand, is to assume partial observability implying that only a subset of the genes in the gene expression data is observable [13]. Whereas full observability provides complete information about the state of the regulation system, partial observability provides only observation signals that are incomplete but probabilistically related with the states of the regulation system [13]. The aim is to find an intervention strategy solely based on those incomplete signals to avoid undesirable states of the system. In this chapter, we are concentrated on controlling partially observable GRNs.

There are several studies for controlling partially observable GRNs. The study in [13] models gene regulation as a Probabilistic Boolean Network (PBN), then tries to find a finite horizon optimal intervention strategy over the constructed PBN. Based on the initial belief state, it branches over all possible paths in the belief space for a finite horizon and finds the optimal action sequence for each time step based on the Dynamic Programming algorithm of [4]. The study in [7] also models gene regulation as a PBN and searches over the belief state space, but unlike the study of [13] searching all belief space blindly, it applies AO* algorithm searching only for the beliefs that are on the path of the optimal policy. These two studies formulates the control problem as a finite horizon Partially Observable Markov Decision Process (POMDP), indeed. The study in [18], on the other hand, formulates the GRN control problem as an infinite horizon POMDP. Their basic difference from the two other previous works is that, instead of making use of PBN for defining system dynamics of gene regulation, they check the similarities between the expression patterns of genes. By solving the constructed POMDP with the fast point-based POMDP solver symbolic Perseus [56], they produce an infinite horizon control policy for partially observable GRNs.

The basic and most important problem with the existing works for controlling partially observable GRNs is again that none of them can solve control problems with more than several tens of genes. The finite horizon work of [13], scales exponentially with respect to the horizon since it branches for all possible action & observation tuples. Their time and space complexity is $O(d^{(h\times(o+a))})$, where d is the discretization level, h is the length of the horizon, o is the number of observation genes and a is the number of action genes. For a regulation systems of 50 genes in which 30 genes are observable and binary discretization is used, it requires to deal with more than 1-billion belief nodes, which is highly infeasible even to keep in the memory. Although the other finite horizon work of [7] employs a heuristic to decrease the search time, still they could not scale for horizons larger than 8 even for small 7-gene regulation systems as Section 4.6.1.2 shows in detail. Moreover, both of the finite-horizon studies requires to build a PBN, which already takes $O(d^k \times n^{k+1})$ time and space complexity, where n is the number of genes, k, is the maximum number of predictor genes and d is the discretization level [60]. The infinite horizon work of [18], on the other hand, requires to solve a POMDP, which still cannot scale for large number of genes.

In this chapter, we propose a novel framework, Batch Mode TD(λ), combining Batch Mode Reinforcement Learning (Batch RL) methods and TD(λ) algorithm for controlling partially observable GRNs [21, 64]. Our idea is to interpret time series gene expression data as a sequence of observations that the gene regulation system produced, and to obtain an approximate stochastic policy directly from those observations without estimation of the internal states of the environment [61, 30]. In their study, Singh et al. (1994) and Jaakkola et al. (1994) have shown that stochastic policies can be arbitrarily better than deterministic policies in partially observable environments. They have also shown that, Temporal-Difference (TD) learning methods can produce arbitrarily worse stochastic policies than Monte-Carlo methods in non-Markovian environments since TD methods bootstrap, which is also pointed out by [65] in its Section 5.8 and 7.11. Hence, in our work, we have coupled Sutton's TD(λ) algorithm (1988) subsuming Monte-Carlo methods, and the Batch RL method Least-Squares Fitted Q Iteration (LSFQI) proposed by [21] to solve the control problem of partially observable GRNs. We convert the time series gene expression samples into a series of experience tuples and feed them into our novel Batch Mode TD(λ) framework producing an approximate stochastic policy mapping all possible observations to actions probabilistically. By doing so, we are able to reduce the time and space complexity greatly since we get rid of the two most time consuming processes of the previous works, inferring a PBN from the gene expression data and solving a finite/infinite horizon POMDP. Figure 4.1 shows the alternative solution methods including our proposal in the box. Results show that we can generate control policies for several thousands of genes just in seconds, while existing studies get stuck even for several tens of genes. Results also show that our approximate stochastic policies have almost the same solution qualities with the deterministic optimal policies produced by the previous works. Moreover, to our best knowledge, it is the first study proposing a Batch RL framework non-Markovian

decision problems with limited number of experience tuples. Since our method is independent of any computational model, it can easily be adapted for different problem domains and used for solving different non-Markovian decision tasks when there are limited number of experience tuples.

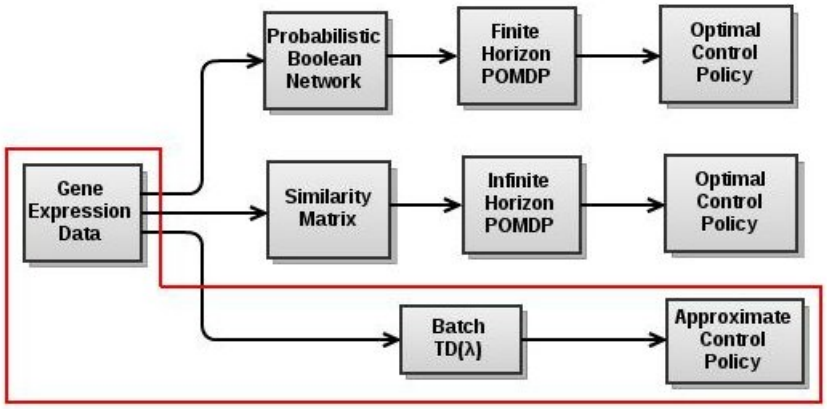

Figure 4.1: Flow of alternative control solutions

The rest of this chapter is organized as follows. Section 4.2 explains Monte-Carlo value estimation method. Section 4.3 explains TD(λ) algorithm. Section 4.4 describes our novel Batch Mode TD(λ) framework. Section 4.5 explains how we adapt our proposed Batch TD(λ) framework for solving the control problem of partially observable GRNs. Section 4.6 presents experimental evaluation of our method and Section 4.7 concludes with a discussion.

4.2 Monte-Carlo Value Estimation

This section describes Monte-Carlo Value estimation method. An RL task can be basically solved by two approaches, Monte-Carlo (MC) learning and Temporal-Difference (TD) learning. We have already explained one of the TD learning methods, Q-Learning in Section 2.2, Algorithm 2. The basic difference between MC and TD learning approaches is the way they update the state-action function. TD learning methods basically bootstrap. That is, they update value of an observed state based on the value of an observed successor state. Technically speaking, Q-learning makes use of the value of $\max_{a'} Q(s', a')$, which is the value of the successor state s', in order to update the value of current state-action pair, $Q(s, a)$, as the Equation in Line 14 of Algorithm 2 shows. MC learning methods, however, instead of updating state-action values by bootstrapping, take averages of the return values obtained after the experienced state-action pairs. Note that return value of a state-action pair is future discounted sum of rewards after visiting that state-action pair up to the end of the episode. Algorithm 4 shows the Monte-Carlo algorithm.

After generating an episode, meaning to collect a series of experience tuples from the

Algorithm 4: Monte-Carlo Algorithm

Input : discount factor γ, learning rate α, number of episodes E

1 **for** $episode = 1, ..., E$ **do**
2 $\quad [Rw, St, Ac] = generateEpisode()$
3 $\quad$ **for** $t = 1, 2..., T$ **do**
4 $\quad\quad R_t \leftarrow \sum_{l=t}^{T} \gamma^{l-t} Rw(St(l), Ac(l))$
5 $\quad\quad Q(St(t), Ac(t)) \leftarrow Q(St(t), Ac(t)) + \alpha(R_t - Q(St(t), Ac(t)))$
6 $\quad$ **end**
7 **end**

Output: Q

environment, MC calculates return values for each visited state-action pair, i.e., the R_t's shown in Line 4; and uses it to update the state-action function Q. Note that the way we have presented to implement MC is called as first-visit MC, since we use only the first visits to the state-action pairs at each episode to calculate return values R_t's. However, a state-action pair may be visited more than once in an episode. Hence, MC can also be implemented by taking the average of return values of every visits to each state-action pair at each episode, which is called as every-visit MC [65]. By the theory of large numbers the state-action function will converge to the optimal state-action function for sufficiently large number of episodes both for every-visit and first-visit MC methods [65]. As it is seen, MC method has never done bootstrapping as TD learning does.

4.3 TD(λ)

This section describes Sutton's TD(λ) algorithm (1988) [64] that we will use in our novel Batch RL framework, Batch Mode TD(λ). TD(λ) is generalization of TD and MC learning methods. TD(λ) provides difference value estimation schema for different values of λ. While TD(0) is same as TD learning algorithm, TD(1) is same as MC learning algorithm. The idea is to combine both TD and MC learning methods into one framework to benefit both of them together. TD learning methods use 1-step return to update the state-action function $Q(s, a)$. If we check Line 14 of Algorithm 2, it can be seen the basic update element is $R(s, a) + \gamma \max_{a'} Q(s', a')$, which is actually 1-step return. MC learning methods, on the other hand, use T-step return, where T is the length of the episode. T-step return is basically future discounted sum of rewards up to the end of the episode. If we check Line 4 of Algorithm 4, it can be seen that the basic update element is $\sum_{l=t}^{T} \gamma^{l-t} Rw(St(l), Ac(l))$, which is actually T-step return. Then the question of TD(λ) is whether we can use both 1-step return and T-step return together to update the state-action function Q. A very simple example would be just taking the average of 1-step return and T-step return to update the state-action function Q as Equations 4.1 to 4.4 show.

$$R_t^{(1)} = R(s_t, a_t) + \gamma \max_{a'} Q(s'_t, a') \tag{4.1}$$

$$R_t^{(T)} = \sum_{l=t}^{T} \gamma^{l-t} R(s_l, a_l) \tag{4.2}$$

$$R_t^{av} = \frac{1}{2} R_t^{(1)} + \frac{1}{2} R_t^{(T)} \tag{4.3}$$

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(R_t^{av} - Q(s_t, a_t)) \tag{4.4}$$

What TD(λ) does is indeed generalizing the idea of using both 1-step and T-step returns together to update the state-action function Q. In order to do this, TD(λ) defines n-step return which can be formulated as below.

$$R_t^{(n)} = R(s_t, a_t) + \gamma R(s_{t+1}, a_{t+1}) + \cdots + \gamma^{n-1} R(s_{t+n-1}, a_{t+n-1}) + \max_{a'} Q(s_{t+n}, a') \tag{4.5}$$

Instead of bootstrapping from next state value by using $\max_{a'} Q(s'_t, a')$ as 1-step return does, and never bootstrapping from any state as T-step return does; n-step return uses future discounted sum of rewards up to the $(t+n-1)^{th}$ time step, and bootstrap from the $(t+n)^{th}$ state value. What TD(λ) does is to update the value of the state-action pair at time t, $Q(s_t, a_t)$, by taking weighted averages of n-step returns, namely $R_t^{(n)}$, for $t \leq n \leq T$, with respect to the λ value, given a T-step episode. Algorithm 5 shows the TD(λ) algorithm. Line 2 of the Algorithm 5 generates an episode constituting a series of experience tuples. Between Lines $4-6$, TD(λ) calculates the n-step returns. Then, as shown by Line 7, it averages the n-step returns with respect to the λ value to find the R_t^{λ} value. Lastly, it updates the value of the state-action pair $St(t+1)$ and $Ac(t+1)$, $Q(St(t+1), Ac(t+1))$, based on the averaged return value R_t^{λ}. TD(λ) is the same as the MC algorithm for $\lambda = 1$ and same as the TD learning algorithm for $\lambda = 0$. For $0 < \lambda < 1$, it is somewhere between MC and TD learning. As λ gets close to 1 TD(λ) get closes to MC learning, and vice versa for TD learning.

4.4 Batch Mode TD(λ) for Partially Observable Environments

This section describes our proposed Batch Mode TD(λ) algorithm for partially observable environments, Least-Squares Fitted TD(λ) Iteration (LSFTDI). What we have done is to embed Sutton's TD(λ) algorithm (1988) into the LSFQI algorithm of Ernst et al. (2005), explained in Section 2.3.1. Our main motivation is that, in partially observable environments, where state information is incomplete, the bootstrapping effect

Algorithm 5: TD(λ) Algorithm

Input : discount factor γ, learning rate α, weight λ, number of episodes E

1 **for** $episode = 1, ..., E$ **do**
2 $[Rw, St, Ac] = generateEpisode()$
3 **for** $t = 0, 1..., (T-1)$ **do**
4 **for** $n = 1, 2, ..., (T-t)$ **do**
5 $R_t^{(n)} \leftarrow Rw(t+1) + \gamma Rw(t+2) + ... + \gamma^{n-1} Rw(t+n) + \gamma^n \max_{a'} Q(St(t+n), a'))$
6 **end**
7 $R_t^{\lambda} \leftarrow (1-\lambda)[R_t^{(1)} + \lambda R_t^{(2)} + ... + \lambda^{T-t-2} R_t^{(T-t-1)}] + \lambda^{T-t-1} R_t^{(T-t)}$
8 $Q(St(t+1), Ac(t+1)) \leftarrow Q(St(t+1), Ac(t+1)) + \alpha(R_t^{\lambda} - Q(St(t+1), Ac(t+1)))$
9 **end**
10 **end**

Output: Q

of TD learning methods, such as Watkin's Q-learning (1989), significantly reduces the quality of the produced policy [61]. The main reason for this is TD learning methods update the value of an observed state based on the value of an observed successor state. For example, Q-learning makes use of the value of the successor state s' in order to find the value of current state-action pair, $Q(s, a)$, as the Equation in Line 14 of the Algorithm 2 shows. However, since state information is incomplete, it results in accumulating mistakenly observed state values into mistakenly observed other state values [61, 65]. Therefore, Jaakkola et al. (1994) have proposed a Reinforcement Learning (RL) algorithm for partially observable environments, which basically calculates an action value for each observation based on the Monte-Carlo (MC) value estimation method, and assigns a probability value for each action of observations in proportion to the calculated observation-action values. By doing so, in their methods [30] and [61] avoid the negative effect of bootstrapping of TD learning, and produce a stochastic policy mapping observations directly to actions probabilistically without estimation of the internal states of the partially observable environment. It has also been shown that their MC-based stochastic policy estimation method is guaranteed to converge to local maximum.

Motivated by the ideas of [61] and [30], and suggestions of [65] in its Section 5.8 and 7.11, we have combined LSFQI and TD(λ) algorithms into a unified Batch RL algorithm, LSFTDI, to solve the problem of controlling partially observable GRNs. Note that, unlike the studies of Jaakkola et al. (1994) employing MC value estimation method, we have used TD(λ) for estimating values of the observations. This is because TD(λ) algorithm is generalization of TD and MC learning into one framework. Therefore, it provides different value estimation schema's for different values of the λ, enabling us not only to test the validity of the proposed claim that MC methods are ac-

tually more successful than TD learning methods in non-Markovian environments, but also provide a more flexible framework which can work in different problem domains.

Algorithm 6 shows our proposed LSFTDI method. LSFTDI basically changes the way of calculating the state values of the observations in the LSFQI from TD learning to TD(λ). Instead of using the TD learning equation, $r_i + \gamma \max_{a'}[F(\Theta_j)](s'_i, a')$, to assign target values between the lines $5-7$ of the Algorithm 3, we have embedded the TD(λ) algorithm into the LSFQI, shown between the lines $3-10$ of the Algorithm 6. By doing so, we propose a novel Batch RL framework for partially observable environments. Note that the s variables in the Algorithm 3 and 5 are turned into o in Algorithm 6 since LSFTDI works on partially observable domains, which actually subsumes the fully observable domains.

Algorithm 6: Least-Squares Fitted TD(λ) Iteration

Input : discount factor γ, learning rate α, n-step return weight λ, approximation mapping F, experience tuples $\{(o_i, a_i, r_i, o'_i) | i = 1, 2, ..., T\}$

1 $j \leftarrow 0$, $\Theta_j \leftarrow 0$, $\Theta_{j+1} \leftarrow \epsilon$
2 **while** $|\Theta_{j+1} - \Theta_j| \geq \epsilon$ **do**
3 **for** $t = 0, 1..., (T-1)$ **do**
4 **for** $n = 1, 2, ..., (T-t)$ **do**
5 $V_{t+n} \leftarrow \max_{a'}[F(\Theta_j)](o'_{t+n}, a')$
6 $R_t^{(n)} \leftarrow r_{t+1} + \gamma r_{t+2} + ... + \gamma^{n-1} r_{t+n} + \gamma^n V_{t+n}$
7 **end**
8 $R_t^{\lambda} \leftarrow (1-\lambda)[R_t^{(1)} + \lambda R_t^{(2)} + ... + \lambda^{T-t-2} R_t^{(T-t-1)}] + \lambda^{T-t-1} R_t^{(T-t)}$
9 $V_{t+1} \leftarrow V_{t+1} + \alpha(R_t^{\lambda} - V_{t+1})$
10 **end**
11 $\Theta_{j+1} \leftarrow \Theta^*$, where $\Theta^* \in \arg\min_{\Theta} \sum_{i=1}^{N} (V_i - [F(\Theta)](o_i, a_i))^2$
12 $j \leftarrow j + 1$
13 **end**

Output: Θ_{j+1}

As LSFQI does, LSFTDI iteratively trains a parameter vector and outputs the parameter vector when it is converged. This parameter vector will be used to find the approximate action values for each possible observation. Unlike the LSFQI, however, LSFTDI finds a stochastic policy mapping observations into actions probabilistically. This is done by getting the ratio of the approximate action values of observations. We again use the Equation 2.17 to find the approximate action values for each observation. Then, based on the Equation below,

$$\pi(o, a_i) = \frac{[F(\Theta)](o, a_i)}{\sum_{j=1}^{k} [F(\Theta)](o, a_j)} \tag{4.6}$$

where k is the number of actions, we set the probability values for each action of observations. Note that, to avoid zero divisions and distribute probability values proportional to the action values fairly, sometimes, we are required to shift the action values appropriately.

4.5 Batch Mode TD(λ) for Controlling Partially Observable GRNs

This section describes how we adapt our proposed LSFTDI algorithm to solve the control problem of partially observable GRNs based solely on gene expression data without making use of any computational model. What we have done is firstly to convert the time series gene expression data into a series of experience tuples, and extract feature values for each experience tuple. Then, we feed the experience tuples and feature vectors into the LSFTDI algorithm to obtain an approximate infinite horizon stochastic policy. Figure 4.2 shows the block diagram of our solution for controlling partially observable GRNs. The following subsections will describe how we convert gene expression data into a series of experience tuples, and what the features we have used are.

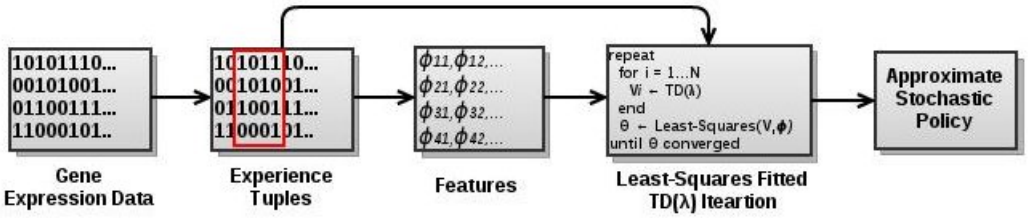

Figure 4.2: Batch TD(λ) for controlling partially observable GRNs

4.5.1 Experience Tuples

This section describes how we convert gene expression data into a series of experience tuples. An experience tuple for a partially observable environment is a 4-tuple (o, a, o', r), where o is the current observation, a is the current action, o' is the next observation and r is the immediate reward. It represents one-step transition in the environment. Here, we explain how each of the four elements of each experience tuple is obtained from the gene expression data.

Observations: In order to define how to obtain observations from the gene expression data, we should firstly explain states. The state of a GRN is defined by the discretized form of the gene expression sample itself [12, 13]. The observation of a GRN, on the other hand, is defined by assuming that we can only observe only a subset of the set of genes in the gene expression data [13, 7]. It is done by specifying the *observation genes* in the context of the control problem. The discretized expression value of the

observation genes of the i^{th} and $(i+1)^{th}$ samples constitute the current observation o and the next observation o' values for the i^{th} experience tuple. Similar to most of the previous studies, we have used binary discretization [13, 7]. There are 2^n possible observations given n observation genes.

Actions: The action semantics for a gene regulation system is mostly implemented through suppressing value of a specific gene or a set of genes, i.e., setting its value as 0 [13, 7]. Those suppressed genes are named as *input genes* and should be specified in the context of the control problem. In order to obtain the action values from the gene expression samples, we have checked the change in values of the input genes from 1 to 0 in the successive gene expression samples. For a regulation system of six genes, for example, let the gene expression sample at time t be 111001 and at time $t+1$ be 001000. If the input genes are the 2^{nd} and 5^{th} genes, the action value, i.e., a in the experience tuple, at time t is 10 in binary representation since the 2^{nd} gene has changed its value from 1 to 0 while the 5^{th} gene has not. There are 2^k distinct actions given k input genes.

Rewards: The only remaining values to be extracted from the gene expression samples is the reward values, i.e., r in the experience tuples. Rewards are associated with the goal of the control problem. Goals can be defined as having the value of a specific gene as 0 as in [13], or as reaching to a specific basin of attractors in the state space as in [7]. If the state of the regulation system satisfies the goal, it is awarded by a constant value κ. Moreover, applying an action for each input gene also has a relatively small cost to realize it. So, the reward function can be defined as follows:

$$R(s,a) = \begin{cases} \kappa - n \times c & \text{if goal(s)} \\ 0 - n \times c & \text{if } \neg \text{ goal(s)} \end{cases} \tag{4.7}$$

where κ is the reward of being in an desirable state, n is the number of input genes whose action value is 1, and c is the cost of action to apply for each input gene.

4.5.2 Features

This section describes the features that are built from the experience tuples obtained from the gene expression samples. We have used the State Features explained in Section 3.2.2. Since, this time we are in partially observable environments, we do not have state information, but observation information. Hence, we have used the current observation values of the experience tuples, i.e., the discretized gene expression values itself, directly as features. That is, for each experience tuple, there are exactly as many features as the number of observation genes, and feature values are equal to the binary discretized expression values of observed genes, which can be formulated as below.

$$\phi_i(o) = \begin{cases} 0 & \text{if } o(i) == 0 \\ 1 & \text{if } o(i) == 1 \end{cases} \tag{4.8}$$

where $0 \leq i \leq n$, n is the number of observed genes and ϕ_i is the i^{th} feature in the feature vector and it is equal to the expression value of the i^{th} gene in the discretized gene expression sample. So, for a 15-gene regulation system in which 6 of them is observable, an observation having binary value as 101011 has its feature vector same as 101011. Note that as suggested in [8], we have used different parameter vectors for each possible action, therefore the action does not affect the feature values in Equation 4.8.

4.6 Experimental Evaluation

4.6.1 Melanoma Application

This section describes our experimental evaluation of the proposed algorithm Least-Squares Fitted TD(λ) Iteration (LSFTDI) for controlling partially observable GRNs. As all the three previous studies for controlling partially observable GRNs, we have used the same melanoma dataset used by Section 3.3.1 [5] to evaluate our algorithm [13, 7, 18]. The dataset contains 31 samples and is composed of 8067 genes. Bittner et al. (2000) reports that WNT5A is highly discriminating factor for metastasizing of the melanoma, and deactivating WNT5A reduces the metastatic effect of melanoma significantly. Therefore, a control strategy keeping WNT5A deactivated is critical to mitigate the metastasis.

We have again considered a 7-gene subset of the complete melanoma dataset, which are WNT5A, pirin, S100P, RET1, MART1, HADHB, and STC2, in order [13]. We have set the κ value in the reward formulation of Equation 4.7 as 10 and set the cost of applying an action, c value in Equation 4.7 as 1. We have set the goal objective of the control problem as having WNT5A deactivated, meaning to have its expression value as 0. We have set our discount factor, γ as 0.9. As LSFQI does, we have set our learning rate $\alpha = 1$. In the following subsections we will investigate the effect of λ for our proposed framework LSFTDI, and present comparative results with all the three previous works in terms of both time and solution quality.

4.6.1.1 Experiments on λ

In this section, we have investigated the effect of λ in our proposed Batch RL framework for partially observable environments. We have set the input gene always as pirin, which is one of the most frequently used input gene in controlling GRNs works [12,

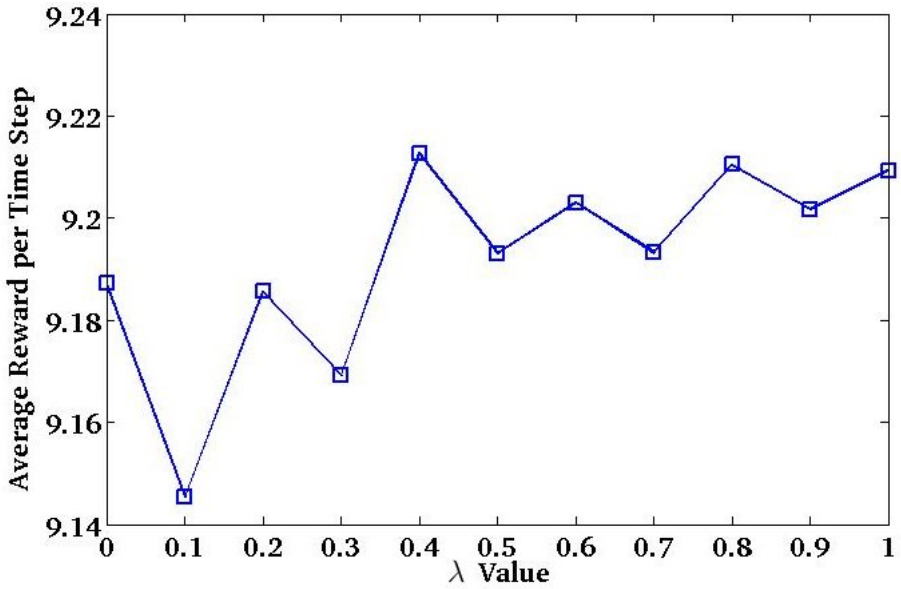

Figure 4.3: Average reward for different λ values

13, 7]. To measure the quality of the stochastic policies for different values of λ, we follow the way of Datta et al. (2004). We have constructed a Probabilistic Boolean Network (PBN) from the 7-gene melanoma dataset and run the stochastic polices produced by our method on the PBN until it reaches to steady state. Then, we have checked the average rewards provided by each set of stochastic policies for different values of λ. For all possible observations in the 7-gene melanoma dataset specified above, consisting of the observations of the length 1, 2, ..., 7 summing up to $2^7 = 128$ different observations, we run LSFTDI on the 7-gene melanoma dataset for different values of $\lambda = 0.1, 0.2, ..., 1.0$, and take the average. Figure 4.3 shows the results.

What we see is that the average reward tends to increase as λ goes to 1 in a zigzag manner. This is consistent with the claims of [61] and [30] that Monte-Carlo (MC) methods are actually better for non-Markovian environments since TD(1) equals to the MC method. However, we also see that the maximum three average rewards, which are 9.2128, 9.2105 and 9.2093, are obtained when we have $\lambda = 0.4$, $\lambda = 0.8$, $\lambda = 1.0$, respectively. This result, on the other hand, confirms our concerns that it is also possible to obtain successful stochastic policies for different values of λ. Because TD(0.4) and TD(0.8) performs better than both TD(0) and TD(1). Therefore, though TD(λ) tends to perform better as λ gets close to 1, it is possible to have better performance than TD(1) for different values of λ for a non-Markovian decision problem.

4.6.1.2 Comparative Results

This section presents our comparative results with the three previous works on controlling partially observable GRNs. In order to evaluate our algorithm effectively, we

Table4.1: Input & Observation genes

	Input Gene(s)	Observation Gene(s)
Exp #1	pirin	WNT5A
Exp #2	pirin	pirin, RET-1
Exp #3	WNT5A	pirin, S100P, HADHB
Exp #4	WNT5A	WNT5A, S100P, MART-1, STC2
Exp #5	pirin	pirin, S100P, RET1, HADHB, STC2
Exp #6	WNT5A	WNTA5, pirin, S100P, MART1, HADHB, STC2
Exp #7	pirin	WNTA5, pirin, S100P, RET1, MART1, HADHB, STC2

have done experiments with different number of observation genes and with different input genes. Table 4.1 shows specifications for each experiment we conduct. For all of the experiments, we set the λ value as 1.0 due to producing one of the three maximum average rewards in the experiments of Section 4.6.1.1. α and γ values are as in specified in Section 4.6.1.

We have compared our work with the three previous studies, Dynamic Programming (DP) method of [13], AO* algorithm of [7], and infinite horizon POMDP (Inf. POMDP) of [18], in terms of elapsed time to produce a control policy and the quality of the produced policy. To compare the quality of the produced policies, we again followed the ways of the previous works. We have constructed a PBN from the 7-gene melanoma dataset, and run it for the different control policies obtained from different solutions. Then, we have checked the average rewards produced by the policies while running the PBN under each of the control policy. Table 4.2 and 4.3 show the results. Note that that the results in Table 4.2 and 4.3 have been obtained by averaging 500 runs of each experiment.

Note that DP of [13] and AO* of [7] are finite-horizon. That means, the horizon should be specified in the context of the control problem. In order to compare those works with our infinite horizon control solution fairly, we have firstly identified maximum horizon that DP and AO* can scale. As it is done in [7], we set the maximum time as 20 minutes and killed the processes exceeding 20 minutes to finish. The first rows of the AO* and DP in Table 4.2 and 4.3 show the maximum horizon, H_{max}, values for the two previous studies. Observe that H_{max} is decreasing exponentially as the number of observation genes increases, both for DP and AO*. This is because both of the algorithm branch for all possible actions and observations from the initial belief state, which takes $O(d^{(h\times(o+a))})$ time and space in worst case, where d is the discretization level, h is the length of the horizon, o is the number of observation genes and a is the number of action genes. Although AO* scales better for Exp#1 and Exp#3 than DP, they are both far from being scalable by scaling up to H_{max} of 8 for the small 7-gene

Table4.2: Comparative results for Exp#$1-4$

		Exp#1	Exp#2	Exp#3	Exp#4
AO*	H_{max}	8	4	3	2
	$AvRew$	6.86	7.05	7.37	6.55
	$Time$	1135.7	170.34	1449.4	262.83
DP	H_{max}	5	4	2	2
	$AvRew$	6.37	7.05	6.40	6.55
	$Time$	442.13	1036.0	22.23	283.56
Inf. POMDP	$AvRew$	4.49	4.53	8.82	8.84
	$Time$	7.6	9.19	8.70	14.02
LSFTDI	$fAvRew$	6.09	6.56	5.70	4.82
	$iAvRew$	9.24	9.29	9.23	9.43
	$Time$	0.07	0.41	0.33	0.26

Table4.3: Comparative results for Exp#$5-7$

		Exp#5	Exp#6	Exp#7
AO*	H_{max}	1	1	1
	$AvRew$	4.55	4.25	4.75
	$Time$	7.68	68.92	312.65
DP	H_{max}	1	1	1
	$AvRew$	4.55	4.25	4.75
	$Time$	1.48	9.76	13.73
Inf. POMDP	$AvRew$	4.57	8.81	4.50
	$Time$	18.08	34.48	101.98
LSFTDI	$fAvRew$	4.53	4.06	4.52
	$iAvRew$	9.27	9.33	9.36
	$Time$	0.43	0.54	1.43

melanoma dataset.

After identifying H_{max} values for the finite horizon works, we obtained control policies from AO* and DP for H_{max} horizon, and from LSFTDI for infinite horizon. Then, we have run the constructed PBN separately for H_{max} number of steps with those control policies. $Time$ rows of Table 4.2 and 4.3 present the required time to construct the policies and $AvRew$ rows present the average reward per time step gathered during the execution of the PBNs under different control solutions. Since LSFTDI produces infinite horizon policies, we can run the PBN with the policy of LSFTDI either until the PBN reaches to steady state, or for a fixed number of steps. $iAvRew$ row of LSFTDI shows the average reward for running until steady state, and $fAvRew$ of LSFTDI row shows the average reward for running H_{max} number of steps of AO*. We see that the required times to construct a control policy is significantly less for

LSFTDI than it is for AO* and DP. LSFTDI always produces infinite horizon policies less than 2 seconds, whereas the finite horizon studies not only get stuck for more than 1 horizon in Exp$\#5-\#7$, but also require much more time to construct a policy even for small horizons. When we compare the *AvRew* values of AO* and *fAvRew* values of LSFTDI, on the other hand, we see that LSFTDI is able to produce near-optimal policies. Both AO* and DP solutions produce the optimal policy for the determined time window, whereas our proposed LSFTDI algorithm produces an approximate and infinite horizon policy. Still, we see that the quality of the policies produced by LSFTDI are comparatively good with respect to the optimal policies of the previous works since the *fAvRew* values are very close to the *AvRew* values of AO*. Hence, we can say that LSFTDI is able to produce control policies with much less required time without losing significant performance. Observe that, in Exp$\#5-7$, AO* requires more time than DP, which is an unexpected result actually, since AO* employs a heuristic to improve DP. However, this case is also reported by [7] in their Section 7.3 with the same melanoma dataset and same experimental settings. The reason is that AO* repeatedly evaluates same belief states due to loose upper bound on the reward value 10. AO* expands the belief states only on the path of the optimal policy, and prunes all the other belief state nodes at each iteration. Once the path of the optimal policy changes more than one time, it may be required to expand the same belief state node again and again, which are the cases for the Exp$\#5-7$.

When we check the *iAvRew* row of LSFTDI, we again observe a very good performance. Since we set the κ value in Equation 4.7 as 10, maximum average reward per time step can be at most 10. The *iAvRew* values are always larger than 9 indicating a very good performance indeed. If we compare *iAvRew*, with the *AvRew* of Inf. POMDP, we see that our method produces always better average reward than Inf. POMDP, and always requires much less time, verifying the effectiveness of our method in infinite horizon as well.

4.6.2 Large Scale Melanoma Application

This section describes the evaluation of our scalable LSFTDI algorithm on a large-scale melanoma dataset. We have used the same 28-gene melanoma dataset presented in Section 3.3.3. We set the input gene as pirin and the observation genes as pirin, MART1, CD63, EDNRB, PGAM1, integrin b1, syndecan4, tropomyosin1, EphA2, GAP43, CD20, RET1. We select the observation genes randomly. λ, γ and α values are as in Section 4.6.1.2.

We run LSFTDI on the 28-gene melanoma dataset with the given setting, and checked the quality of the produced policy by running it on the PBN constructed from the 28-gene melanoma dataset. We run the PBN until it reaches to its steady-state. The average reward per time step we get is 9.3140, which is a very good value when compared with the *iAvRew* values of the Table 4.2 and 4.3.

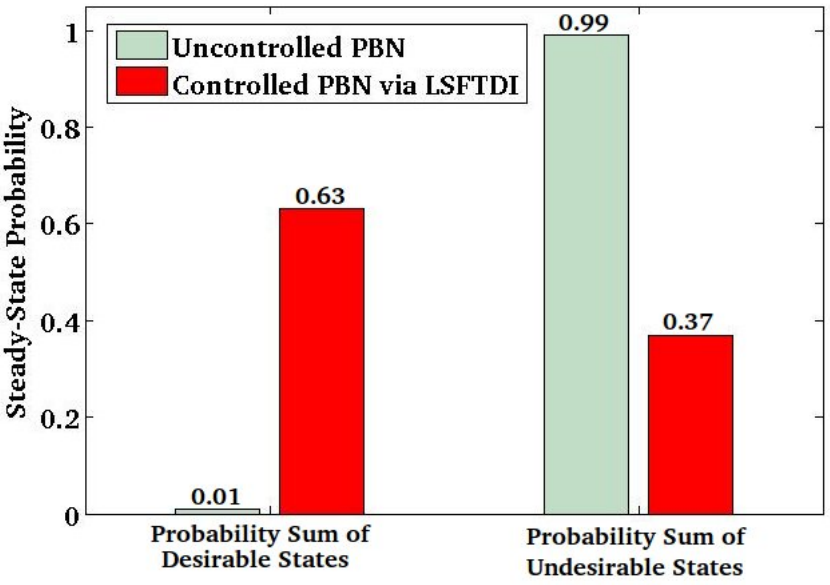

Figure 4.4: Large scale melanoma steady-state probability shift

To understand better whether LSFTDI works properly for the large-scale dataset, we also checked the steady-state probability distribution of the PBN when we run it under the control policy of LSFTDI. Note that we set the WNT5A as the most significant bit in the gene expression data. Therefore, the states of the PBN between $[0-2^{27})$ has WNT5A as deactivated, and the states of the PBN between $[2^{27}-2^{28})$ has WNT5A as activated. Therefore, we can say that the states between $[0-2^{27})$ are desirable whereas, the states between $[2^{27}-2^{28})$ are undesirable. We compared the sum of the probability values of the desirable and undesirable states in the controlled and uncontrolled PBNs. Figure 4.4 shows the results. We see that, the control policy produced by our LSFTDI method is able to shift the probability mass from undesirable states to desirable states significantly. The probability value of being in one of the desirable states increases from 0.01 to 0.63 when we control the PBN with respect to the policy produced by our LSFTDI method, which shows the effectiveness of our method for a partially observable large scale gene regulation systems.

4.6.3 Time Requirements

In this section, empirically, we tried to analyze time requirement of our method for regulation systems of several thousands of genes. We again used the gene expression data presented in [5]. As in Section 3.3.4, we gradually increased the number of observation genes in the dataset from 10 genes to 8067 genes, applied our method and checked the elapsed time to obtain a controlling policy. We set the input gene as pirin, and the λ, γ and α as in Section 4.6.1.2. Figure 4.5 shows the results. We observe that the maximum required time is just about 16 seconds to obtain a control policy for a regulation system of 8067 genes. This is a great improvement since existing studies

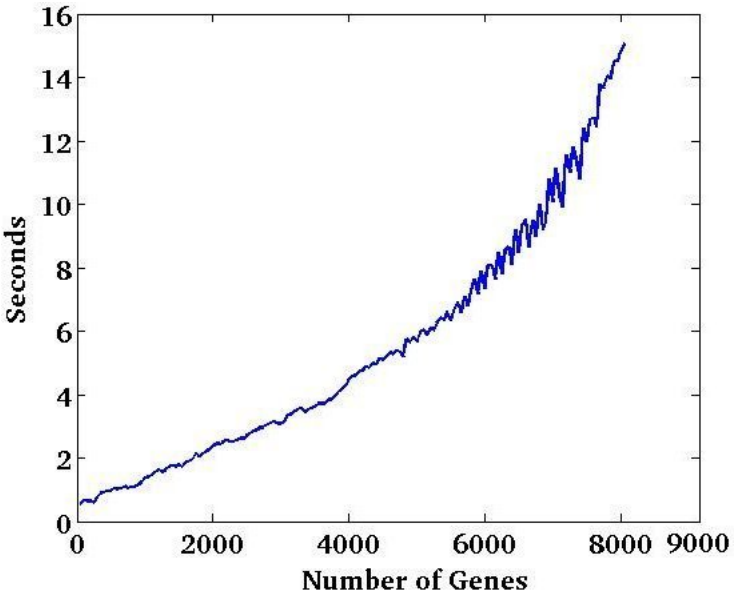

Figure 4.5: Execution time

does not work for the control problems of even several tens of genes. We also observe that the curve in Figure 4.5 presents quadratic behavior, which can be explained by solving a regression problem at each iteration of LSFTDI scaling quadratically with the number of features. We have used MATLAB's pinv (Moore-Penrose pseudoinverse) function for solving the least-squares regression problems in LSFTDI.

Note that in Section 3.3.4 we have mention an important point about the linear behavior of our LSFQI implementation. Although we are applying the same computational process both in this section and Section 3.3.4, we get linear behavior for LSFQI, and quadratic behavior for LSFTDI. We think the reason for getting quadratic behavior for our partial observability solution, LSFTDI, is that the computation time for one iteration of LSFTDI is considerably large comparing that of LSFQI. Because, at each iteration, we apply TD(λ) algorithm spanning all of the experience tuples. Hence, it is enough for LSFTDI to present its original behavior. If we the dataset we have used in Section 3.3.4, for example, had much larger number of genes, such as 100.000, it would most probably present quadratic behavior as well.

4.7 Discussion

In this chapter, we have proposed a novel Batch RL framework for partially observable environments to solve the control problem of partially observable GRNs. Our idea is to interpret gene expression data as a series of observations produced by the gene regulation system we want to control, and obtain an approximate stochastic policy mapping observations to action probabilistically without estimation of the internal states. Results show that our method is able to solve control problems of several thou-

sands of genes in seconds, whereas existing studies cannot solve the control problem of several tens of genes. Results also show that the approximate policies produced by our method has near-optimal quality with respect to the previous finite horizon works and better than the only existing infinite horizon work. Moreover, to our best knowledge, our proposed Batch TD(λ) framework is the first study providing approximate control policies from limited number of experience tuples in partially observable environments. Since Batch TD(λ) is independent of any computational model, it can easily be adapted for different non-Markovian decision tasks, especially when there are limited number of experience tuples.

CHAPTER 5

CONTROLLING GENE REGULATORY NETWORKS: BATCH MODE POMDP LEARNING

5.1 Introduction

In this chapter, we introduce our second proposed algorithm for solving control problem of partially observable GRNs. Since the environment is partially observable, as in Chapter 4, we do not know actual internal states of the gene regulation system. Remember that in our previous solution presented in Chapter 4, we skip to find the actual internal states of the regulation system, and produce stochastic policy mapping observations directly to actions probabilistically. However, in our second solution for controlling partially observable GRNs, we have firstly tried to identify the actual internal states of the system, and then tried to solve the control problem based on the identified internal states and their relations with the observations the system produces. Remember that observations that a gene regulation system produces are consecutive gene expression samples.

In order to do this, we have used the ideas of POMDP learning methods presented in [46]. POMDP learning basically means to apply Reinforcement Learning (RL) in partially observable environments, and learn the POMDP model of the partially observable environments based on the experience tuples collected from the environment, i.e., the interactions between the agent and the environment. The core of the POMDP learning methods is to learn the actual internal states of the partially observable environment. Because, without learning the internal states, solving a POMDP would always be incorrect as explained in Section 2.4. To do this, agent keeps an observation-action function, and checks the statistical differences between the observation-action values for different observations [47, 45, 48, 46]. The idea is that, for the same action, observations having similar observation-action values should be produced from similar states, and vice versa. In our gene regulation control domain, we interpret successive gene expression samples as successive observations produced from the gene regulation system that we want to control, and make use of Batch Mode Reinforcement Learning (Batch RL) methods to obtain approximate observation-action values for each possible observation-action pairs as explained in Chapter 3. Based on those approximate

observation-action values, we tried to identify the actual internal states of the gene regulation system and construct a POMDP model. Note that the problem of identifying the actual internal states of a partially observable environment is also called as hidden state identification problem, since internal states are hidden, and we are trying to identify them [46]. Again, our method does not depend on a specific GRN model such as Probabilistic Boolean Network (PBN), but is directly applied to the available gene expression data. Therefore, it can easily be adapted for different partially observable control problems. Results show that our POMDP construction method is successful as it produces both faster and better solutions than all the previous studies for controlling partially observable GRNs.

The rest of this chapter is organized as follows. Section 5.2 introduces POMDP learning methods and hidden state identification technique we used. Section 5.3 explains our POMDP model for partially observable GRN control problem. Section 5.4 presents the experimental evaluation of our method. Lastly, Section 5.5 concludes with a discussion.

5.2 POMDP Learning

POMDP learning is learning the POMDP model of the environment. That is, an agent who does not know anything about the environment starts to interact with the environment by applying its own actions, and gathers some experience tuples from the environment. Based on those collected experiences, the agent tries to deduce the components of the POMDP model. The common way to do this is to assume an hypothetic POMDP including hypothetic set of states, state transition function, observation function and then to refine them statistically based on the collected experiences [46]. Remember that a POMDP is a 6-tuple (S, O, A, T, Ω, R), where S is the finite set of states; O is the finite set of observations; A is the finite set of actions; $T(s'|s, a)$ is the transition function which defines the probability of observing state s' by firing action a at state s; $\Omega(o|s)$ is the observation function which defines the probability of observing observation o at state s; and $R(s, a)$ is the immediate reward received in state s firing action a [9].

Let us review the components of the POMDP and try to understand which components the agent should learn, and how this will happen. The core part of POMDP learning tasks is to learn the internal states of the partially observable environment. This mainly because of the fact that without knowing the internal states of the partially observable environment we cannot apply the available RL methods to find a policy as explained in Section 2.4. Note that the problem of identifying internal state of a partially observable environment is called as hidden state identification problem [46]. So, the set of states, S, component is the most critical component for the agent to learn. In order to do this, initially, agent assumes each newly encountered observation is produced from a different internal state. Then, it collects a set of experience tuples

and updates its observation-action function just like it is done by Q-Learning shown in Algorithm 2. After collecting enough number of samples, agent checks the statistical difference between the observation-action values for different observations. The idea is that, for the same action, observations that do not have statistically significant difference between their observation-action values should be produced from the same internal states. That leads to merge the assigned internal states into one internal state for the observations identified as having no statistically significant difference between their observation-action values. This operation is named as merge operation. If there is statistically significant difference between the observation-action values, on the other hand, the observations are assumed to be produced from different internal state dynamics. That leads to split the assigned internal state for the observations having statistically significant difference between their observation-action values, if they were assigned to be produced from the same state somehow in the learning process. Thereby, POMDP learning methods is able to find the set of states component of the POMDP model [46]. The set of observations O is learned by the observations coming the environment. The set of actions A is known since actions are internally defined by the agent. The transition function T is not known. In order to find it, POMDP learning methods just counts the state transitions in the collected experience tuples, and normalizes it. The observation function Ω is not known. It is again learned by counting the observations observed from each identified internal state. Note that transition and observation functions are recalculated when a new internal state is discovered and added to the set of state component. The reward function R is learned by the reward values obtained from the environments.

There are several well-defined hidden state identification algorithms. The Perceptual Distinction Approach (PDA) proposes to split the collected experience tuples into two parts and measure the statistical difference between the observation-action values of the observations seen in the first and second part of the experiences [11]. Utile Distinction Memory (UDM) algorithm, on the other hand, checks the statistical difference between the observation-action values of different observations and uses the idea that similar states would produce similar observation-action values [47]. Utile Suffix Memory (USM) builds a suffix tree from the history of the agent, so that each leaf of the suffix tree corresponds to one state of the environment. The suffix tree is split not only based on the statistical differences between the observation-action values, but also path of of the history chain of the experience tuples [48]. Nearest Sequence Memory (NSM) algorithm again makes use of the history of the agent. For each newly encountered observation, its idea is to average the observation-action values of the k previous observations whose history chains are most similar to the history chain of the newly encountered observation [45].

5.2.1 Hidden State Identification

This section describes how we capture the statistical significance between the observation-action values. The basic difference between the POMDP learning methods is the way they capture the statistical difference. PDA, USM and NSM extensively uses the history of the agent. UDM, on the other hand, checks the statistical difference between observation-action values without making use of any history information, which makes it less successful than USM and NSM, but much more applicable for our problem domain. Hence, we have used UDM's hidden state identification technique.

UDM computes a confidence interval for each observation-action value based on the Equation 5.1.

$$q \pm t_{\alpha/2}^{(n-1)}(\frac{s}{\sqrt{n}}) \tag{5.1}$$

where q is the observation-action value for a specific observation-action pair, $t_{\alpha/2}^{(n-1)}$ is the Student's t function with $n-1$ degrees of freedom and $\alpha/2$ confidence level, s is the standard deviation of the observation-action values and n is the number of observations. If, for the same action, the calculated confidence intervals intersect for a set of observations, it is assumed there is not statistically significant difference between the observations. Hence, a merge operation is applied. If confidence intervals do not intersect for a set of observations, they are assumed to be produced from different system dynamics. Hence, a split operation is applied.

Figure 5.1 shows a system before a merge operation; and Figure 5.2 shows the same system after the merge operation. After merge/split operations, the probability value of observing an observation are equal for all observations produced from the same state. Here, it is important to note that the merging process is NP-complete, but greedy merging works well enough [47]. At the first step, we merge all of the states having overlapped confidence intervals with the first state. Next, we do merging with the k^{th} state, which is the first state that has not been overlapped with the first state, and we continue so on.

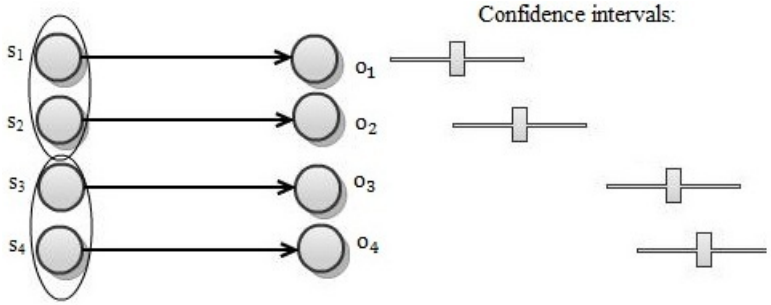

Figure 5.1: System before merge

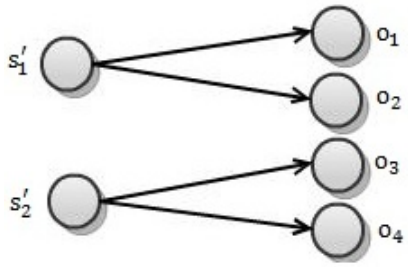

Figure 5.2: System after merge

5.3 Batch Mode POMDP Learning for Controlling Partially Observable GRNs

This section describes our novel Batch Mode POMDP Learning method building a POMDP model directly from gene expression data for controlling partially observable GRNs. Figure 5.3 depicts the flow our approach. What we have done is to convert gene expression data into experience tuples, extract feature values from the experience tuples, and apply Least-Squares Fitted Q Iteration (LSFQI) to obtain an approximate and generalized observation-action function, just like it is done in Chapter 3, Section 3.2. Then, based on the approximate and generalized observation-action function coming from LSFQI, we have applied the hidden state identification technique explained in Section 5.2.1, which enables us to learn the internal state of the gene regulation system, and construct the ultimate POMDP model. Note that we have used State Features for LSFQI explained in Section 3.2.2. Note also that since we assume gene regulation system we want to control is partially observable, the gene expression samples constitute observations the regulation system produced, and the approximate state-action function coming from LSFQI is approximate observation-action function, indeed. Below, we have explained how each of six component of the POMDP model, (S, A, O, T, Ω, C), is obtained.

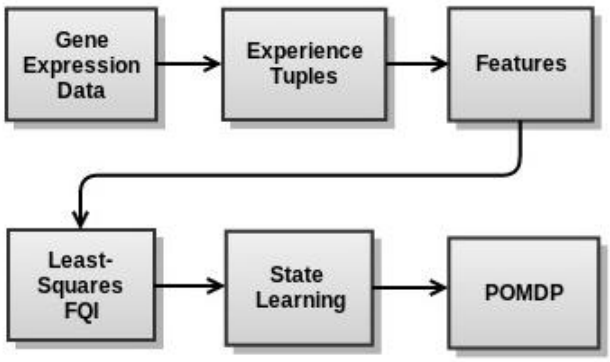

Figure 5.3: Flow of our POMDP construction method

Set of States: We have learned the set of state component by applying the hidden state identification technique explained in Section 5.2.1. In order to this, we have applied the same procedure explained in Chapter 3, Section 3.2 with State Features upto LSFQI application process. Due to our problem domain, we have only a series of gene expression samples, whose size is generally very small. Hence, the procedure explained in Section 3.2 provides an approximate and generalized observation-action function which enables us to apply the hidden state identification technique explained in Section 5.2.1. Based on those approximate observation-action values, we cluster the observations into states, and get the set of state component along with their Bayesian relationship with the observations as Figure 5.2 shows.

Set of Observations: Each gene expression sample is the concatenation of the continuous gene expression values. We assume the successive gene expression samples are the observations that the gene regulation system produces. So, the set of observations is the set of all possible values of gene expression samples. Since there would be infinitely many number of observations if we use continuous values for gene expressions, we discretized gene expression values. As most of the previous studies have done, we used binary discretization [13, 7, 18]. Hence, if there are n genes, then the number of all possible observations, i.e., the length of the set of observations $|O|$, is 2^n. Note that this definition is same as the state definition explained in Section 3.2.1. Since our problem definition in this chapter assumes gene regulation system is partially observable, gene expression samples constitutes observations coming from the regulation system, rather than the actual internal states.

Set of Actions: In GRN control problem, action semantics is defined in terms of input genes. An input gene is the gene that the agent in the laboratory applies its actions. The set of input genes are defined in the scope of the GRN control problem. Action application, on the other hand, is to reverse the expression value of the input genes or not reversing them. Since, we use binary discretization, action 1 means to reverse the value of the input gene from 0 to 1, or 1 to 0; and action 0 means not to interfere the value of the gene. There may be more than one input genes. The number of all possible actions, i.e., the length of the set of actions $|A|$, is 2^k if there are k input genes.

State Transition Function: The state transition function is obtained by counting the number of state transitions in the gene expression data. Based on the Bayesian relationship between states and observations, we have just counted the states observed in the gene expression data. For the states that are not observed in the gene expression samples, it is assumed that they have transition to themselves with probability 1.0.

Observation Function: Based on the Bayesian relationships between states and observations, we have calculated the observation probabilities as distributed equally among the observations produced from same state. For example, if there are five observations produced from the same state s, the probability of observing o_i, for $1 \leq$

$i \leq 5$, in s, $\Omega(o_i|s) = 0.2$.

Cost Function: Costs are associated with the observations, i.e., gene expression samples with respect to whether they satisfy the goal objective of the control problem or not. A POMDP, however, requires to associate costs with states of the system. In order to do this, for each state, we have take the probabilistic expectation of the costs of the observations that can be produced from that state. As in Section 3.2, we set a constant penalty for observations not satisfying the goal and find the expected cost value for each state with respect to the observations produced from that state. Equation 5.2 and 5.3 show the cost formulation.

$$cost_{observation}(o, a) = \begin{cases} 0 + n \times c & \text{if goal(o)} \\ \kappa + n \times c & \text{if } \neg \text{ goal(o)} \end{cases} \tag{5.2}$$

$$cost_{state}(s, a) = n \times c + \sum_{i}^{k} \Omega(o_i|s) \times cost_{observation}(o_i, a) \tag{5.3}$$

where κ is the penalty of being in an undesirable state, n is the number of input genes whose action value is 1, c is the cost of action to apply for each input gene, and k is the number of observations that can be produced from state s. To illustrate, if a state can produce 2 of 5 observations not satisfying the to goal, the total penalty for that state is multiplied by 0.4. Goal of a control problem can be having value of a specific gene as 0 or reaching to a desirable basin of attractors.

5.4 Experimental Evaluation

5.4.1 Melanoma Application

This section describes the experimental evaluation of our novel Batch Mode POMDP Learning method for controlling partially observable GRNs on the melanoma dataset presented in [5]. It is reported that WNT5A gene is highly discriminating factor for metastasizing of melanoma and deactivating the WNT5A significantly reduces the metastatic effect of WNT5A [5, 12, 7]. Hence, a control strategy for keeping the WNT5A deactivated may mitigate the metastasis of melanoma [12].

We have used the same LSFQI setting explained in Section 3.3.1 to obtain approximate observation-action function. We have used seven most significant genes in the complete melanoma dataset, which are WNT5A, pirin, S100P, RET1, MART1, HADHB, and STC2, in order [13, 7, 18]. Therefore WNT5A is the most significant bit, and STC2 is the least significant bit in binary observation encoding. We have selected 2^{nd} gene, pirin, as input gene. That is, there are two possible actions defined in the control problem, reversing the value of pirin, or not reversing it. We set the cost of applying

an action as 1. We have defined the goal objective to have WNT5A deactivated and set penalty of not satisfying the goal as 10. Hence, the cost formulation in Equation 5.2 is as following.

$$cost(o,a) = \begin{cases} 0 & \text{if goal(o) and a} = 0 \\ 1 & \text{if goal(o) and a} = 1 \\ 10 & \text{if } \neg \text{ goal(o) and a} = 0 \\ 11 & \text{if } \neg \text{ goal(o) and a} = 1 \end{cases} \tag{5.4}$$

Note that for observations between $0-63$, WNT5A has the value of 0 and for the remaining observations $64-127$, WNT5A is 1. Hence, we can say that observations between $0-63$ are desirable while observations $64-127$ are undesirable. We set the α value as 0.05 in Equation 5.1. Note that unlike our previous partially observable solution explained in Chapter 4, here, we did not do different experiments for different number of observation genes. Instead, we assumed all the genes included in the gene expression data constitutes the observation genes. So, for the melanoma dataset we are using, the number observation genes is 7. It corresponds to the Exp#7 of the experimental evaluation of our previous solution shown in Section 4.6.1.2.

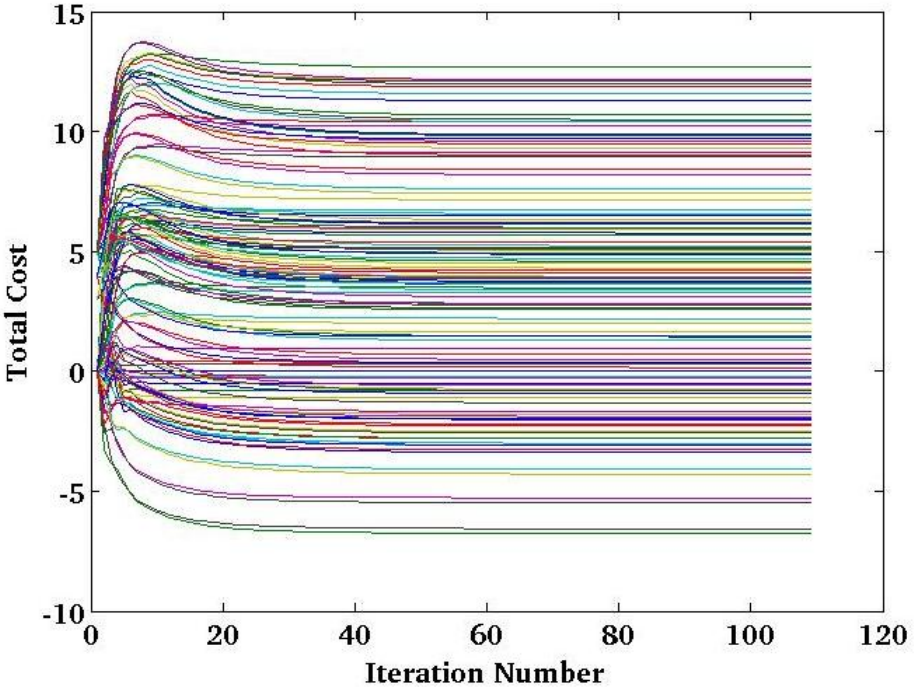

Figure 5.4: State values obtained from Batch RL

Based on these experimental settings, we run LSFQI (LSFQI) algorithm on the 7-gene melanoma dataset, and obtain the approximate and generalized observation-action function. Figure 5.4 shows the observation values over the iterations of LSFQI. Note that each line stands for a different observation. Hence, there are 128 different lines in Figure 5.4 as required for a 7-gene regulation system. Based on those observation

values, we applied the hidden state identification technique explained in Section 5.2.1 and construct the POMDP model for controlling the 7-gene melanoma regulation system. We observe that we learned 26 hidden states based on the observation values shown in Figure 5.4. We have used the point-based POMDP solver Persues [62] to solve the POMDP we learned. We have compared our method with the previous finite horizon methods, Dynamic Programming (DP) of Datta et al. (2004), AO* algorithm of Bryce and Kim (2010), the previous infinite horizon POMDP solution of Erdogdu et al. (2011), and our proposed solution LSFTDI for controlling partially observable GRNs explained in Chapter 4, in terms of average reward per time step, and required time to construct a policy.

Figure 5.5 shows the average reward per time step produced by different solutions for controlling partially observable GRNs. As shown, our novel POMDP model is able to provide much better average reward per time step than all of the previous works, AO* of [7], DP of [13] and previous POMDP solution of [18]. Here it is important to note that the finite horizon works of AO* and DP can only scale upto 1 horizon as explained in Section 4.6.1.2, shown in Exp#7 of the Table 4.3. Remember that we set the maximum time limit as 20 minutes, and killed the processes exceeding 20 minutes. So, the average rewards for the finite horizon works are just for horizon 1, which is the reason for our POMDP model to produce greater average reward then the finite horizon works. Because finite horizon works are provably optimal. Yet, the shown results confirm that the POMDP model we constructed works well enough to produce more the average reward per time step than the two finite horizon works. Moreover, our POMDP model exceeds the infinite horizon POMDP solution of [18], as well. Hence, we can say that the POMDP we constructed directly from gene expression data based on LSFQI is successful enough to produce better policies then the previous finite and infinite horizon works. The only method our POMDP model cannot exceed is LSFTDI, which is the other solution we have proposed for controlling partially observable GRNs. Hence, once again we have verified that LSFTDI is a very successful method providing almost optimal solutions as explained in Section 4.6.1.2 as well.

Figure 5.6 shows the execution times for constructing policies for different solution methods. As it is seen AO* perform much worse than all of the methods. Although DP solves the control problem in 13.73 seconds, this is the result for only horizon 1. This also means that, for horizon 2, DP cannot produce a solution within 20 minute time. Hence, we can say that the finite horizon works of AO* and DP search entire belief state inefficiently. For the previous infinite horizon work of [18], our infinite horizon POMDP requires less time to construct a control policy. Since, it has also been showed that our POMDP produces better average reward then the POMDP solution of [18], we can say that our novel POMDP solution is better than the previous infinite horizon POMDP work of [18] in terms of both time requirements and solution quality. For the finite horizon works, we can again say that our method provides reasonably good policies providing larger average reward per time step than the finite horizon works of

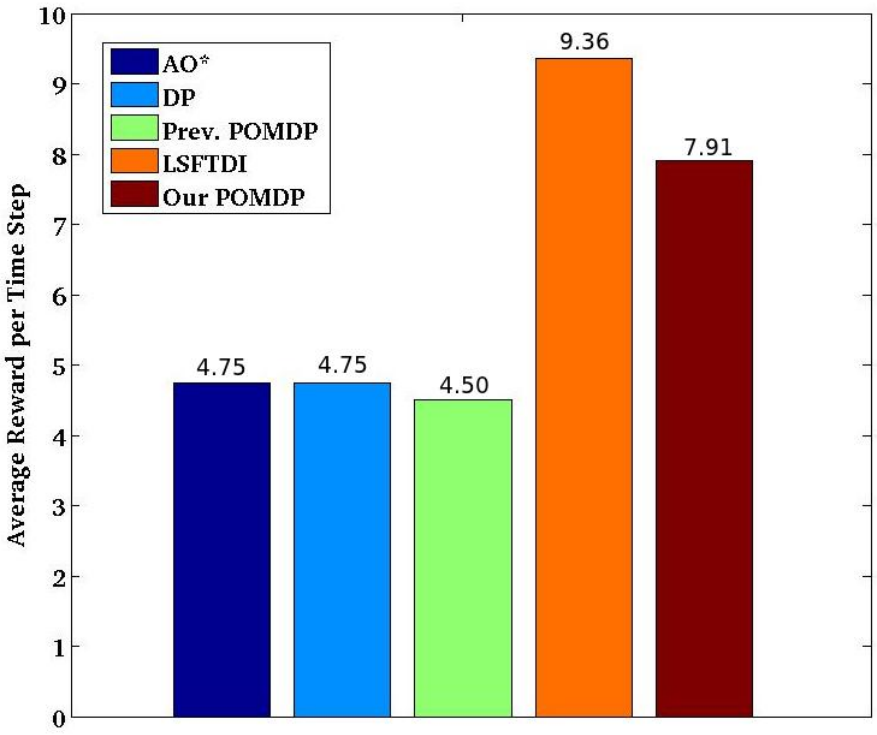

Figure 5.5: Average reward per time step

[13] and [7], and requires much less time providing not a finite both an infinite horizon policy. Again, the only method that our POMDP solution cannot exceed is LSFTDI, which is the other partial observability solution we proposed. LSFTDI requires only 1.43 seconds to provide a stochastic policy producing 9.36 average reward per time step.

Observe that AO* requires more time than DP, which is an unexpected result actually, since AO* employs a heuristic to improve DP. However, this case is also reported by [7] in their Section 7.3 with the same melanoma dataset and same experimental settings. The reason is that AO* repeatedly evaluates same belief states due to loose upper bound on the reward value 10. AO* expands the belief states only on the path of the optimal policy, and prunes all the other belief state nodes at each iteration. Once the path of the optimal policy changes more than one time, it may be required to expand the same belief state node again and again, which are the cases for the Exp#$5-7$ of Section 4.6.1.2 as well.

5.4.2 Yeast Application

This section describes the experimental evaluation of our POMDP model for controlling partially observable GRNs on the same 10-gene yeast cell cycle dataset we have used in Chapter 3, Section 3.3.2. We have used the same LSFQI setting explained in Section 3.3.2 to obtain approximate observation-action function. We have selected

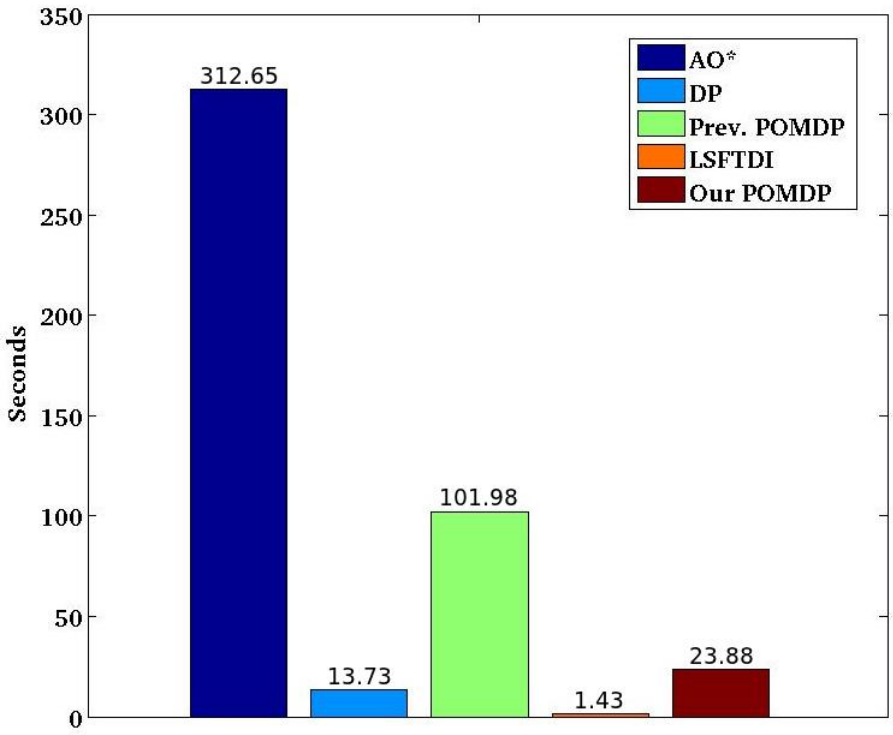

Figure 5.6: Execution time

5^{th} gene, MBP1, as input gene. That is, there are two possible actions defined in the control problem, reversing the value of MBP1, or not reversing it. We set the cost of applying an action as 1. We have defined the goal objective to have MCM1 deactivated and set penalty of not satisfying the goal as 10. Hence, the cost formulation is same as the one in Equation 5.4.

Note that for observations between $0 - 511$, MCM1 has the value of 0 and for the remaining observations $512 - 1024$, MCM1 is 1. Hence, we can say that observations between $0 - 511$ are desirable while observations $512 - 1024$ are undesirable. We set the α value as 0.05 in Equation 5.1. Note that unlike our previous partially observable solution explained in Chapter 4, here, we did not do different experiments for different number of observation genes. Instead, we assumed all the genes included in the gene expression data constitutes the observation genes. So, for the melanoma dataset we are using, the number observation genes is 10.

Based on these experimental settings, we run LSFQI (LSFQI) algorithm on the 10-gene yeast cell cycle dataset, and obtain the approximate and generalized observation-action function. Figure 5.7 shows the observation values over the iterations of LSFQI. Note that each line stands for a different observation. Hence, there are 1024 different lines in Figure 5.7 as required for a 10-gene regulation system. Based on those observation values, we applied the hidden state identification technique explained in Section 5.2.1 and construct the POMDP model for controlling the 10-gene yeast regulation system. We observe that we learned 48 hidden states based on the observation values shown

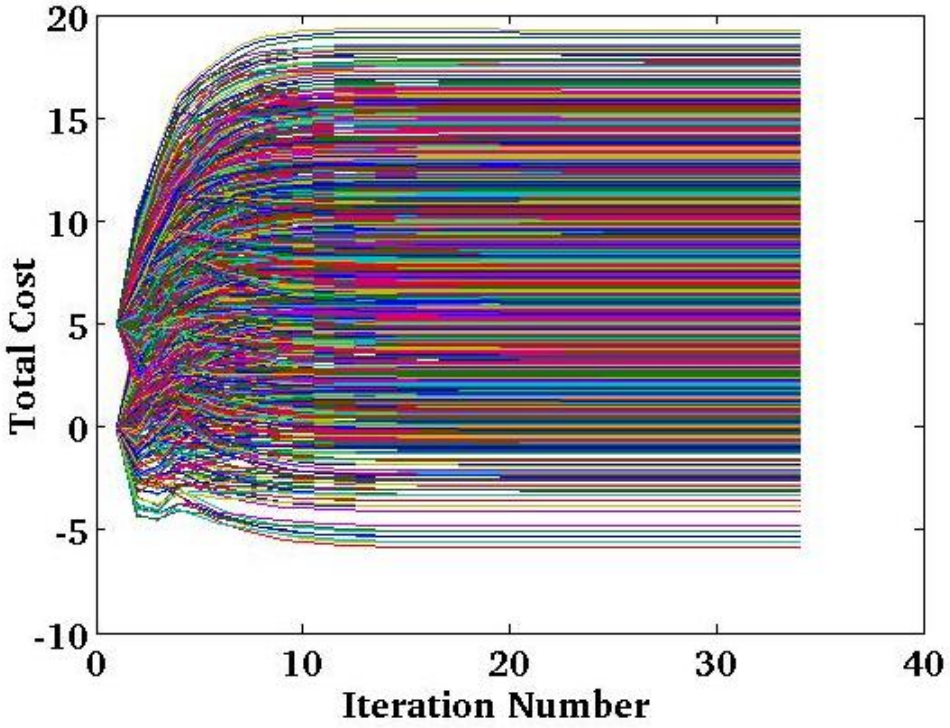

Figure 5.7: State values obtained from Batch RL

in Figure 5.7. We have used the point-based POMDP solver Persues [62] to solve the POMDP model we constructed. We have compared our method with the previous finite horizon methods, Dynamic Programming (DP) of Datta et al. (2004), the previous infinite horizon POMDP solution of Erdogdu et al. (2011), and our proposed solution LSFTDI for controlling partially observable GRNs explained in Chapter 4, in terms of average reward per time step, and required time to construct a policy. Note that we could not compare our method with AO* algoritm of Bryce and Kim (2010) due their extreme time requirement. We have run AO* on the 10-gene yeast cell cycle dataset for horizon 1, and wait more than 12 hours for AO* to finish. However, we could not get any result. We anticipate the reason is again the repeated belief estimation of AO*, which made it perform worse in terms of time requirements than DP in Section 5.4.1 as well.

Figure 5.8 shows the average reward per time step produced by different solutions for controlling partially observable GRNs. As shown, our novel POMDP model is able to provide better average reward per time step than all of the previous works, DP of [13] and previous POMDP solution of [18], and our previous solution LSFTDI as well. The finite horizon work DP can only scale upto 1 horizon as explained in Section 4.6.1.2. So, the average rewards for DP is just for horizon 1, which is the reason for our POMDP model to produce greater average reward then DP. Because DP is provably optimal. Yet, the shown results confirm that the POMDP model we constructed works well enough to produce more the average reward per time step than DP. Our POMDP model exceeds the infinite horizon POMDP solution of [18], as well.

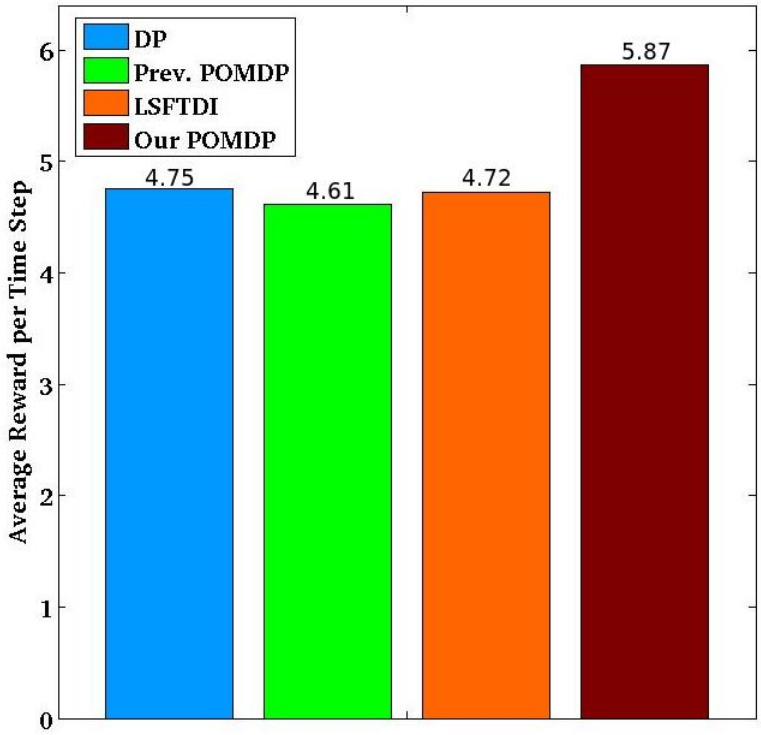

Figure 5.8: Average reward per time step

Hence, we can say that the POMDP we constructed directly from gene expression data based on LSFQI is successful enough to produce better policies then the previous finite and infinite horizon works. Unlike the results in Section 5.4.1, this time our POMDP model exceeds the other method we have proposed to solve control problem of partially observable GRNs, LSFTDI. That means, while both of the control solutions we have proposed to control partially observable GRNs produce better control policies than the existing works both for melanoma and yeast datasets, they cannot beat each other in terms of the solution quality they provide. While LSFTDI is works more successfully with melanoma dataset, our POMDP construction method produces better control policies with yeast cell cycle dataset.

Figure 5.6 shows the execution times for constructing policies for different solution methods. As it is seen DP performs much worse than all of the methods. It requires more than 1.5 hours to find a control policy for the 10-gene yeast dataset with horizon only 1. Hence, we can again say that the finite horizon work DP searches entire belief state inefficiently. For the previous infinite horizon work of [18], our infinite horizon POMDP requires much less time to construct a control policy. Since, it has also been showed that our POMDP produces better average reward then the POMDP solution of [18], we can again say that our novel POMDP solution is better than the previous infinite horizon POMDP work of [18] in terms of both time requirements and solution quality. For the finite horizon works, we can again say that our method provides reasonably good policies providing larger average reward per time step than the finite

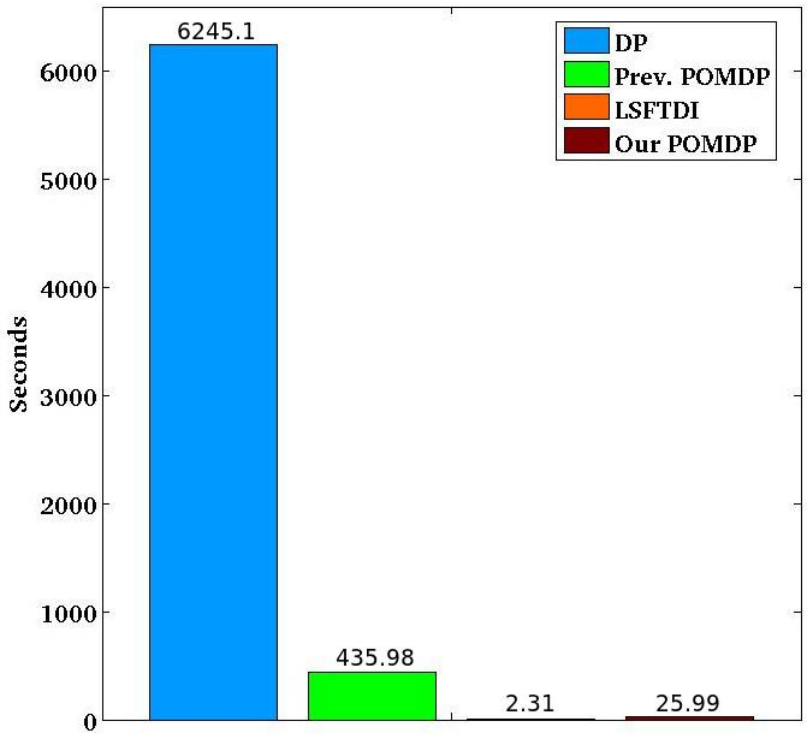

Figure 5.9: Execution time

horizon works of [13] and [7], and requires much less time providing not a finite both an infinite horizon policy. That actually means, our proposed POMDP solution, together with our other proposed solution LSFTDI, makes solvable many gene regulation control problems for systems larger than several tens of genes. Again, the only method that our POMDP solution cannot exceed in terms of time requirement is LSFTDI, which is the other partial observability solution we proposed. LSFTDI requires only 2.31 seconds to provide a stochastic policy producing 4.72 average reward per time step. Note that since our POMDP solution requires reasonable amount of time, 25.99 seconds, to find the control policy, it is acceptable to use for yeast application since it provides more average reward than LSFTDI. However, as the scale of the regulation system gets larger, indispensably, our POMDP solution will require much more time than LSFTDI. This is basically due to the fact that, we have to solve a POMDP to construct a policy which is intractable idealistically [9, 62].

To our best knowledge, together with our other partial observability solution LSFTDI, our POMDP construction method are the two only studies providing approximate control policies from limited number of experience tuples in partially observable environments. Since both of our methods does not depend on any computational model such as Probabilistic Boolean Network (PBN) for modeling GRNs, they both can easily be adapted for different non-Markovian decision tasks, especially when there is limitedly available experience tuples.

5.5 Discussion

In this chapter, we have proposed a novel algorithm, Batch Mode POMDP Learning, for controlling partially observable GRNs constructing a POMDP directly from gene expression data. We have assumed gene expression samples are the observations that the gene regulation system produces and the actual states are hidden. Our method is based on hidden state identification techniques checking statistical difference between observation-action values. The idea is that, for the same action, observations having similar observation-action values should be produced from similar states. We have used one of the Batch RL algorithms, LSFQI, to obtain an approximate and generalized observation-action function from gene expression data for each possible observation. We have applied hidden state identification techniques to the approximate observation-action values coming from LSFQI for learning the actual internal states of the gene regulation system we want to control. Based on the learned internal states and their Bayesian relationship with the observations, we have constructed the ultimate POMDP model. Results show that our POMDP construction method is better than both finite and infinite horizon previous control solutions for partially observable GRNs in terms of both solution quality and time requirements. Our other proposed control solution for partially observable GRNS, LSFTDI, on the other hand, worked better for melanoma dataset and worse for yeast dataset from our constructed POMDP in terms of solution quality. Since LSFTDI does not deal with internal states of the gene regulation system, it always provides the fastest solution among all the previous works and our POMDP construction method.

CHAPTER 6

MULTI-MODEL GENE EXPRESSION DATA ENRICHMENT FRAMEWORK

6.1 Introduction

Gene expression data has vital importance for genome research. In fact most of the research in genome research is driven by the gene expression data, and it is generally difficult to come up with datasets having large number of samples. Further, for small datasets the number of genes is very high compared to number of samples. Unfortunately, the low number of samples is a problem for computational methods. The low number of samples usually decreases the confidence levels in the results of the computational methods that work on gene expression data [55, 10, 25, 35]. Datasets might draw an accurate picture of the underlying genome mechanics, however computationally, low number of samples is a challenge regardless of the expressive power of the data. Several domains, especially health informatics and molecular biology research is effected by the low number of samples and the knowledge discovery task becomes more challenging. An example domain where low number of samples forms a significant problem is cancer biomarkers prediction. In such studies predictive gene sets constructed by different research groups have very small number of similar genes in general. Thus, high number of samples is necessary to construct reliable predictive gene lists [16]. Another example domain is control domain which we have studied in Chapter 3, Chapter 4 and Chapter 5. All of the solutions we proposed to control GRNs are directly based on the gene expression data, and number of available gene expression samples certainly determines our boundaries on the quality of the control solutions.

There are several studies proposed to overcome the problem of low number of samples of gene expression datasets. For instance, the works described in [39, 52, 68, 59] try to obtain a formula for the number of required samples based on different experimental parameters and apply the most appropriate experiment to increase the available samples. The study in [39] proposes to repeat the microarray experiments to increase the number of samples. The studies described in [19, 20], have applied different types of generative models and try to enhance the available data by combining the simulated

results from the different generative models. Once they sample different generative models separately, they evaluate each sample, and selects the best ones to output. However, both [19] and [20] employ a weak sample selection mechanism. Their sample selection mechanism determines the quality of the generated samples based on a linear combination of the metrics that cannot be transformed to each other. Furthermore, in the experimental evaluations, the samples used for model building have also been used in the metric evaluation. In other words, the training and test sets were identical, which is a crucial drawback and makes the experimental justification also weak and unreliable.

In this chapter, we have proposed a robust multi-model gene expression data enrichment framework. We have continued the line of research initiated by [17]. Our main observation is that although there are many gene regulation models in the literature, none of them is able to perfectly capture the system dynamics of gene regulation. Hence, we have decided to built a comprehensive multi-model gene expression data generation framework by integrating four different computational generative models, in which each are responsible for carrying different types of relations from existing gene expression data. Whatever the model is, single gene regulation model will most likely converge to its own system dynamics. However, by integrating different models, we are able to benefit from different models concurrently and activate the most successful one according to the contextual information that the systems demonstrate. Our framework firstly builds four different gene regulation models. Then, it generates samples from all the constructed models and pool the generated samples. The pool of generated samples is a rich source of gene expression data carrying various characteristics of gene regulation. To utilize the diverse types of generated gene expression samples effectively, our framework applies a multi-objective sample selection mechanism based on three defined metrics providing to evaluate generated samples from different aspects. Each generated sample is evaluated multi-objectively based on these metrics. The samples having the best scores in terms of their metric results are outputted by our multi-model data generation framework at the final stage. Results show that our multi-model framework is certainly much more effective than single computational models it includes. Moreover, the produced samples by our framework is so valuable that it can even capture new biological relations that can not be captured by real gene expression datasets. We have also proposed a bound for the number of required samples to train our multi-model framework, which may give a bound for the cost of the real life gene expression data generation experiments.

The rest of this chapter is organized as follows. Section 6.2 describes the multi-model gene regulation approach, the defined metrics to evaluate the produced samples from different models, and the multi-objective selection mechanism. Section 6.3 describes the formulation of the four generative models. Section 6.4 presents the the experimental setup. Section 6.5 presents the experimental results justifying the effectiveness of our proposed sample generation method. Section 6.6 concludes with a discussion.

6.2 Multi-Model Approach

Multi-model gene regulation means to integrate different gene regulation models into one unified framework. There are many gene regulation models, such as Probabilistic Boolean Networks (PBNs) or Ordinary Differential Equations (ODEs) [60, 1]. However, each of them has different intrinsic drawbacks to simulate gene regulation. Under different contextual information, different models may present very different results. For one type of dataset, for example, ODE may simulate system dynamics more successfully than PBN; whereas, for another type of dataset, PBN may simulate system dynamics more successfully than ODE. In their study Hurley et al. (2011) [29] and Marbach et al. (2012) [43], propose to infer Genetic Regulatory Networks (GRNs) based on several GRN inference algorithms, such as ARACNE, BANJO, MIKANA and SiGN-BN [44, 70, 36, 73], and to combine the inferred networks for capturing relationships more clearly and successfully, which is actually named as wisdom of crowds by [43]. In our work, on the other hand, we also propose to compare and combine different gene regulation models, not over the inferred networks, but over the generated gene expression samples. We construct alternative gene regulation models and try to produce high quality gene expression data based on the combination of samples. That means we propose a new way to infer a better GRN, indeed. Instead of combining different inferred networks as in [29] and [43], or improving single GRN inference algorithm [73], we propose to improve the size and quality of the available gene expression data by combining gene expression samples from different computational models for gene regulation. We enrich gene expression dataset artificially, and infer more realistic GRNs based on those artificially enriched set of gene expression samples.

The block diagram of our proposed framework is shown in Figure 6.1. For generating k samples from our multi-model framework, we generate k different samples from each of four generative models, which are Probabilistic Boolean Networks (PBNs), Ordinary Differential Equations (ODEs), multi-objective Genetic Algorithm (GA) and Hierarchical Markov Model (HIMM). Then, we evaluate each generated samples based on well-defined three metrics and employ a multi-objective sample selection mechanism over the three metrics to select *the best* k samples from the $4k$ samples collected from the different generative models. For the sample selection phase, each generated sample is evaluated by three metrics that are calculated both by using training and test data. Final samples are determined by a multi-objective selection mechanism. This mechanism determines the quality of the generated samples separately on all metrics and then ranks them in a multi-objective way. In the last step, the highest scoring samples are selected for inclusion in the newly generated dataset. Note that, as shown in Figure 6.1, except the ODE model, all of the three models works on discrete domain. That is, they both use and produce discretized gene expression samples. Hence, to build the PBN, GA and HIMM models, we used binary discretized values of available gene expression samples and convert the generated binary gene expression samples from the models into continuous samples just before feeding them into selection

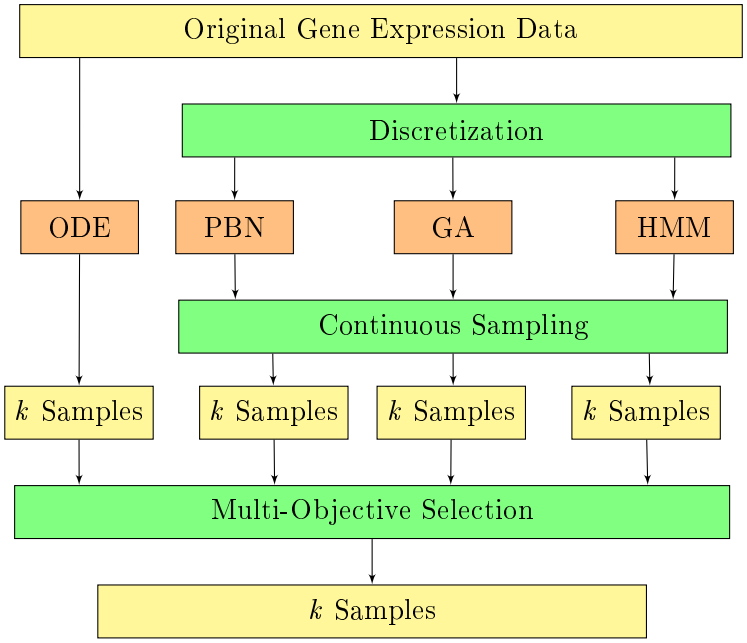

Figure 6.1: Block diagram of our sample generation method.

mechanism. We have also applied a rich set of experiments for proving the effectiveness of our multi-model framework and the quality of the generated samples from our multi-model framework.

The four models employed in the process are not closely related so that each model might contribute its own characteristics to the generated data. The first model is the PBN [60]. This model is an extension of boolean networks [34]. The second model is ODE formulation of gene regulation system. This model tries to find a regulation function for the dynamics of each gene in the regulation system. Hence, the dynamic system of each gene is associated with all other genes in terms of the internal effects on each other. There are several studies about modeling gene regulations by using ODEs. The work described in [26] proposed an algorithm named 'Network Identification by Multiple Regression' (NIR) by applying multiple regression to the system of ODEs. It requires both steady-state gene expression data and knowledge of specific perturbations for each gene. The work described in [15] proposed an algorithm named 'Microarray Network Identifcation' (MNI) which also requires steady-state data, but finds perturbations automatically from the gene expression data. The other study about modeling gene regulation as system of ODEs is described in [1], which proposed the algorithm 'Time Series Network Identification' (TSNI). TSNI can cover time series gene expression data. Besides, it can discover the perturbations of the system automatically from the gene expression data and can be applied to very large datasets comprised of thousands of genes. The algorithm proposed in [14] performs multiple regression of each gene on all other genes, whereas the 'Inferelator' proposed in [6] uses L1 shrinkage method for modeling gene regulatory systems with ODEs. The al-

gorithm 'Differential Equation-based Local Dynamic Bayesian Network' (DELDBN) proposed in [40] combines differential equation modeling with Bayesian networks as the name suggests. The third generative model we used is multi-objective GA. It applies crossovers to the available gene expression samples and try to produce better candidate samples based on a multi-objective fitness function. And the forth model is the HIMM. HIMM constructs probabilistic Context Free Grammar rules for each gene, based on the string of discretized gene expression values.

The three defined metrics are *Compatibility*, *Diversity* and *Coverage* measuring different aspects of the generated data. Compatibility measures how much the newly generated samples are close with the original gene expression samples. Diversity measures how much the newly generated samples different from the original and already generated samples. Lastly, coverage measures how much the generated samples cover the sample space.

- **Compatibility:** We want the newly generated samples to be similar to the existing ones. For any new sample, we measure Euclidean distances to all samples in the original dataset. The compatibility of a newly generated sample is the mean of these calculated distance values.
- **Diversity:** We do not desire the new samples to be duplicates of the existing ones. We calculate the diversity of each new sample as the change in the total entropy of the original dataset. In order to find out the change in information content, for each i^{th} newly generated sample, we append all newly generated samples up to $i-1$ to the original dataset. Then, we calculate the entropy of each sample in the appended dataset and sum the differences of entropy values of samples. This forms a basis for the total information held by the appended dataset. For the i^{th} newly generated sample, we calculate the total information with the same procedure explained above except adding just the i^{th} newly generated sample to the previous appended dataset. By dividing the latter value of total information by the former one we get a ratio representing the contribution of each newly generated i^{th} sample to the total information held by the original dataset and the samples produced up to the $(i-1)^{th}$ sample. This value is taken as diversity metric value.
- **Coverage:** This metric is to measure how close a new sample is to the other samples created by our method. Coverage of a new sample is calculated as the sum of the Euclidean distance values to all the other new samples. If a single sample is created, the value of the coverage metric is set to the maximum of the normalization interval.

In order to select k samples among collected $4k$ samples, we have used **multi-objective selection mechanism** based on the three defined metrics. The mechanism is based of Fonseca-Flemming ranking [24]. The ranking checks vector dominance between pairs

of samples with respect to the three metrics. For each sample, there is a 1×3 vector of metric results. The ranking value of each sample is set as the number of samples whose vectors of metric results strictly dominate the vector of metric results of the selected sample. Hence, the most successful candidates are the ones that are least dominated by the other samples. Based on Fonseca-Flemming ranking, the $4k$ samples are sorted and the best k samples are chosen to output. Ties are broken randomly if necessary. Having the multi-objective selection mechanism, we generate samples having different characteristics, which improves the effectiveness of our multi-model framework.

6.3 Generative Models

This section describes the four generative models that we have integrated in our multi-model data generation framework. We can list them as following.

1. Probabilistic Boolean Network
2. Ordinary Differential Equations
3. Multi-Objective Genetic Algorithm
4. Hierarchical Markov Model

The idea behind all these models is to use the existing data for building up the models and generating samples having different characteristics. That provides to simulate gene regulation much realistically and successfully.

6.3.1 Probabilistic Boolean Network

Probabilistic Boolean Networks (PBNs) model gene regulation as a collection of Boolean Networks [60, 34]. PBN represents gene regulation as a Boolean interactions among genes. Each gene is represented as a node and each interaction is represented as a Boolean function and a wiring diagram. There are several Boolean functions associated with each gene and several wiring diagrams associated with each Boolean function. Additionally, there is a probability distribution over the associated Boolean functions and wiring diagrams for each gene. At each time $t + 1$, the values of genes are determined based on the associated Boolean functions, wiring diagrams and the probability distribution over the Boolean functions.

In order to construct a PBN, we have used the method proposed by Shmulevich et al. [60]. It uses coefficient of determination (COD) value for each possible Boolean function and its wiring diagram for each gene[60]. The COD value of a Boolean function determines the error differences between the constant Boolean function and

the actual Boolean function. For the i^{th} gene x_i, its k^{th} Boolean function f_k^i and its l^{th} wiring diagram ω_l^i, the COD value can be formulated as in Equation 6.1.

$$C_k^i = \frac{\Delta - \Delta(x_i, f_k^i(\omega_l^i))}{\Delta} \tag{6.1}$$

where Δ is the error for constant Boolean function and $\Delta(x_i, f_k^i(\omega_l^i))$ is the error for actual Boolean function. Among all possible Boolean functions and wiring diagrams we are able to find the best estimators with respect to the COD values. Moreover, by normalizing the COD values for each Boolean function and wiring diagram we can get the required probability distribution to construct the PBN.

This approach is sound for choosing the best estimators among the possible Boolean Network functions and wirings and combining these best estimators for building the PBN. However, actually it is not possible to enumerate and evaluate all possible Boolean Network functions and wirings. If there are n genes in the network, the number of possible wirings for a single gene is 2^n, since each wiring is a subset of genes. Each wiring has a cardinality k, which is the cardinality of the gene subset. Each wiring of k cardinality is associated with a Boolean function of k parameters and a Boolean value, thus there are 2^k possible functions that can be assigned to each wiring of cardinality k. If we sum up the number of all possible Boolean functions, we get:

$$2 * [\binom{n}{0}.2^0 + \binom{n}{1}.2^1 + \binom{n}{2}.2^2 + ... + \binom{n}{k}.2^k] \tag{6.2}$$

For a very simple gene expression data of 20 genes, each gene has around 1 million possible wirings and 3 billion Boolean functions should be evaluated to find the best wiring and Boolean function. So, it is mandatory to restrict the wiring and Boolean functions to a smaller set in order to be able to evaluate each and every candidate. Hence, we have set the number of wirings and the number of Boolean functions as 3 for each gene as it is suggested in [60].

We have constructed the PBN based on the available gene expression data. We specify the first [1; s - 1] samples as inputs and the last [2; s] samples as outputs of the PBN. Hence, for each sample at time t, the output of the PBN is the sample at $t + 1$. By adjusting the parameters of the PBN based on the output values obtained from the available gene expression samples, and the COD values calculated over the output values, we are able to construct the PBN that would best fit the available gene expression data.

Having constructed the PBN, the network can be easily used for generating new data by simply running the constructed PBN k times recursively. Note that we set the initial value of the PBN as the last sample in the available gene expression sample.

Moreover, we use perturbation to consider the diversity in the PBN. At each time step, there is a 0.1 of perturbation probability that the value of a gene may change independently.

6.3.2 Ordinary Differential Equations

Modeling gene regulatory systems by using ordinary differential equations (ODEs) is one of the oldest and common methods. Specifically, differentiation of each gene is associated with a regulation function of the expression levels of the other genes [31]

$$\dot{x_i} = f_i(\mathbf{x}), \tag{6.3}$$

where $1 \leq i \leq N, \mathbf{x} = [x_1, ..., x_N]'$ and N is the number of genes in the system. ODEs provide a continuous time dynamical system framework for modeling the interactions in a gene regulatory system. They are simple to use and powerful for capturing the relations between all variables.

Among different approaches capable of constructing the ODE model for gene regulations, in this work, we have used the Time Series Network Identification (TSNI) algorithm presented in [1] due to its prevailing properties on other regulation modeling algorithms. TSNI is not only able to cover time series data but also can be applied to large datasets as facilitated by the principal component analysis. Moreover, it is able to determine the external perturbations to the system automatically from the available data. After finding the regulation equations by using the TSNI algorithm, it is easy to generate new samples by just simulating the system of the ODEs.

The main task in gene regulation modeling using ODEs is to find the regulation functions for each gene. The TSNI algorithm assumes the regulation functions have the form

$$\dot{x_i}(t_k) = \sum_{j=1}^{N} a_{ij} x_j(t_k) + \sum_{l=1}^{P} b_{il} u_l(t_k) \tag{6.4}$$

where $1 \leq i \leq N, 1 \leq k \leq M$, N is the number of genes, and M is the number of samples in the existing data. Here $x_j(t_k)$ is the gene expression level of gene j at time t_k, a_{ij} is the effect of gene j on gene i, b_{il} is the effect of l^{th} external perturbation to gene i and $u_l(t_k)$ is the l^{th} external perturbation to the system at time t_k.

If we combine all differential equations in a single matrix equation, we can rewrite Equation (6.4) as

$$\dot{X}(t_k) = A * X(t_k) + B * U(t_k) \tag{6.5}$$

Here, $X(t_k)$ is the N element vector of the gene expression levels at time t_k and $\dot{X}(t_k)$ is the N element vector of the first derivative of $X(t_k)$. $U(t_k)$ is the P element perturbation vector storing the P external perturbations to the system at time t_k. A is the $N \times N$ regulation matrix and B is the $N \times P$ perturbation matrix. It is important to note that $U(t_k)$ represents the P external perturbations to the system, whereas, B represents the effects of the P perturbations to each of the N genes.

As ODEs provide continuous time modeling approach, TSNI converts Equation (6.5) into its discrete time space form

$$X(t_{k+1}) = A_d * X(t_k) + B_d * U(t_k), \tag{6.6}$$

where, A_d is the discrete time space form of regulation matrix A and B_d is the discrete time space form of perturbation matrix B. These two matrices will be converted back to the continuous time space form with the bilinear transformation in Equations (6.7) and (6.8):

$$A = \frac{2 * A_d - I}{\Delta t * A_d + I} \tag{6.7}$$

$$B = (A_d + I)^{-1} * A * B_d, \tag{6.8}$$

where I is the identity matrix and Δt is the sampling interval. In our study, we assume that sampling interval is 1.

Having the compact form of differential equations as in Equation (6.6), the only unknowns are the regulation matrix A_d and the perturbation matrix B_d. $X(t_{k+1})$ is the last $M-1$ data points, $X(t_k)$ is the first $M-1$ data points and $U(t_k)$ is the $M-1$ perturbations that are done to the system for each time t_k. Hence, it can be said that A_d and B_d are still $N \times N$ and $N \times P$ matrices, respectively, whereas $X(t_{k+1})$ is $N \times (M-1)$, $X(t_k)$ is $N \times (M-1)$ and $U(t_k)$ is $P \times (M-1)$ matrices.

By combining the unknowns in a single matrix, Equation (6.6) may further be rewritten as:

$$X(t_{k+1}) = H * Y, \tag{6.9}$$

where

$$H = \begin{bmatrix} A_d & B_d \end{bmatrix}$$

$$Y = \begin{bmatrix} X(t_k) \\ U(t_k) \end{bmatrix}$$

In order to solve the Equation (6.9) for the two unknown matrices A_d and B_d, we have to have $M \geq N + P$, which means that the number of samples should be greater than or equal to the sum of the number of genes and the number of perturbations. However, the number of genes is usually much greater than the number of samples. TSNI applies Principal Component Analysis (PCA) to overcome this problem. Matrix Y is decomposed by singular value decomposition,

$$X(t_{k+1}) = H * U * S * V' \tag{6.10}$$

This will provide to reduce the dimension of Equation (6.9) and take only k singular vectors of U or V' depending on the number of singular values in matrix S. After reduced the dimension of Equation (6.9), Equation (6.10) can be solved easily as described in [1] leading to A_d and B_d. Using Equations (6.7) and (6.8), A and B can be computed from A_d and B_d, concluding the algorithm.

There are two important points about the TSNI algorithm: the k value, i.e., the number of principal components to be considered and the P value, i.e., the number of external perturbations to the system. The number of principal components depends on the noise in the data. If the noise level is very low then 3 PCs (Principle Components) are the best, if the noise level is about 10% then 2 PCs are better and if the noise level is higher than 10% then 1 PC works best [1]. Though this parameter may be adjusted by the user of the system, in this study, we assume that the noise is about 10% and take the k value as 2. The number of external perturbations to the system may also be adjusted by the user of the system, however, in this study we set the P value to be 1 as in [1].

After finding the regulation and perturbation matrices of the differential equation model, it is easy to generate new samples from the model by just solving the system of differential equations numerically. At each time t_k, the system takes an $N \times 1$ vector of gene expression levels $X(t_k)$ as input, finds the instantaneous difference by using the regulation and perturbation matrices and gives an $N \times 1$ vector $X(t_{k+1})$ as the simulated output at time t_{k+1}. Dynamics of the system continue by taking the simulated output vector $X(t_{k+1})$ as input vector at the next time step.

6.3.3 Multi-Objective Genetic Algorithm

Genetic algorithm is a popular method which have been used for solving DNA sequence prediction, protein structure prediction or gene expression data generation [28, 53, 54, 20]. In our work, we have used multi-objective genetic algorithm method as our third generative model, which we have adapted from the genetic algorithm data generation method proposed in [20]. As its name suggests, our multi-objective genetic algorithm applies multi-objective fitness function and selection mechanism.

Multi-objective genetic algorithm applies iteratively crossovers over the available individuals and try to produce better generations in terms of their fitness. Each individual corresponds to binary value of a gene expression sample. Crossovers among the individuals are uniform crossovers [49, 66]. That is, value of each gene in the next generation is decided based only on the value of that gene in the parent generations.

Every time the crossover step is applied, two parents should be selected among the population. We have used a multi-objective selection method for this task. Three criteria we used for evaluating and selecting best samples from different models, explained in Section 6.2, are also used as fitness functions of the genetic algorithm. *Compatibility*, *diversity* and *coverage* criteria are calculated for each individual in the population to use in the selection process. Compatibility measures how much an individual is close to the other individuals, diversity measures how much an individual is different from other individuals and coverage measures how much an individual covers the sample space (See Section 6.2 for detailed explanations of the metrics). Compatibility and diversity values used in the fitness function are calculated as outlined in the sample selection process. However coverage value used in the sample selection process measures the coverage value of a set of samples. For the fitness function, we need to measure the coverage value of a single sample, thus an incremental version of the coverage calculation is used here. For measuring the coverage value of a single individual we actually measure how the coverage of the whole population changes when the individual is removed.

After calculating three criteria for all of the individuals in the population, a multi-objective selection is applied. The multi-objective selection method used is based on a ranking mechanism proposed by Fonseca and Flemming [24]. This ranking is based on vector dominance of an individual by another individual. The ranking of a given individual is the number of individuals in the population whose fitness values dominate the fitness values of the given individual. After the ranking is formed by the outlined method, the best two individuals, who are the least dominated ones in the population, are selected as mates for the crossover phase. Ties in the ranking are randomly broken if necessary.

The population is initialized as the original dataset and genetic algorithm steps are applied to individuals continuously. At each step of the genetic algorithm, there is also

0.1 probability of mutation which may change value of a gene randomly. After the k^{th} iteration, genetic algorithm produces the required k many new samples.

6.3.4 Hierarchical Markov Model

Hierarchical Markov Model (HIMM) have been firstly proposed by [71]. In his study Witten, proposed to formulate a Probabilistic Context Free Grammar to store texts. The idea is to group similar strings in the text and introduce a set of probabilistic hierarchical rules to model the strings in the text. As gene expression dataset also constitutes a set of strings, HIMM suitably fits to integrate into our multi-model framework as the forth generative model [20].

To be able to construct HIMM over the available gene expression data, we firstly discretized the gene expression samples. Then, for each gene we introduced a set of rules that are modeling the substrings of the binary values of that gene. If there are 6 samples in the dataset, for example, and 101110 is the values of the first gene in the dataset. Then HIMM introduces a rule as $X \rightarrow 10$ and represent the 101110 string as $X11X$. This is because the substring 10 is the longest common substring in the original string. In the next iteration, a new rule will be introduced as $Y \rightarrow 11$ and the string $X11X$ will be represented as XYX. This will constitute a hierarchical rule set for each gene in the dataset. Note that there may also be uncertain rules. For a string 110010, HIMM introduces a rule $X \rightarrow 1(1|0)$ and represents the string as $X00X$. Here, the symbol X has two possible values, 11 and 10, each has a probabilistic value to occur. For a probabilistic rule, we assumed the probability of producing different strings is shared equally among the all possible values for the rules. So, its probability value is 0.5 for the rule $X \rightarrow 1(1|0)$ to produce 11 or 10. We also assume the longest common substrings are matched from beginning of the string and end of the string due to reduce the exponential search time to match substrings in the original strings.

Having constructed the HIMM, the rules for each gene can be used to generate new samples. One important point is to generate fixed length of strings for each gene. This may not be possible at each step since rules are derived from fixed length of strings, which is the number of available gene expression samples in the original dataset. Hence, to generate k samples, we run the HIMM, s times such that $s \geq k$. In case, we generate more than required number of samples for a gene, we just delete some values randomly to obtain exactly k many samples. By concatenating the gene expression values produced from the HIMM for each gene, we obtain the required k samples from HIMM.

6.4 Experimental Setup

In order to quantitatively measure the performance and quality of the proposed framework, we conducted five sets of experiments with three real life biological datasets and two synthetic datasets. The following sections explain the datasets, experimental settings and evaluation semantics.

6.4.1 Datasets

The first real life dataset we used is the gene expression profile of metastatic melanoma cells [5]. This data originally contains 8067 genes and 31 samples; however, we used seven most significant genes and their expression levels [12]. We will call this dataset as *melanoma*. The second real life dataset is the previously selected set of 25 genes related to yeast cell cycle of *Saccharomyces cerevisiae* [67, 3]. Gene expression data for this set of genes is available from Spellman *et al.* [63], consisting of 6178 genes and 77 samples in total. We will call this dataset as *yeast*. The third and last real life dataset we used is siRNA disruptant dataset in human umbilical vein endothelial cells (HUVECs) [29]. It has 379 genes relevant to *Rel/NFkB* family and 400 samples. We will call this dataset as *HUVECs*.

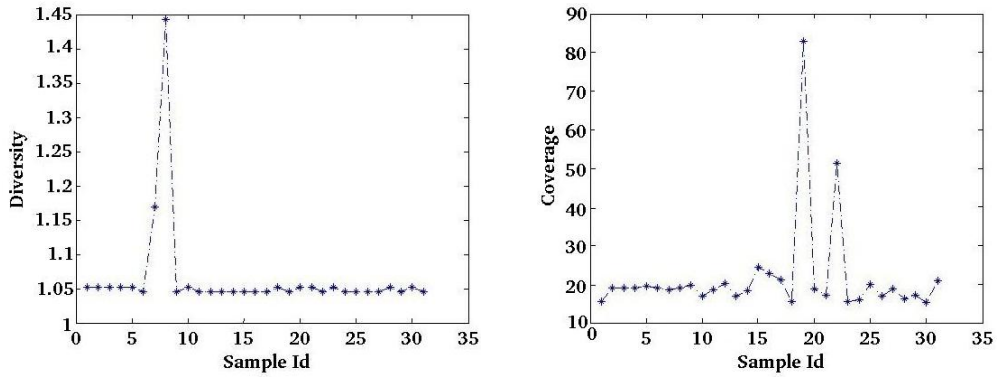

Figure 6.2: The entropy and coverage values for melanoma dataset

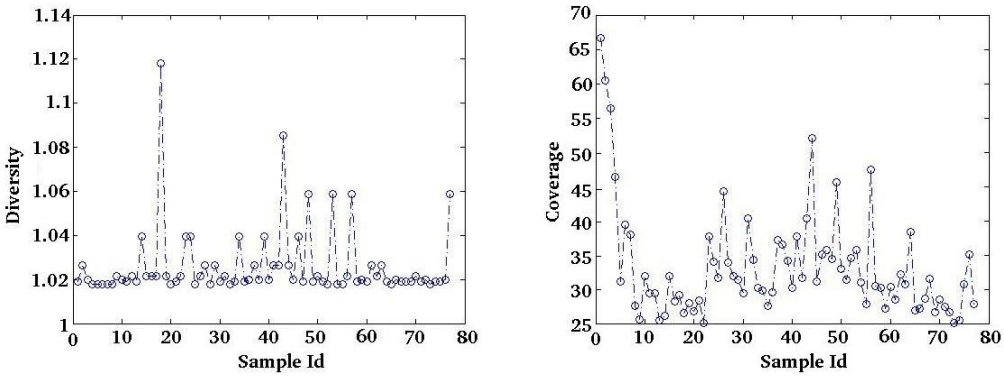

Figure 6.3: The entropy and coverage values for yeast dataset

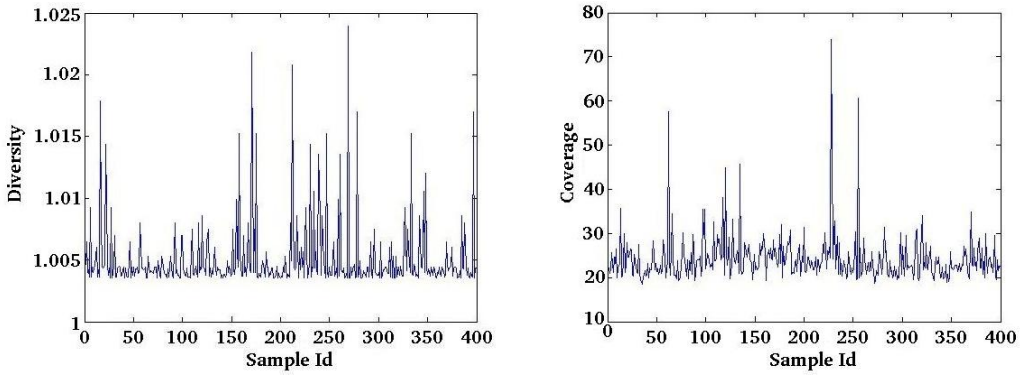

Figure 6.4: The entropy and coverage values for HUVECs dataset

In order to be able to understand the results of the experiments effectively, for each real life biological dataset, the diversity and coverage values of each set of samples with respect to their original datasets are shown in Figure 6.2, Figure 6.3 and Figure 6.4. Here, we see that the average informational contribution of each original sample to its original dataset (which is the diversity value of each sample) is 1.06 for melanoma dataset, 1.03 for yeast dataset and 1.005 for HUVECs dataset. Therefore, it can be said that the overall informational contribution of each original sample to its original dataset is about 3.2% on average. The coverage value, which represents the average distance of each sample to their original dataset, is 21.61 for melanoma dataset, 33.26 for yeast dataset and 24.12 for HUVECs dataset. Hence, the overall average distance for an original sample to its original dataset is 26.33. These diversity and coverage values will constitute the basis for results of the experiments.

We also generated two different synthetic datasets for set of experiments in Section 6.5.5. We used the tool GeneNetWeaver presented in [58] for generating the synthetic datasets. By using the tool, we extracted two subnetworks from automatically loaded *E. coli* network and simulated the subnetworks still by using the tool. The first synthetic dataset contains 25 genes and 600 samples and the second synthetic dataset contains 40 genes and 600 samples.

6.4.2 Experimental Settings

We conducted five sets of experiments to test the effectiveness of our framework. In the first set of experiments, we generated new samples by using the two real life biological datasets, melanoma and yeast, which are composed of relatively small number of samples, 30 and 77 samples, respectively. By using these two datasets we trained our multi-model data enrichment framework, produced different number of samples and showed the quality of the produced samples. The important point here is we have obtained the metric results based on all of the samples [19, 20]. That is, we trained and tested our framework based on the same dataset. Although training the models and

testing the results with respect to the same dataset lowers the confidence level of our evaluation, small number of samples does not allow to separate the data into training and testing parts (although the yeast dataset has enough number of samples to split the data, we have used it in this set of experiments to see the results for different datasets). Here, we aimed to achieve the quality of the produced samples and show the effectiveness of our framework when we have small number of samples to train our model. We showed the quality of the produced samples both by evaluating the metric results and by applying unpaired two-sample t-test to each generated dataset.

In the second set of experiments, on the other hand, we have used the yeast and HUVECs datasets, which are composed of relatively large number of samples, 77 and 400 samples, respectively. This time we divided each dataset into two disjoint parts for training and testing purposes. We built the models based on the training samples and assessed the quality of the new samples based on the testing samples. The idea is to measure the performance of our system based on the samples that the system has not seen while building the models. Thereby, we aimed to get more confident insight about the quality of the generated data by our framework and improve the confidence level of first set of experiments. We again produced different number of samples and checked the quality of the results still by evaluating the metrics and by applying unpaired two-sample t-test to each generated dataset.

We also conducted a set of experiments for justification of our multi-model data generation framework. As one of our major claims is that we can simulate the complex internal dynamics of gene regulation by combining different models together, we analyzed the contribution of each single model to our multi-model framework. We generated different number of samples by using our multi-model framework, and at each generated sample set, we identified which samples are generated from which model. Based on this identification, we estimated the contribution of each single model to our multi-model framework. Moreover, we compared the metric results for the samples that are produced by only single models and for the samples that are produced by our multi-model framework. This set of experiments provided a confident verification for effectiveness and robustness of our multi-model framework.

In the next set of experiments, we try to understand the quality of the data generated by our framework by comparing inferred GRNs from the data generated by our framework and from the original data. Based on a reference network and its original dataset, we checked the precision and recall values of the inferred network of the original dataset and the inferred network of the generated dataset. We aimed to show that improving the quality of the original dataset with our framework leads to infer better regulatory networks which may reveal some new biological discoveries.

Lastly, we conducted experiments on the number of required samples for training our multi-model framework. Empirically, we tried to assess the required number of training samples by generating datasets with increasing number of training samples

and checking the quality of each generated dataset with respect to the previous one. We expect to see that after some point the increase in the number of training samples will not improve the quality of the data generated by our multi-model framework. We aimed to get a hypothetic number for the samples to be generated in the laboratory environment so that the biologists can enlarge their datasets confidently. That will not only be a guidance on how many real samples will be enough to generate in laboratory environment, but also will bound the cost of the experiments on gene expression data generation.

6.4.3 Evaluation Semantics

The details of the evaluation criteria are discussed in Section 6.2. As compatibility and coverage metrics are distance based metrics, their results are mapped into $0-100$ interval so that higher compatibility and coverage means newly generated samples are relatively more similar to the original dataset and comprehensive in its solution space, respectively. The diversity measure, on the other hand, represents the ratio of change in the total entropy of the original dataset for each newly generated sample. So, it remains as it is. One important point about diversity is that the expected diversity value for each newly generated sample is a value greater than 1.0 because any value greater than 1.0 means to carry new information with respect to the available samples (See Section 6.2 for detailed explanation of diversity). For evaluation semantics, it can be said that higher values for coverage is desired as the increase in covered sample space improves the quality of the generated data. For compatibility and diversity metrics, it can be said that there is a balance between them. As an optimum result we want to have high values of compatibility and high values of diversity, which implies that we do not duplicate the original data, and have very close samples still carrying new information. If we have lower values of compatibility, on the other hand, lower values of diversity is desired. Because, if the data is not close to the original data we want it to be similar to the original one. Although this case is not an optimum result, it may reasonably be acceptable. For the other cases, however, higher values of compatibility and lower values of diversity, and lower values of compatibility and higher values of diversity implies either duplication or irrelevance of original data, respectively. After evaluating the newly generated samples by the defined metrics, we applied the multi-objective selection mechanism for determining the valuable samples.

Our multi-model framework has a highly probabilistic nature. Except the ODE model, all the other three models have probabilistic results. Hence, each step of each experiment is repeated 10 times to decrease the effects of the probabilistic nature of our framework. The reported metrics are the average values over 10 repetitions.

6.5 Experimental Evaluation

6.5.1 Experiments on Sample Quality Using Small Number of Samples

First, we want to analyze how our proposed framework performs when there is small number of available samples. We have used both melanoma and yeast datasets to be able to see the difference between the results and check the dependency of the performance of our system to different datasets. Here, the performance is evaluated based on the complete original dataset. We have run our system 50 times, produced $10, 20, ..., 500$ new samples and checked the results of the metrics. The compatibility, diversity and coverage results are shown in Figure 6.5.

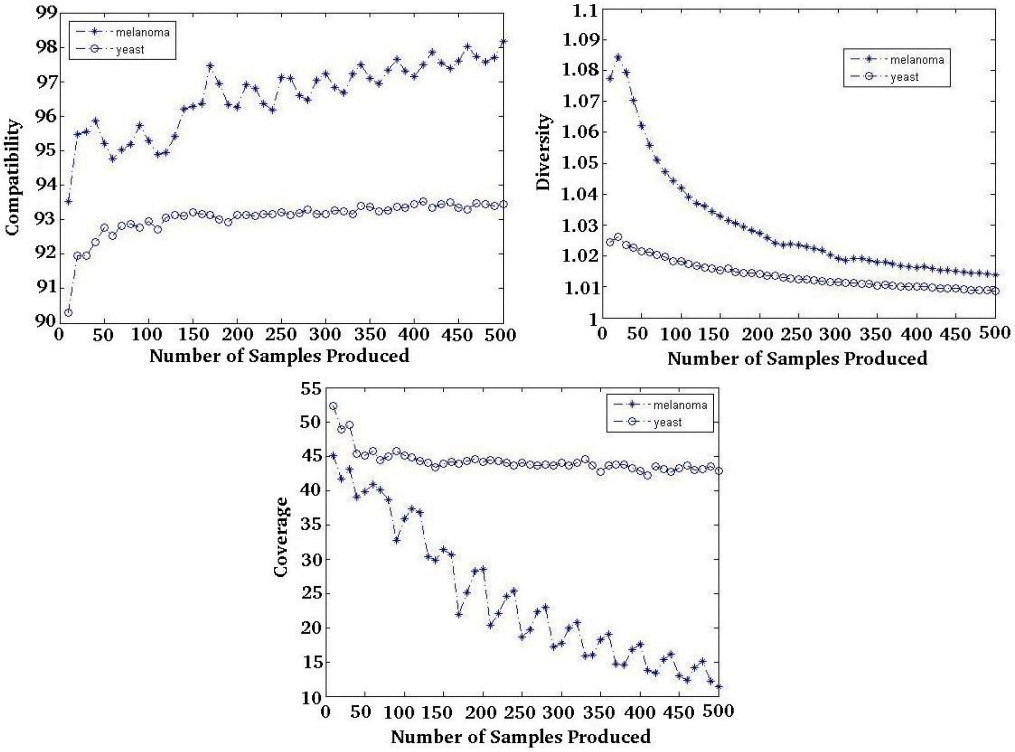

Figure 6.5: Compatibility, diversity and coverage results for different number of samples produced

For both datasets, compatibility values increase as the number of generated samples increases. It is mainly because of the fact that system produces more similar samples to the original dataset as the number of produced samples increases. We can say that generated samples converge to the original dataset in terms of compatibility. This proves that our system does not produce irrelevant but highly compatible samples with respect to the original dataset. Moreover, if we compare the compatibility values with respect to the coverage values of the original samples, as examined in Section 6.4.1, we can say that our generated samples are much closer to the original samples than the original samples are close to each other. The average coverage value of the original

samples is 21.61 for the melanoma dataset, and 33.26 for the yeast dataset. However, here we have always compatibility values greater than 90.0 which is even more than twice of the original coverage values. That means, in terms of compatibility metric, our system achieved very good results. One critical point here is that very much high compatibility values may lead to duplication of the original samples, which we consider while evaluating the diversity values of the resultant samples.

Diversity values decrease as the number of produced samples increase, which is an expected result since we are calculating the diversity values in a cumulative way. For the i^{th} newly generated sample, we calculated the entropy value with respect to the samples composed of the original dataset plus all the generated samples up to the $(i-1)^{th}$ newly generated sample. There are several important points here. First, all newly generated samples, even for 500 produced samples, have diversity value greater than 1.0. This proves that our multi-model system produces samples always carrying some new information with respect to the original dataset. If we combine this result with the compatibility results, we can safely say that our system generates valuable samples since they are both close to the original ones and always carry new information. The other point is the comparison of diversity values of the newly generated samples with respect to the diversity values of the original samples explained in Section 6.4.1. For the melanoma dataset, the average diversity value for its original samples is 1.06. In the generated datasets, on the other hand, interestingly greater diversity value for the generated dataset is noticed up to 100 samples. Each original sample brings 6% more information to the original dataset, whereas, our newly generated samples always bring information greater than %6 to the original dataset up to generating 100 samples. That means our system produces not only very close samples to the original melanoma dataset as their compatibility values are so high, but also produces samples carrying even more information than the original ones. Since we only have 31 samples in the original dataset of melanoma, confidently generating 100 new samples is fortunately a very good result. For generating more than 100 samples, we still have a reasonable diversity value, which is 1.03 on average and these samples also may be used to enrich the dataset although not having as much confidence as for the sample sets up to 100 samples. For the yeast dataset, the results are fairly well. Up to generating 100 samples, we have 1.02 diversity value on the average and for rest of the produced samples we have 1.014 average diversity value. The average diversity value for the original yeast dataset is 1.03. Therefore, the results for the yeast dataset is not as good as the results for the melanoma dataset. However, they are still acceptable since their diversity values are always greater than 1.0. Hence, we can say that our framework generates samples that are very close to the original ones and always carry reasonably good amount of new information with respect to the original samples.

The coverage values decrease as the number of produced sample increases for the samples generated from the melanoma dataset. We can observe a similar situation for the yeast dataset case but having proportionally greater values. This is consistent

with the compatibility and diversity values meaning that the generated data is getting closer to each other. As we produce more data, the possibility of generating a sample similar to one of the original samples increases. Hence, after some point, our generated samples started to resemble each other, lowering the coverage value. The impressive result here is again the comparison between the coverage values of the original samples explained in Section 6.4.1 and the coverage values of the newly generated samples. For the melanoma dataset, the average coverage value for the original samples is 21.64. The coverage values for the generated samples, on the other hand, is greater than this value up to generating 200 samples. This not only supports the claim that our framework generates successful results mostly up to generating 100 new samples, but also improves it up to 200 samples for the melanoma dataset. Since our framework applies a multi-objective selection mechanism, we can say that coverage values compensate the diversity values between 100 and 200 values. Therefore, up to generating 200 samples, it can be said that the generated samples have high quality. For yeast dataset, we have even better results in terms of coverage. The average coverage value for the original samples is 33.26 in Section 6.4.1. In the generated datasets, however, we never have a coverage value less than 43.0. This shows that we always have greater coverage values for the samples generated by our framework from the yeast dataset. Although the diversity values of the generated samples from the yeast dataset is less than the diversity values of the original samples, the multi-objective selection mechanism always compensate the diversity value in terms of coverage value. Therefore, we can again say that the samples generated from the yeast dataset also have high quality.

In order to improve our estimation on the samples generated by our framework, we also applied unpaired two-sample t-test to the generated datasets. We compared the generated datasets, $10, 20, ..., 500$, and the original datasets with the null hypothesis that they have normal distribution, equal means and variances. We rejected the null hypothesis at the 5% significance level and checked the results of the p-values of the t-test for each generated dataset. T-test is applied to each gene separately. That is, each gene's distribution is compared in the two datasets, which are the generated dataset by our framework and the original dataset; then the mean of the p-values of the all of the genes are plotted. Since our significance level (the α-value) is 0.05 we reject the null hypothesis for p-values less than 0.05. As we do not want that our generated samples are significantly different than the original samples, we want the p-value results to be greater than 0.05. Figure 6.6 shows the p-value results. For the melanoma dataset, we only get close to 0.05 p-value for around 130 and 210 sample generation experiments. Other than these, we always have a p-value higher than 0.05. Even, for experiments generating 250 and more samples, we get higher p-values, meaning that our generated samples poses increasingly similar distribution to the original melanoma dataset. For the yeast dataset, on the other hand, we always have p-values between 0.3 and 0.4, meaning that we never get close to p-value of 0.05 but have reasonably high p-values. Hence, once again we can say that the generated datasets and the original datasets

always have highly similar distributions.

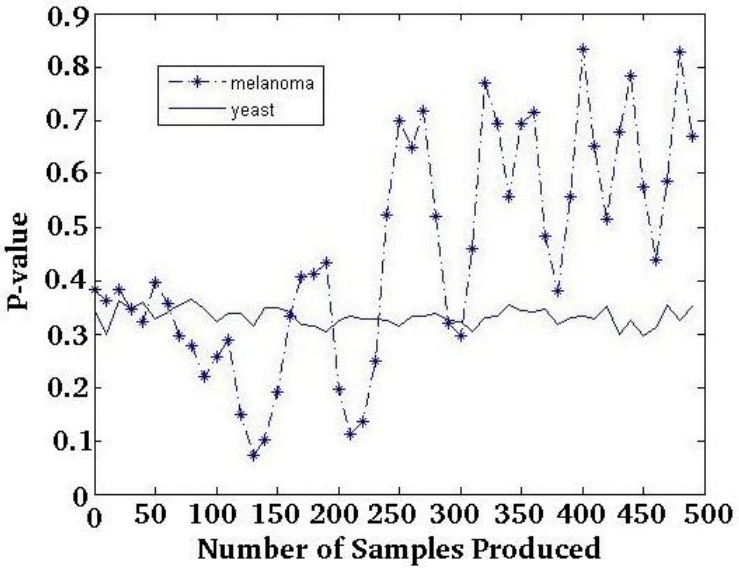

Figure 6.6: T-test results for different number of samples produced

Having very high compatibility results, still carrying new information, spanning sample space better than the original samples and resulting in high p-values of the t-tests, we can say that our framework produces highly valuable samples. Although some metric results drop at some points in the experiments, we observe that our system compensates the decrease in one metric with an increase in another metric multi-objectively, thereby producing high quality of samples. To give an approximate value for the number of samples that can be generated confidently by our framework for melanoma and yeast datasets, we can say that it is safe to generate 200 samples from melanoma dataset since at least two of the three metrics pose better results than the original samples. On the other hand, it is safe to generate samples up to 500 from yeast dataset since there are always at least two metrics having always greater values than the original samples have.

One important point here is that all of the results of the experiments in this section are based on original sample sets from which the generative models were already built. Hence, it can be said that the training and test sets are same in this set of experiments, which lowers the confidence level of the results, indeed. Since all of used generative models adapt a general estimation rather than learning a supervised dataset, we can actually say that the results in this section have a reasonable confidence level and our framework fits well for datasets having small number of samples. However, we believe that our system should also be trained and tested based on different datasets. This is because, superposing different models in such a big multi-model framework may lead to *over learn* the internal dynamics of gene regulation, and reflect somehow skewed or misleading results when tested with the samples from which the models are already built. Therefore, in the following section, we examine our experimental results based

not only on the training set, but also on a separated test set that our system has not seen during the model building phase. That will certainly improve the confidence level of our experiments and will allow us to evaluate the results much more reliably.

6.5.2 Experiments on Sample Quality Using Large Number of Samples

In the previous section, we have evaluated our defined metrics with respect to the original datasets. The aim was to measure how much valuable the newly generated samples are relative to the original samples. Although this comparison gives fair results and represents the quality of the generated data acceptably, we cannot be sure about the results since the original datasets are used not only for *training*, but also for *testing* purposes.

In this section, thereby, we have divided yeast and HUVECs datasets into two parts, training and test sets. The reason for choosing these datasets is that they contain relatively large number of samples. For yeast dataset we have used the first 50 samples as training set and the last 27 samples as test set. For HUVECs dataset we have used the first 300 samples as training set and the last 100 samples as test set. We have used training sets for building the models, and the test sets for evaluating and selecting the best k samples from the generated $4k$ samples from the 4 different generative models. We have calculated metric results relative to the training set as well to be able to see the difference. As in previous section, we have run our system 50 times to produce $10, 20, ..., 500$ samples and checked the results of the metrics.

In Figure 6.7, the compatibility, diversity and coverage values are shown for the generated samples from yeast dataset with respect to both training and test sets. From the plots, it can clearly be seen that our generated samples are closer to training set than to test set since the compatibility values are less with respect to the test set than with respect to the training set. The diversity values of the generated samples, on the other hand, are larger with respect to the test set than with respect to the training set. These two results validate our intuition about the misleading evaluations and skewed metric results when we test the generated samples with respect to the training set itself. This is because, the generated samples seemed to be more close to the original sample set than actually they are and less diverse from the original sample set than actually they are. Moreover, it misleads us to estimate the results as going to be duplication of original samples. However, the actual results show that while we still have reasonably high values for the compatibility metric, we have much better diversity values. If we compare the compatibility and diversity values with those values of the original samples explained in Section 6.4.1, the importance of the results will be more apparent. The compatibility values are still much higher than the coverage values of the original samples. The average coverage value of the original samples was 33.26 for yeast dataset, however, here we have always compatibility values greater than 85.0 which is even more than twice the original coverage values. Again, it means that our

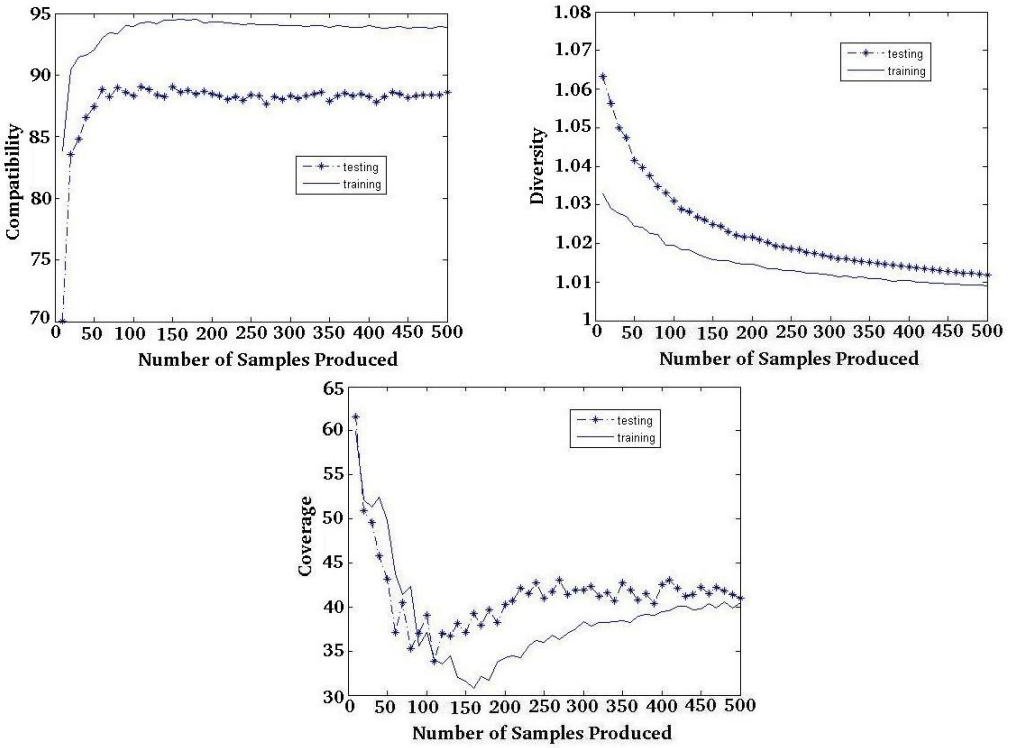

Figure 6.7: Compatibility, diversity and coverage values based on training and test sets for each generated sample set from yeast dataset

generated samples are much closer to the original samples than the original samples are close to each other.

The average diversity value for the original samples was 1.03 for the yeast dataset. Here, it can be seen that the diversity values are always less than 1.03 with respect to the training set for the generated samples from the yeast dataset. However, we can see that the diversity values are greater than 1.03 up to generating 100 samples and it is about 1.02 up to generating 300 samples with respect to the test set. Therefore, we can claim that our results are even better than we think they are since their informational content shows itself better when it is evaluated with respect to an unseen dataset, the test set. For yeast dataset, up to 100 samples, the generated datasets are very close to the original samples and carry more information than the original samples carry.

The coverage values for the generated samples are very similar with respect to the training and test sets up to generating 100 samples. After 100 samples, we generated better samples based on the test set than based on the training set in terms of coverage. Note that, again the coverage values are always better than the coverage values for the original sample set explained in Section 6.4.1. This means that we always cover the sample space better than the original dataset. Moreover, this result shows that our multi-objective selection mechanism works effectively since it immediately compensate the diversity value with coverage value after generating 100 samples. Because, after

100 samples the diversity values decrease below the 1.03 value, the diversity value of the original samples, and the system compensates this decrease by generating samples having better coverage. Therefore, for yeast dataset we confidently claim that our system produces very good samples especially up to generating 100 samples since all three metrics are better than the metric results for the original samples explained in Section 6.4.1. After that point, we still produce good samples, since the compatibility and coverage values are much better than the values of the original ones, and the diversity value is always greater than 1.0, which means that it always brings new information to the available dataset.

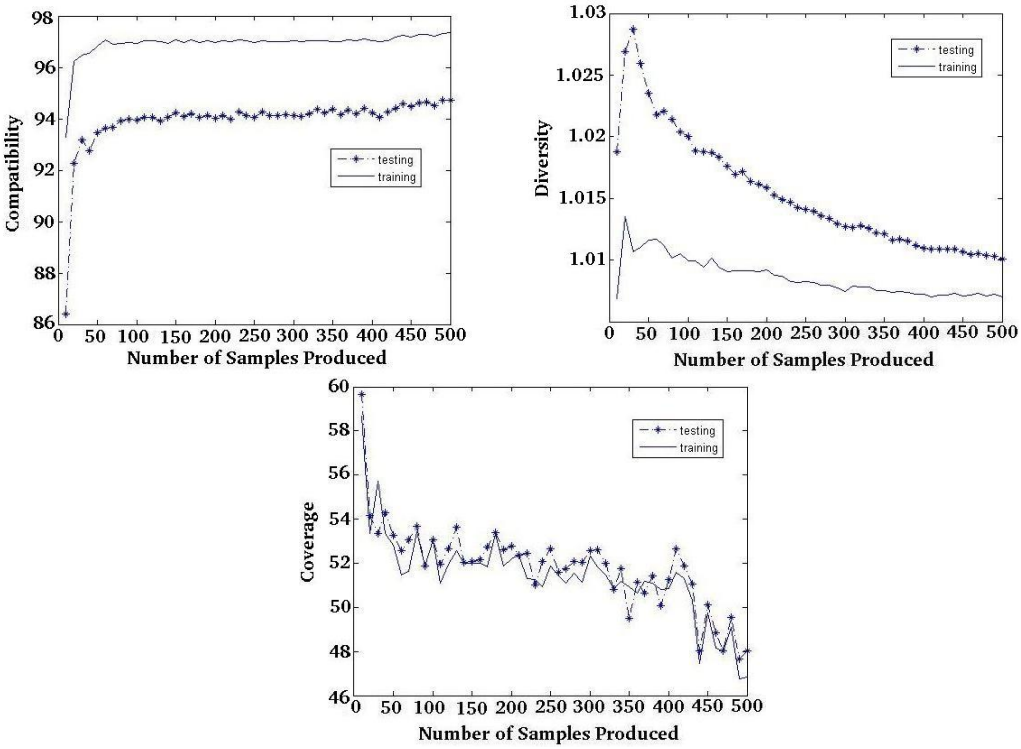

Figure 6.8: Compatibility, diversity and coverage values based on training and test sets for each generated sample from HUVECs

The results for the HUVECs dataset are shown in Figure 6.8. They are similar to that of yeast dataset. The compatibility results are less with respect to the test set compared to the training set, and the diversity values are greater with respect to the test set compared to the training set. Again, we can say that the evaluation based on the training set misleads us posing skewed results, and the evaluation based on the test set shows that our samples are actually better than what we think they are. The average coverage value for its original samples in the HUVECs dataset was found as 24.12 in Section 6.4.1. From the plot, we can see that the average compatibility value is about 94% which is more than four times of the value of the original samples. Therefore, again, this means that our generated samples are much closer to the original samples than the original samples are close to each other.

Moreover, the diversity values of the newly generated samples present very good results. Figure 6.8 shows that the diversity values for the generated samples are *always* much better than 1.005, which is the average diversity value of the original samples shown in Section 6.4.1. Even, it never gets below to 1.01, and up to generating 200 samples, it is 1.02 which is about four times the original diversity value. Note that, when we calculate the diversity values with respect to the training set, we again see that the skewed results mislead us since the values on the plot converge to a value below 1.01 whereas the values are almost always greater than 1.01 with respect to the test set.

The coverage values on the other hand, are almost greater than twice of the average coverage values of the original samples calculated in Section 6.4.1, 24.12. This means that in this set of experiments we always cover the sample space better than what the original samples cover.

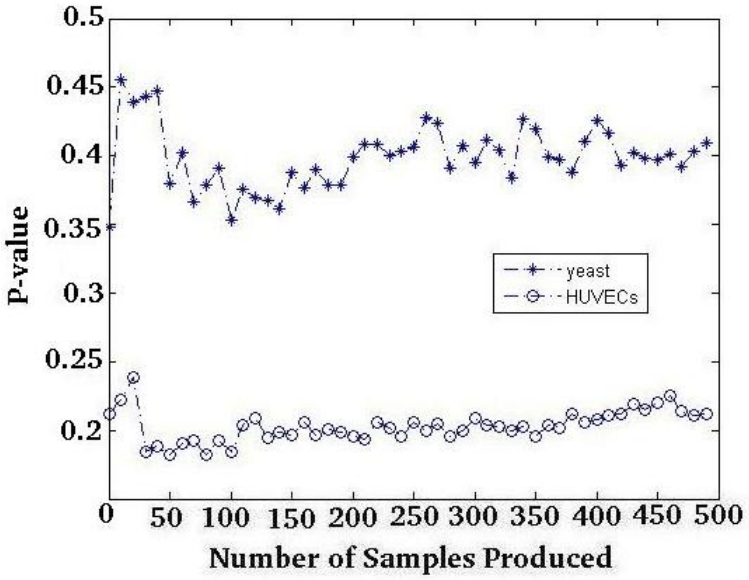

Figure 6.9: T-test results for different number of samples produced

In order to improve our estimation on the samples generated by our framework, we again applied unpaired two-sample t-test to the generated samples, $10, 20, ..., 500$, from the yeast and HUVECs datasets. The null hypothesis is again that the generated samples have normal distribution, equal means and variances. The t-test is again applied to each gene separately as explained in Section 6.5.1. Note that, unlike it is done in Section 6.5.1, this time we used the separated test set in the t-tests, instead of the whole original dataset. We reject the null hypothesis at the 5% significance level and check the results of the p-values of the t-test for each generated dataset. As before, we reject the null hypothesis for the p-values less then 0.05. Since we do not want that our generated samples are significantly different than the original samples, we want the p-value results to be greater than 0.05. Figure 6.9 shows the p-value results. For yeast dataset, p-values are always between 0.3 and 0.4 just like in Figure 6.6. This

is expected because we use the similar dataset for generating the models, instead of using all of the 77 samples in yeast dataset, we used only 50-sample training set. Since we always have higher values than 0.05 for the p-values, our generated samples always have highly similar distribution with the original dataset. For the HUVECs dataset, on the other hand, the results never get close to 0.05 but flows between 0.18 and 0.25. Although these results are not as high as the previous t-test results, we can still say that our generated samples have similar distributions with the original dataset as we never get below 0.05. The relatively lower p-values can be related to the number of genes in the HUVECs dataset since we are applying the t-test to each gene separately. There are 379 genes, and it is hard to perfectly maintain all of the genes' distributions for our system. Still, we have high results of p-values implying the required sample distribution is achieved on the average.

To give an approximate value for the number of samples that can be generated confidently by our framework for yeast and HUVECs datasets we can say it is always safe to generate samples up to 500 from the yeast dataset since there are always at least two metric results for the generated sample sets having better values than the original samples as in the case in Section 6.5.1. For the HUVECs dataset, it is even safer to generate samples up to 500 since all the three metric results of the generated sample sets have *always* better values than the original samples.

The metric results for the HUVECs dataset are the best we achieved up to now. First of all, we separated the training and test sets. Therefore, we have high confidence level in the results of the experiments. The compatibility value is about 94%, which is a very high result. The diversity and coverage results, on the other hand, are *always* greater than the diversity and coverage values of the original ones. First time in the experiments, we do not need to use the multi-objective mechanism, and get always better results than the original samples. It is a very impressive result, indeed. Because it can be stated that computationally we are able to generate many new samples which are highly qualified with respect to the original samples. The complex internal dynamics of the gene regulation can be simulated successfully by superposing different models and generating data as if it were generated by the complex internal dynamics. This does not only show the power of the computational methods but also provide practically qualified gene expression data, which can be used for many other purposes.

We believe that the main reason for being able to generate more valuable results with HUVECs dataset is the number of samples we used while building the models. We have used 300 samples from HUVECs dataset for building the generative models, while it was 50 for yeast and 31 for melanoma dataset. Therefore, if there is enough data to train our multi-model framework, which actually will be examined in Section 6.5.5 in detail, it can be said that integrating machine learning techniques can produce successful results for generating new samples without even trying to deal with the details of different models and the complexity of the internal dynamics of the gene

regulation. Instead of doing real experiments with very high costs, combining different generative models from different domains and producing artificial new samples with our multi-model gene expression data generation framework is a very effective way of enriching the available, and generally low number of, gene expression samples.

6.5.3 Experiments on Multi-Model Justification

In this section, we try to show the effectiveness of our multi-model approach experimentally. In order to do this, we generated $10, 20, ..., 500$ samples by our framework, and checked the contribution of each model to the resultant sample set. To illustrate, for a generated 50 samples, we checked the number of samples produced by PBN, HIMM, genetic algorithm, and ODE separately. Then, we map those numbers into $0 - 100$ interval and get a percentage value of the contributions. Figure 6.10, 6.11, and 6.12 show the results of the contributions with respect to the number of generated samples from melanoma, yeast and HUVECs datasets, respectively.

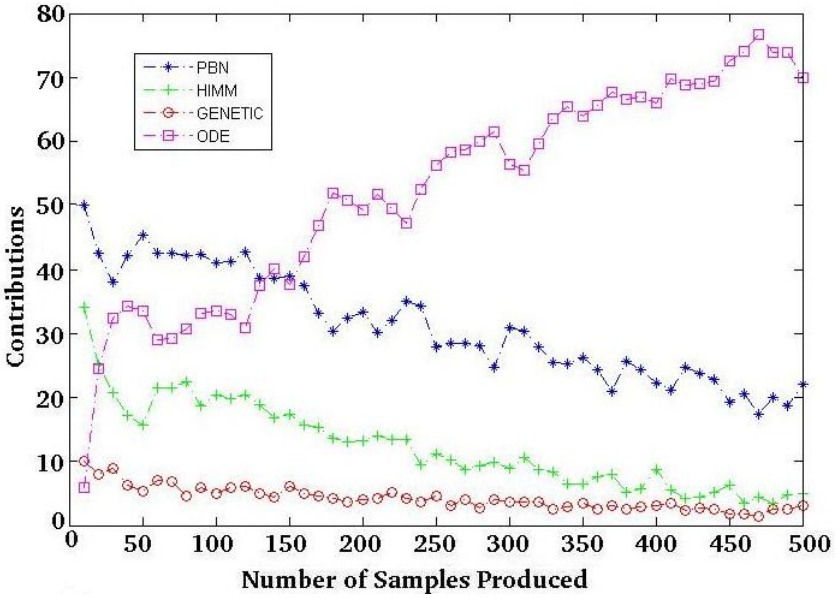

Figure 6.10: Contribution of each model for the samples generated from melanoma dataset

Figure 6.10 shows the contribution percentages for the samples generated from melanoma dataset. Up to generating 150 samples, it can be seen that the PBN generally produces %50 of the samples, and the rest is partitioned between ODE, HIMM and genetic algorithm in proportion %30, %15, and %5 respectively. However, after 150 samples, ODE contributes to the resultant samples with increasing percentages reaching to $\%70 - \%80$ at the end. PBN mostly holds the remaining parts of the generated samples, since HIMM and genetic algorithm goes down to $\%2 - \%3$ contributions. We can say that ODE and PBN govern the multi-model framework for melanoma dataset as the number of generated samples increases.

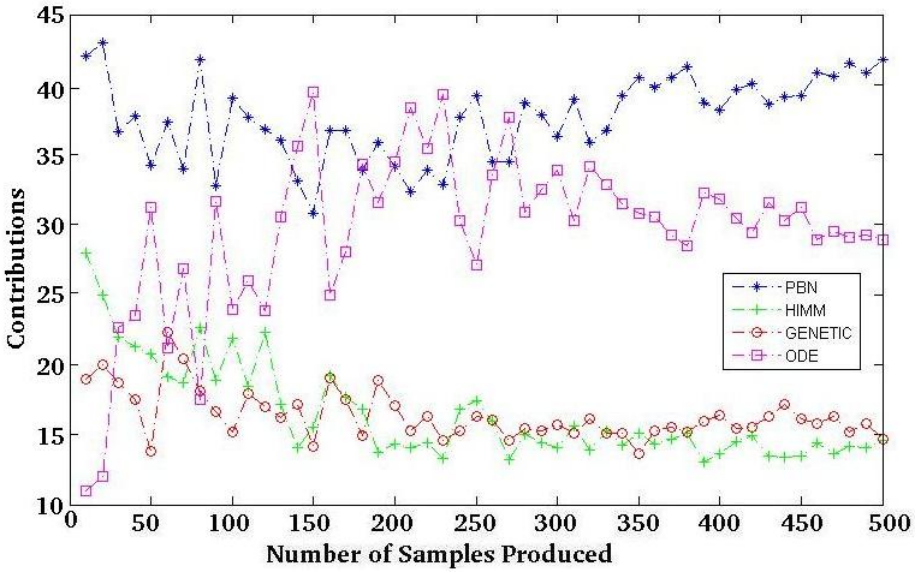

Figure 6.11: Contribution of each model for the samples generated from yeast dataset

In Figure 6.11, the contribution percentages for the samples generated from yeast dataset is shown. Here, we see that PBN almost always contributes most among the generative models. However, here, PBN and ODE do not have much difference as in melanoma case, but fairly share contributions. PBN has contribution percentage between $\%35 - 40$, whereas ODE has contribution percentage of $\%30$ on the average. Both HIMM and genetic algorithm contribute $\%15$ on the average. We can again say that PBN and ODE govern the multi-model framework, however the difference between the contribution proportions of the generative models is relatively small with respect to the melanoma dataset.

Figure 6.12, lastly, shows the contribution percentages for the samples generated from HUVECs dataset. Interestingly, in this plot, the HIMM increases its contribution significantly. PBN still contributes almost $\%50$ of the samples, however, unlike the previous contribution results, here, HIMM governs the two other models, ODE and genetic algorithm. Moreover, most of the time, ODE seems the worst model contributing to the system although it was reported among the two best models in Figure 6.10 and 6.11. ODE and genetic algorithm have contribution percentages as $\%10$. This result shows that our multi-model framework actually works very effectively since it chooses samples from different models with respect to the quality of the samples that the models generated. If there would not be such a multi-model framework, instead only single model for example, the system would have always to choose the samples from that single model, which may lead to produce highly low quality samples. For example, for the samples generated from the HUVECs dataset, if we used only ODE, which is a popular model in the literature, it would be terrible since we would produce almost one-tenth quality of the samples that we are able to generate now with our multi-model gene expression data generation framework. Therefore, we can confidently

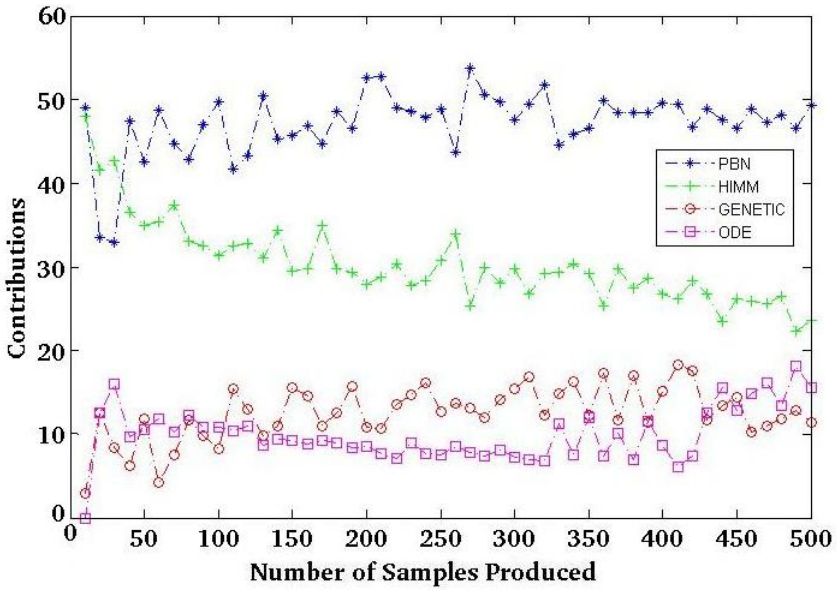

Figure 6.12: Contribution of each model for the samples generated from HUVECs dataset

say that our multi-model framework is so robust that, even if one of the models in our framework produces low quality samples due to some characteristics of the datasets, other models provide to generate samples that have always high quality, which can also be observed by the experiments in Section 6.5.1 and Section 6.5.2.

In order to show the robustness of our system, we also show the metric results for the samples that are generated from each single model of our multi-model framework, and the metric results for the samples that are generated from our multi-model framework. Here, we have used HUVECs dataset for training and testing our framework. We have generated $10, 20, ..., 500$ samples for checking the results. Figure 6.13 shows the compatibility, diversity and coverage values for each single model and for our multi-model framework. As can be seen from the plot, by combining valuable samples multi-objectively from different models our system always produces samples having better quality than a single model can produce. If we would choose samples only from HIMM whose coverage values are better than our multi-model framework, for example, our compatibility and diversity values would be less than the ones generated by our multi-model framework. If we choose samples only from ODE, whose diversity values are better than our multi-model framework, the compatibility and coverage values would be much less than the ones generated by our multi-model framework. Therefore, we further support our claim that our multi-model approach effectively combines qualified samples from different models so that the resultant samples always have higher quality than each single model since the results of the multi-model framework, multi-objectively, are always better than the results of each single model.

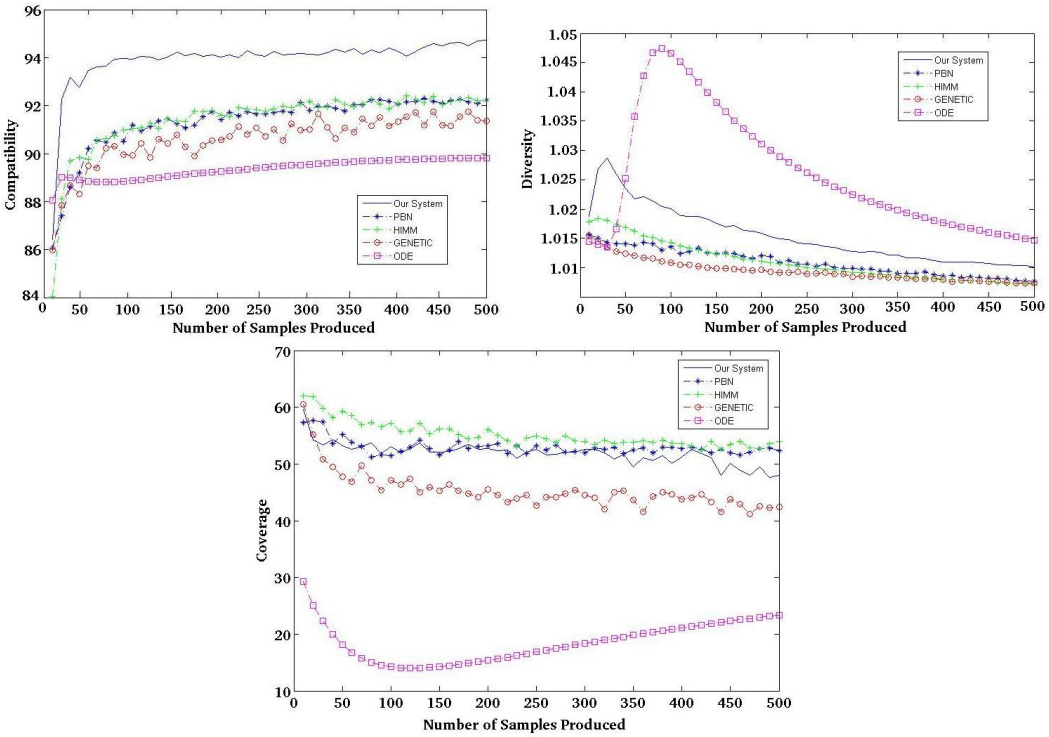

Figure 6.13: Compatibility, diversity and coverage values for each single model and for our multi-model framework

6.5.4 Gene Regulatory Network Inference

In this set of experiments, we aimed to show the effectiveness of our multi-model data generation framework through showing the quality of the inferred GRNs from the samples generated by our framework. The idea is to improve the quality of the inferred GRN by *improving the quality of the dataset* that the network is inferred from, instead of *improving the algorithm* that infers the network. Since many gene regulatory inference algorithms try to improve the algorithm to infer better regulatory networks, e.g., [44, 70, 27, 36, 73], here we propose a new way of improving quality of the inferred regulatory networks.

We have used 14-genes reference sub-network of the yeast cell-cycle pathway obtained from KEGG [33] presented in [72]. We extracted the original dataset of the 14 genes from the complete dataset of the yeast cell-cycle [63]. It contains 77 samples. By using this 77-sample-original dataset, we generated 77 new samples with our multi-model framework. Then, we inferred two regulatory networks by using the original 77 samples and the 77 samples generated by our framework and evaluated them with respect to the reference network we have. The reference network is shown in Figure 6.14.

The evaluation of the inferred network is done by using the precision, recall and f-measure metrics, which are formulated as below.

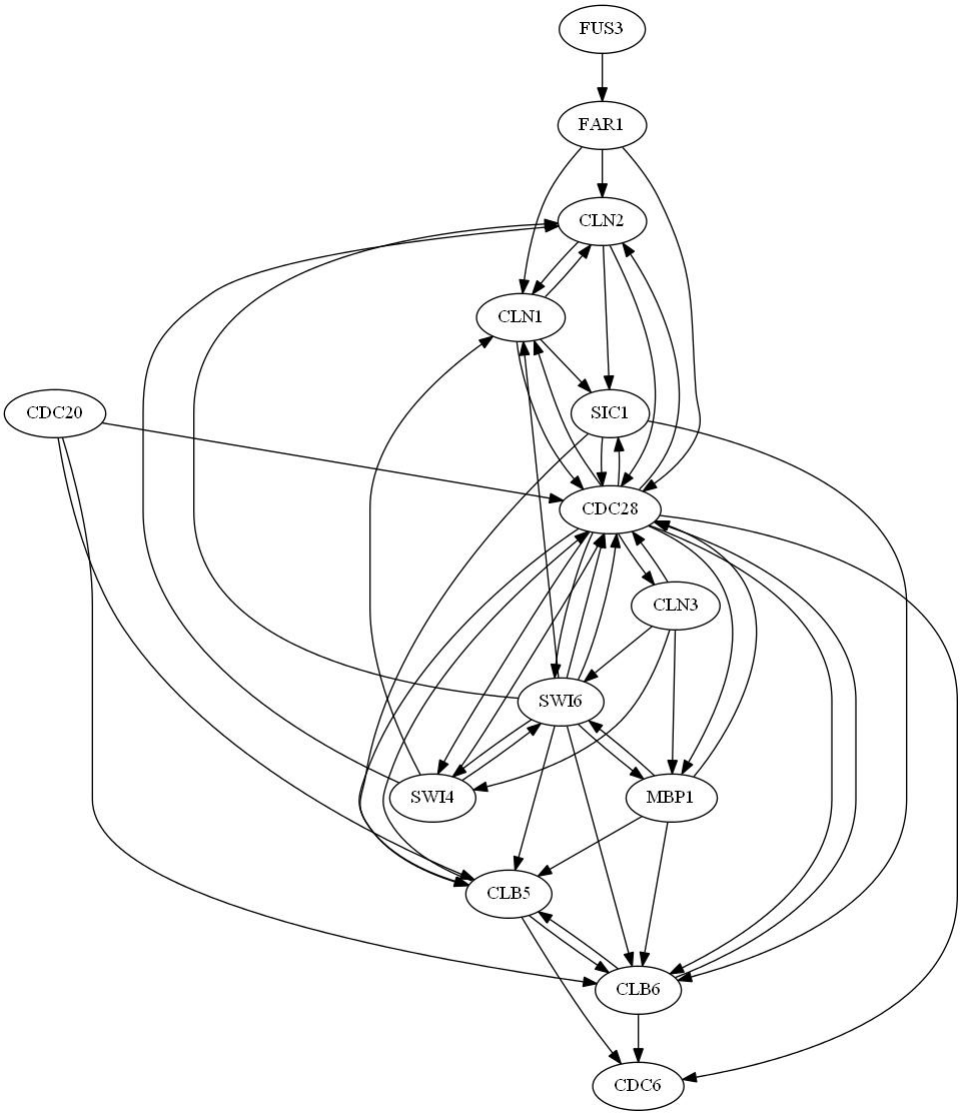

Figure 6.14: The reference subnetwork for yeast cell cycle

$$precision = \frac{\#\_of\_correctly\_predicted\_edges}{\#\_of\_predicted\_edges} \tag{6.11}$$

$$recall = \frac{\#\_of\_correctly\_predicted\_edges}{\#\_of\_edges\_in\_the\_reference\_network} \tag{6.12}$$

$$f - measure = 2 * \frac{precision * recall}{precision + recall} \tag{6.13}$$

We have used two different network inference algorithms for obtaining the networks. The first algorithm is BANJO presented in [73], which obtains a dynamic Bayesian network from the gene expression data. The second algorithm is ARACNE presented in [44], which obtains a GRN based on mutual information. Note that the two algorithms are designed to work on different types of data in the ideal case. BANJO tries to capture the system dynamics by using time series gene expression data, whereas,

Dataset	Precision	Recall	F-measure
Original	0.3846	0.0962	0.1538
Artificial	0.4211	0.1538	0.2253

Table6.1: Network evaluations for BANJO algorithm

ARACNE obtains a network by using steady-state gene expression data. We have used different types of algorithms to be able to see the effectiveness of our framework more clearly.

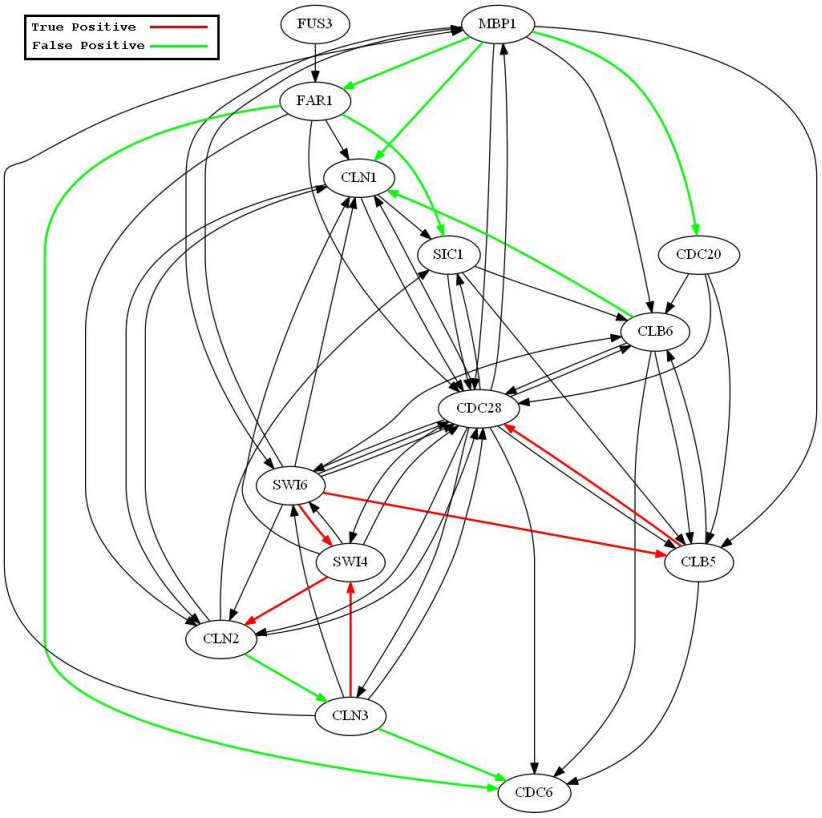

Figure 6.15: The regulatory network obtained from original dataset using BANJO

The networks obtained from the BANJO algorithm using the original dataset and using the samples generated by our framework are shown in Figure 6.15 and 6.16, respectively. The results on precision, recall and f-measure metrics are shown in Table 6.1. As can be seen from Figures 6.15 and 6.16, the network inferred from the original dataset has predicted 5 correct edges, whereas the network inferred from the samples generated by our framework predicted 8 correct edges. This is an important improvement in the regulatory network. Moreover, the three metric results, precision, recall and f-measure values, are all greater for the network obtained from the generated dataset compared to the network obtained from the original dataset. Having better precision value means that false positive edge ratio is also better for the network inferred from the generated data. Having better recall value means that the network inferred from the generated data covers the reference subnetwork better. Hence, we

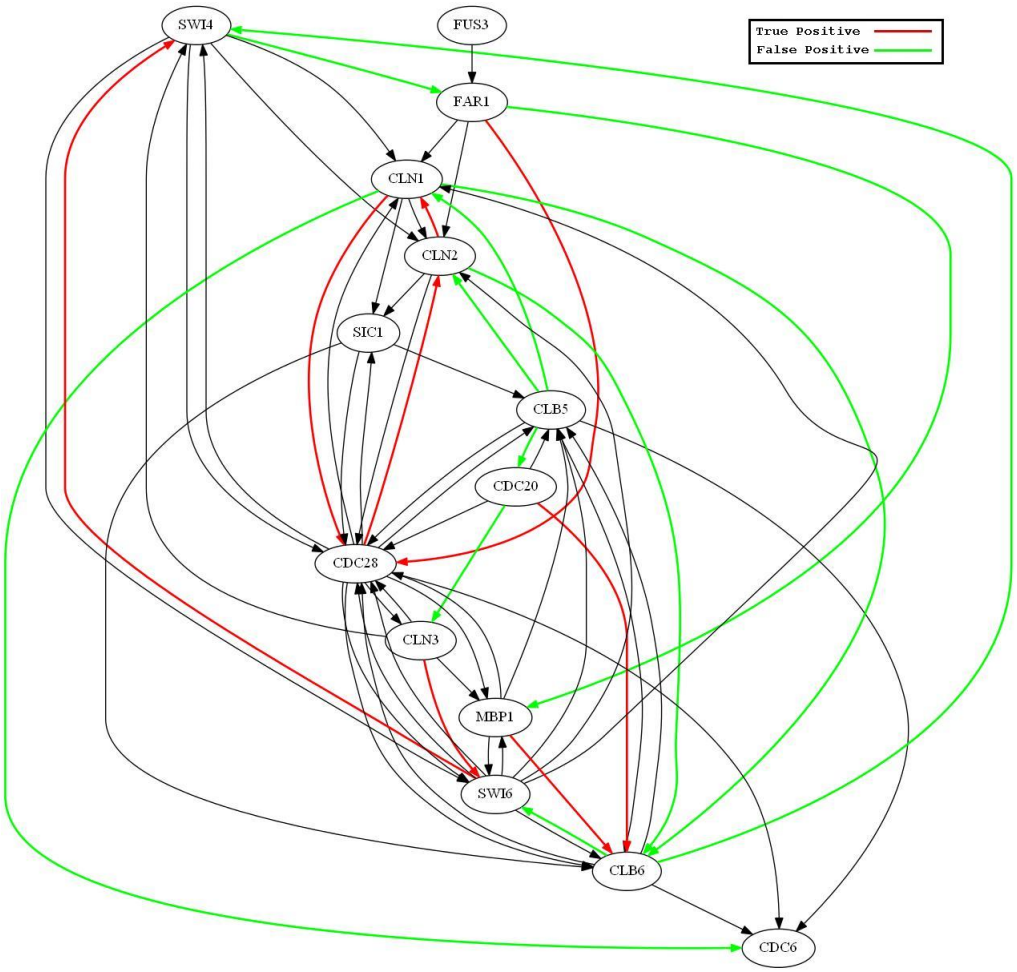

Figure 6.16: The regulatory network obtained from samples generated by our framework using BANJO

can safely say that the network inferred from the generated dataset is better than the one inferred from original dataset.

The networks obtained from the ARACNE algorithm using the original dataset and the samples generated by our framework are shown in Figure 6.17 and 6.18, respectively. Results on precision, recall and f-measure metrics are shown in Table 6.2. Note that we have taken the p-value which determines the the threshold for mutual information in the ARACNE algorithm as 10^{-6} since ARACNE is able to produce safe results for p-values up to 10^{-4} as stated in [44]. ARACNE infers undirected networks, therefore we have removed the direction of the edges in the reference network to be able compare the inferred networks. The network in Figure 6.18 shows that our generated dataset produced very good results with the ARACNE algorithm. From 38 undirected total number of edges in the reference network, ARACNE predicted 24 correct edges by using the dataset produced by our framework. The number of correctly predicted edges in the network inferred from the original dataset, on the other hand, is only 3, which is a very low number compared to the 24 correctly predicted edges from our

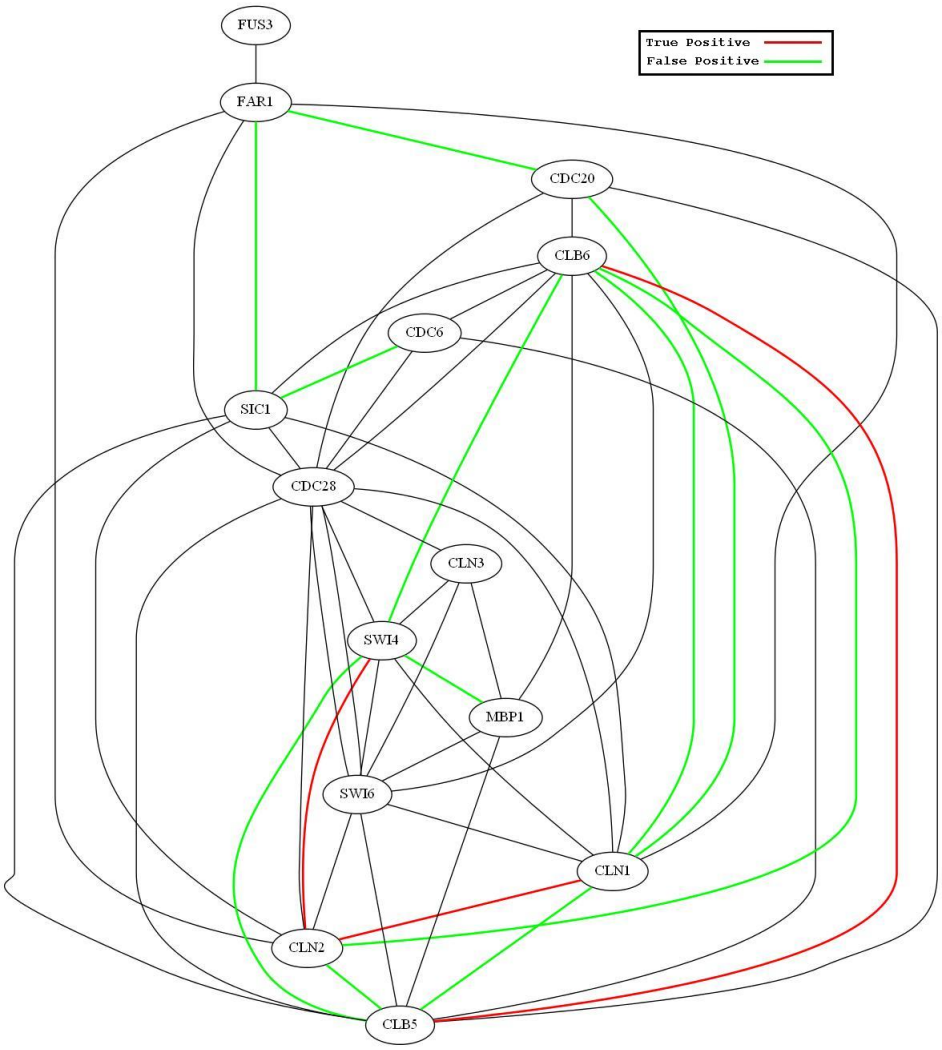

Figure 6.17: The regulatory network obtained from original dataset using ARACNE

generated dataset. Moreover, the precision, recall and f-measure values are also much better for the network inferred from our generated dataset than for the network inferred from the original dataset. Based on the precision values we can say that our dataset improves not only the number of correctly predicted edges but also the precision, i.e., the ratio of correctly predicted edges over the false positive edges. The recall values are much better for the network inferred from the generated data since the number of correctly predicted edges is almost 8 times larger than the number of correctly predicted edges in the network inferred from the original dataset. These results show that our generated dataset is certainly highly qualified since the network inferred from our generated dataset is much better than the network inferred from the original dataset. We can relate this result with the fact that we combined different gene regulation models and obtained a multi-model system dynamics simulation. This multi-model framework simulates the system in such a realistic way that the samples produced by the framework reflect the internal dynamics of gene regulation much better than the original samples. Therefore, we can confidently say that the samples generated by our

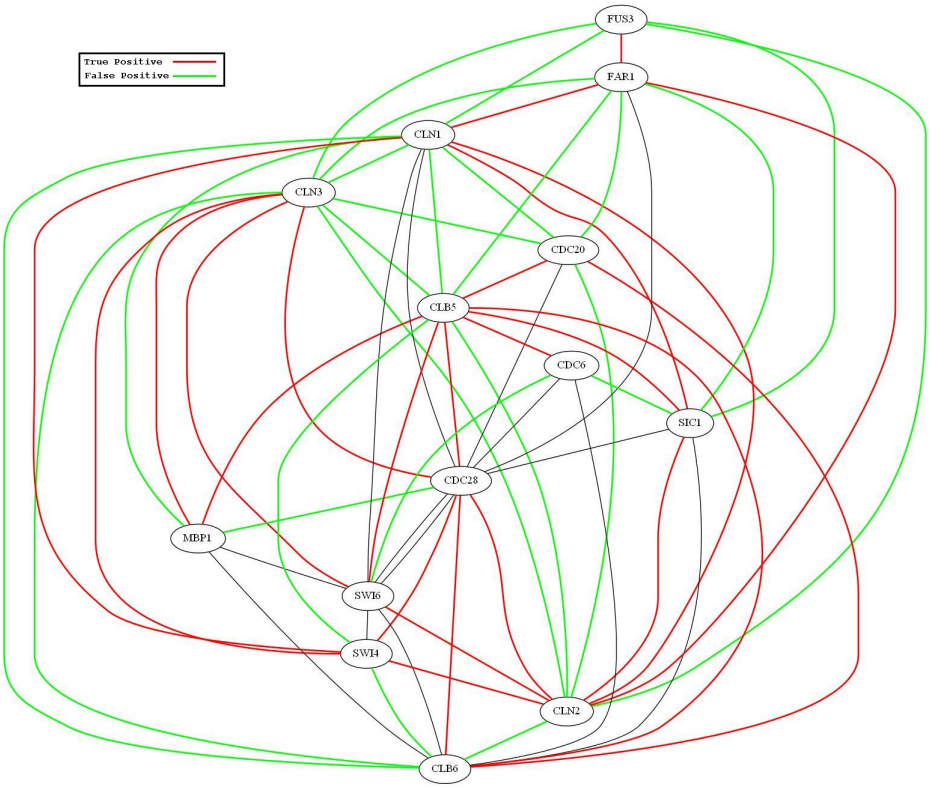

Figure 6.18: The regulatory network obtained from samples generated by our framework using ARACNE

Dataset	Precision	Recall	F-measure
Original	0.2143	0.0789	0.1161
Artificial	0.4898	0.6316	0.5517

Table6.2: Network evaluations for ARACNE algorithm

framework are highly qualified so that the inferred regulatory networks may lead to discover some new biological foundations that may not be caught from the original samples, which is one of the most important contributions of this chapter.

6.5.5 Experiments on Number Of Required Samples for Training

In this section, we tried to assess the required number of samples to train our multi-model framework. We try to assess this number empirically by checking the quality of the samples generated from different sets of training samples. We train our system with increasing number of training samples and produce always 50 samples from our trained framework. Then, we compared the generated 50 samples in terms of their metric results and try to understand when the quality of the generated samples stabilizes.

We generated two synthetic datasets by using the tool GeneNetWeaver (GNW) [58] in order to be able to carry out the experiments. The tool provides the complete *E.*

coli network automatically loaded and provide to extract some subnetworks of the *E. coli* network and simulate them. We have extracted two subnetworks of the *E. coli* network using the GNW tool. The extracted subnetworks are shown in Figure 6.19. Then, by again using the tool we simulated the extracted subnetworks and got two synthetic datasets comprising of 25 genes and 600 samples in the first one and 40 genes and 600 samples in the second one.

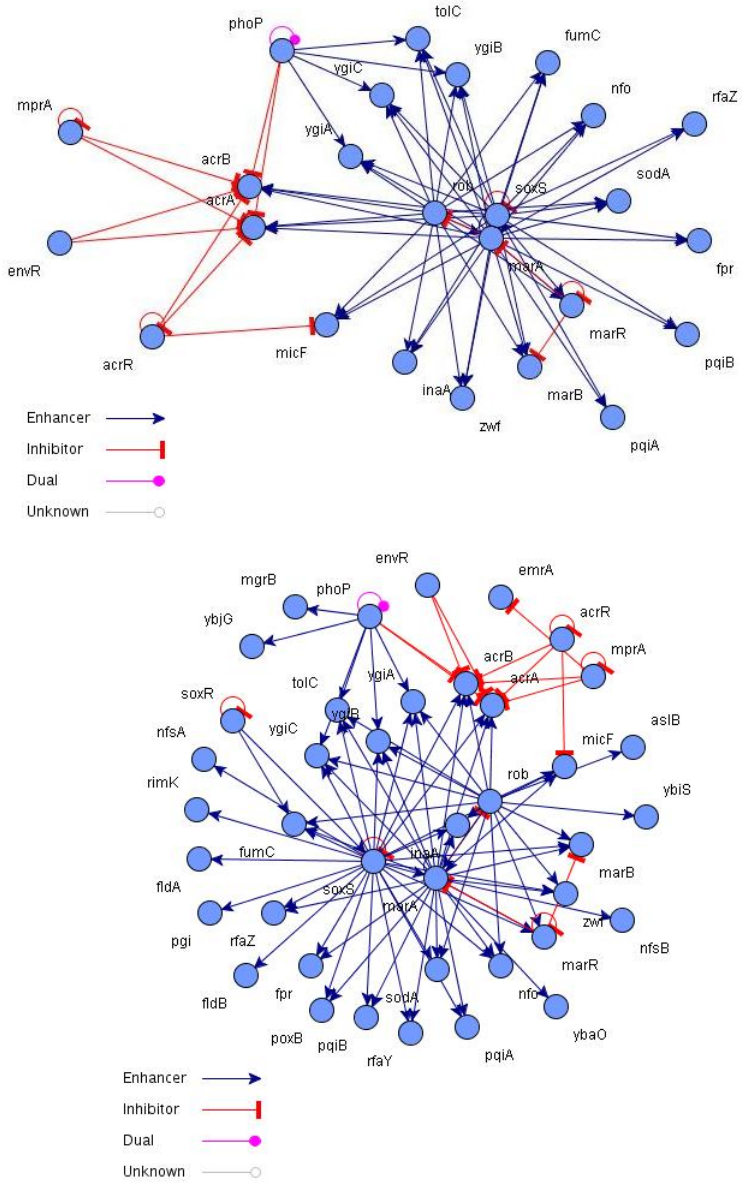

Figure 6.19: The *E. coli* subnetworks used for generating synthetic datasets

After having generated the synthetic datasets, we applied the following procedure to each dataset. We have separated the last 40 samples as test set. Then, successively we generated 50 samples by using the first $40, 60, ..., 560$ samples for training. After having those sample sets generated by different number of training samples, for each i^{th} sample set, we compared it with the $(i-1)^{th}$ sample set by checking the multi-

objective quality of the i^{th} sample set with respect to the $(i-1)^{th}$ sample set. The multi-objective quality is achieved by concatenating the $(i-1)^{th}$ and i^{th} sample sets and sorting them multi-objectively. In the first 50 samples of the concatenated and sorted 100 samples, we checked the percentage of the samples that are coming from the i^{th} sample set and use this value as the contribution of the i^{th} sample set to the $(i-1)^{th}$ sample set. At some point, we expect to observe a stabilization in terms of the contribution of the i^{th} sample set to the $(i-1)^{th}$ sample set. Figure 6.20 shows the results for the two synthetic datasets. Here, interestingly we see that contribution of each sample set to its previous sample set is decreasing exponentially, which supports our expectations on the number of required samples for training our system. Moreover, from the plots we can see that after having 240 samples for training our framework the contribution values nearly do not change. Therefore, empirically we can say that, 240 samples is the bound for the required number of samples to build our multi-model data enrichment framework. That is, enlarging the training samples up to 240 is expected to improve the quality of the generated samples by our framework. After having 240 samples, however, it is not necessary to enlarge the training samples since the quality of the generated samples most probably will not change significantly after having 240 samples for training as Figure 6.20 shows.

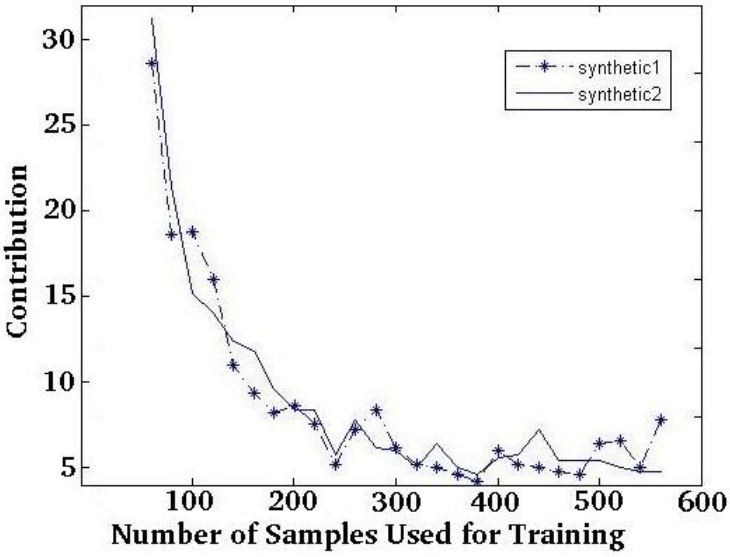

Figure 6.20: Comparison of generated sample sets for synthetic datasets

We have applied the same procedure explained above to the real life HUVECs dataset as well. We again split the last 40 samples as test set, and successively generated 50 samples by using the first $40, 60, ..., 360$ samples for training. Note that we have 400 samples in total in the HUVECs dataset. Then we calculated the contribution values for each i^{th} sample set just as explained above. Figure 6.21 shows the results. Here, we again see that the decrease in contributions occurs exponentially. There is only one jump between 220 and 240 after having 200 training samples. However, after having 240 samples for training the contributions again stabilizes. This verifies

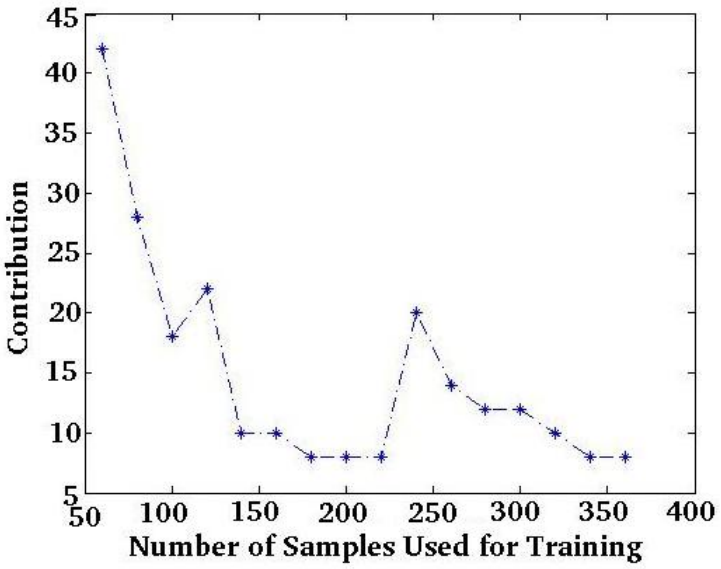

Figure 6.21: Comparison of generated sample sets for HUVECs dataset

our claim that 240 samples for training our multi-model framework will be enough to generate qualified samples. Because having more samples than 240 samples to train our framework most probably will not change the quality of the generated samples significantly as Figure 6.21 verifies once again.

Note that 240 is a rough bound for the required number of samples to train our framework. As explained and proved in Section 6.5.1, our multi-model framework still produces very good results even with very small number of samples such as melanoma dataset whose sample size is 31. By providing the bound on the required number of samples to train our framework, we both guide the scientists on how many real samples should be generated in the laboratory environment and give a way to estimate and bound the cost of the real life gene expression data generation experiments. We believe that this is an important contribution of our multi-model framework to the literature.

6.6 Discussion

In this chapter, we proposed a comprehensive multi-model data generation framework integrating four different generative models. The generative models carry different types of characteristics for gene regulation dynamics so that the built multi-model framework benefits from all of them as much as possible. In order to enrich the existing gene expression datasets, we generated samples from different generative models and pool them. Then, we evaluated the pooled samples by using three well-defined metrics: compatibility, diversity, and coverage. Having these three metrics evaluated, we outputted the best samples to the user. The selection mechanism considers these three metrics together, therefore applies a multi-objective selection mechanism based

on dominance of samples. The three metrics check how much the samples are close to the original samples, reasonably diverse from original samples, and cover the sample space. Therefore, produced gene expression samples sustain those three important characteristic and improve the resultant samples from different aspects.

Experimental results showed that our multi-model framework generates samples that are highly close to the original samples, always carrying reasonable amount of new information and cover the sample space successfully. Moreover, the experiments on multi-model justification showed that our multi-model approach certainly improves each single model. Our system adapts itself by choosing samples from different generative models for different datasets with respect to the quality of the generated samples, which proves the robustness of our multi-model framework. Moreover, we showed that the samples generated by our multi-model framework lead to infer much better regulatory networks than what the original samples lead to. Therefore, we can confidently say that merging different generative models into one multi-model framework not only improves the single model quality and produces qualitative samples, but also may lead to find some new biological relations. This means that without dealing with real life experimental difficulties, we can catch new biological relations by generating synthetic samples from a multi-model framework mitigating the need for costly real life experiments. Furthermore, empirically we tried to achieve a bound for the required number of samples to train our framework so that the generated samples are sufficiently qualitative. This estimation also provides a bound for the required number of real samples to be obtained in the laboratory environment, thereby a bound for the cost of the real life experiments.

Having successful results on sample quality, multi-model justifications, regulatory network inference and obtaining a bound for the required number of samples to train our framework, it can be stated that our multi-model framework can be used for many areas suffering from scarcity of samples, and mitigates the need for gene expression data.

CHAPTER 7

CONCLUSION AND FUTURE WORK

In this thesis, we have proposed three solutions for controlling gene regulation systems, and a multi-model gene expression data generation framework. The first control solution we proposed is for controlling fully observable gene regulation systems. The idea is to convert the available gene expression data into a series of experience tuples, and apply available Batch RL techniques to obtain approximate and generalized control policies directly from those experience tuples. Results show that we can generate control policies for regulation systems of several thousands of gene just in seconds, while existing studies cannot solve control problems even for several tens of genes. Interestingly, results also show that our generated approximate policies are almost as good as the ones generated by the existing optimal solutions. The second control solution we proposed is to solve the control problem of partially observable gene regulation systems. In order to do this, we have proposed a novel framework Batch Mode TD(λ), and its novel algorithm Least-Squares Fitted TD(λ) Iteration (LSFTDI), combining Batch RL works and Sutton's TD(λ) algorithm (1988). The idea is to obtain a probabilistic mapping of observations to actions by generalizing from the available gene expression data. Again, we have converted gene expression data into experience tuples, and applied our novel LSFTDI algorithm to those experience tuples. Results show that the stochastic policies LSFTDI generates are almost as good as the ones generated by the optimal solutions. Result also show that LSFTDI can produce successful stochastic policies for regulation systems of several thousands of genes, while existing studies cannot solve control problem of even several tens of genes. To our best knowledge, Batch Mode TD(λ) is the first framework solving non-Markovian decision tasks with limited number of experience tuples. The third solution we have proposed is to again control partially observable gene regulation systems. Unlike the second solution we proposed, this time we have tried to identify the actual internal states of the gene regulation system we want to control. We have again converted gene expression samples into a series of experience tuples, and apply Batch RL to assign an approximate observation-action value for each possible observation-action pairs. Then, we have used hidden state identification techniques to find the internal states and their Bayesian relationships with the observations, which leads to construct a POMDP. Results show that our constructed POMDPs are more successful than the previous studies in terms of

both solution quality and time requirements. To our best knowledge, Batch Mode POMDP Learning is the first study that can construct a POMDP directly from a limited number of experience tuples.

We have also proposed a novel multi-model gene expression data generation framework combining different computational models into a unified framework. Our idea is that, in order to capture biological relationships more accurately, instead of trying to improve a single computational model, integrating different models provides a more robust framework. We have integrated four generative models, PBN, ODE, HIMM and GA. What we do is to sample from each generative model separately and pool them together; then, based on a multi-objective selection mechanism, to rank the generated samples, and output the best ones. Results show that the generated samples from our multi-model framework not only very close to the original samples, but also always extend the informational content hold by the data. Results also show that our multi-model framework generates always higher quality samples than any single model it includes, and provides to capture biological relationships that cannot be captured by real gene expression samples.

As future work, we are planning to match our two solutions on controlling gene regulation systems and generating high quality artificial gene expression data. Since all of our proposed control solutions are obtaining a control policy directly from data, it is highly suitable to match these two studies. We plan to investigate the relationship between the produced control solutions and the number of samples in the datasets. Secondly, we are planning to extend our novel solutions for controlling partially observable GRNs, which are Batch Mode TD(λ) and Batch Mode POMDP Learning, for different non-Markovian decision tasks. Since both of the solutions do not depend on any computational model for gene regulation, they are highly suitable for employing different decision tasks. We are also planning to improve our proposed Batch Mode POMDP Learning method by integrating it with different state learning algorithms.

REFERENCES

[1] M. Bansal, G. D. Gatta, and D. Di Bernardo. Inference of Gene Regulatory Networks and Compound Mode of Action from Time Course Gene Expression Profiles. *Bioinformatics*, 22(7):815–822, Apr. 2006.

[2] R. Bellman. *Dynamic Programming*. Dover Publications, 1957.

[3] A. Bernard and A. Hartemink. Informative Structure Priors: Joint Learning of Dynamic Regulatory Networks from Multiple Types of Data. In A. R., D. A.K., H. L., J. T., and K. T., editors, *Pacific Symposium on Biocomputing 2005 (PSB05)*. World Scientific: New Jersey, 2005.

[4] D. P. Bertsekas. *Dynamic Programming and Optimal Control*. Academic Press, New York, 1976.

[5] M. Bittner, P. Meltzer, Y. Chen, Y. Jiang, E. Seftor, M. Hendrix, M. Radmacher, R. Simon, Z. Yakhini, A. Ben-Dor, N. Sampas, E. Dougherty, E. Wang, F. Marincola, C. Gooden, J. Lueders, A. Glatfelter, P. Pollock, J. Carpten, E. Gillanders, D. Leja, K. Dietrich, C. Beaudry, M. Berens, D. Alberts, and V. Sondak. Molecular Classification of Cutaneous Malignant Melanoma by Gene Expression Profiling. *Nature*, 406(6795):536–540, Aug. 2000.

[6] R. Bonneau, D. J. Reiss, P. Shannon, M. Facciotti, L. Hood, N. S. Baliga, and V. Thorsson. The Inferelator: An Algorithm for Learning Parsimonious Regulatory Networks from Systems-biology Data Sets de novo. *Genome Biology*, 7(5):R36, 2006.

[7] D. Bryce, M. Verdicchio, and S. Kim. Planning Interventions in Biological Networks. *ACM Transactions Intelligent Systems Technology*, 1(2):11:1–11:26, 2010.

[8] L. Busoniu, R. Babuska, B. D. Schutter, and D. Ernst. *Reinforcement Learning and Dynamic Programming Using Function Approximators*. CRC Press, 2010.

[9] A. R. Cassandra, L. P. Kaelbling, and M. L. Littman. Acting Optimally in Partially Observable Stochastic Domains. In *Proceedings of the Twelfth National Conference on Artificial intelligence (vol. 2)*, AAAI'94, pages 1023–1028, Menlo Park, CA, USA, 1994. American Association for Artificial Intelligence.

[10] W. Chen, H. Lu, M. Wang, and C. Fang. Gene Expression Data Classification Using Artificial Neural Network Ensembles Based on Samples Filtering. *Proceedings of the International Conference on Artificial Intelligence and Computational Intelligence*, 1:626–628, 2009.

[11] L. Chrisman. Reinforcement Learning with Perceptual Aliasing: the Perceptual Distinctions Approach. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, AAAI'92, pages 183–188. AAAI Press, 1992.

[12] A. Datta, A. Choudhary, M. L. Bittner, and E. R. Dougherty. External Control in Markovian Genetic Regulatory Networks. *Machine Learning*, 52:169–191, 2003.

[13] A. Datta, A. Choudhary, M. L. Bittner, and E. R. Dougherty. External Control in Markovian Genetic Regulatory Networks: The Imperfect Information Case. *Bioinformatics*, 20(6):924–930, Apr. 2004.

[14] P. D'Haeseleer, X. Wen, S. Fuhrman, S. Fuhrman, and R. Somogyi. Linear Modeling of mRNA Expression Levels during CNS Development and Injury. pages 41–52, 1999.

[15] D. di Bernardo, M. J. Thompson, T. S. Gardner, S. E. Chobot, E. L. Eastwood, A. P. Wojtovich, S. J. Elliott, S. E. Schaus, and J. J. Collins. Chemogenomic Profiling on a Genome-wide Scale Using Reverse-engineered Gene Networks. *Nature Biotechnology*, 23(3):377–383, 2005.

[16] L. Ein-Dor, O. Zuk, and E. Domany. Thousands of Samples are Needed to Generate a Robust Gene List for Predicting Outcome in Cancer. *PNAS*, 103:5923 – 5928, 2006.

[17] U. Erdogdu. *Efficient Partially Observable Markov Decision Process Based Formulation of Gene Regulatory Network Control Problem.* PhD thesis, Ankara, Turkey, 2012.

[18] U. Erdogdu, R. Alhajj, and F. Polat. The Benefit of Decomposing POMDP for Control of Gene Regulatory Networks. In *Proceedings of the 2011 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology*, pages 381–385. IEEE Computer Society, 2011.

[19] U. Erdoğdu, M. Tan, R. Alhajj, F. Polat, D. Demetrick, and J. Rokne. Employing Machine Learning Techniques for Data Enrichment: Increasing the Number of Samples for Effective Gene Expression Data Analysis. In *Bioinformatics and Biomedicine (BIBM), 2011 IEEE International Conference on*, pages 238 –242, nov. 2011.

[20] U. Erdoğdu, M. Tan, R. Alhajj, F. Polat, J. Rokne, and D. Demetrick. Integrating Machine Learning Techniques into Robust Data Enrichment Approach and Its Application to Gene Expression Data. *International Journal of Data Mining and Bioinformatics*, 2012.

[21] D. Ernst, P. Geurts, and L. Wehenkel. Tree-Based Batch Mode Reinforcement Learning. *Journal of Machine Learning Research*, 6:503–556, 2005.

[22] B. Faryabi, A. Datta, and E. Dougherty. On Approximate Stochastic Control in Genetic Regulatory Networks. *Systems Biology, IET*, 1(6):361 –368, 2007.

[23] B. Faryabi, A. Datta, and E. R. Dougherty. On Reinforcement Learning in Genetic Regulatory Networks. In *Proceedings of the 2007 IEEE/SP 14th Workshop on Statistical Signal Processing*, SSP '07, pages 11–15, Washington, DC, USA, 2007. IEEE Computer Society.

[24] C. Fonseca, P. Fleming, et al. Genetic Algorithms for Multiobjective Optimization: Formulation, Discussion and Generalization. In *Proceedings of the fifth international conference on genetic algorithms*, volume 1, page 416. Citeseer, 1993.

[25] H. Franz, C. Ullmann, A. Becker, M. Ryan, S. Bahn, T. Arendt, M. Simon, S. Paabo, and P. Khaitovich. Systematic Analysis of Gene Expression in Human Brains Before and After Death. *Genome Biology*, 6(13):R112, 2005.

[26] T. S. Gardner, D. di Bernardo, D. Lorenz, and J. J. Collins. Inferring Genetic Networks and Identifying Compound Mode of Action via Expression Profiling. *Science*, 301(5629):102–105, 2003.

[27] T. S. Gardner, D. di Bernardo, D. Lorenz, and J. J. Collins. Inferring Genetic Networks and Identifying Compound Mode of Action via Expression Profiling. *Science*, 301(5629):102–105, July 2003.

[28] J. Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, 1975.

[29] D. Hurley, H. Araki, Y. Tamada, B. Dunmore, D. Sanders, S. Humphreys, M. Affara, S. Imoto, K. Yasuda, Y. Tomiyasu, K. Tashiro, C. Savoie, V. Cho, S. Smith, S. Kuhara, S. Miyano, D. S. Charnock-Jones, E. J. Crampin, and C. G. Print. Gene Network Inference and Visualization Tools for Biologists: Application to New Human Transcriptome Datasets. *Nucleic Acids Research*, 2011.

[30] T. Jaakkola, S. P. Singh, and M. I. Jordan. Reinforcement Learning Algorithm for Partially Observable Markov Decision Problems. In G. Tesauro, D. S. Touretzky, and T. K. Leen, editors, *Advances in Neural Information Processing Systems*, pages 345–352. MIT Press, Cambridge, MA, 1994.

[31] H. D. Jong. Modeling and Simulation of Genetic Regulatory Systems: A Literature Review. *Journal of Computational Biology*, 9:67–103, 2002.

[32] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement Learning: A Survey. *Journal of Aftificial Intelligence Research*, 4:237–285, 1996.

[33] M. Kanehisa and S. Goto. KEGG: Kyoto Encyclopedia of Genes and Genomes. *Nucleic Acids Res.*, 28:27–30, 2000.

[34] S. A. Kauffman. *The Origins of Order: Self-Organization and Selection in Evolution.* Oxford University Press, USA, 1 edition, June 1993.

[35] M. Kim, S. B. Cho, and J. H. Kim. Mixture-model Based Estimation of Gene Expression Variance from Public Database Improves Identification of Differentially Expressed Genes in Small Sized Microarray Data. *Bioinformatics*, 26:486–492, 2010.

[36] S. Kim, S. Imoto, and S. Miyano. Dynamic Bayesian Network and Nonparametric Regression for Nonlinear Modeling of Gene Networks from Time Series Gene Expression Data. In *Proceedings of the First International Workshop on Computational Methods in Systems Biology*, CMSB '03, pages 104–113, London, UK, UK, 2003. Springer-Verlag.

[37] S. Kim, H. Li, E. R. Dougherty, N. Cao, Y. Chen, M. Bittner, and E. B. Suh. Can Markov Chain Models Mimic Biological Regulation? *Journal of Biological Systems*, 10:337–357, 2002.

[38] S. Lange, T. Gabel, and M. Riedmiller. *Reinforcement Learning: State of the Art.* Springer, 2011.

[39] M. Lee and G. Whitmore. Power and Sample Size for DNA Microarray Studies. *Statistics in Medicine*, 21(23):3543–3570, 2002.

[40] Z. Li, P. Li, A. Krishnan, and J. Liu. Large-scale Dynamic Gene Regulatory Network Inference Combining Differential Equation Models with Local Dynamic Bayesian Network Analysis. *Bioinformatics*, 27(19):2686–2691, Oct. 2011.

[41] L.-J. Lin. Self-Improving Reactive Agents Based on Reinforcement Learning, Planning and Teaching. *Machine Learning*, 8(3-4):293–321, 1992.

[42] M. L. Littman, T. L. Dean, and L. P. Kaelbling. On the Complexity of Solving Markov Decision Problems. In *Proceedings of the Eleventh International Conference on Uncertainty in Artificial Intelligence*, pages 394–402, 1995.

[43] D. Marbach, J. C. Costello, R. Kuffner, N. M. Vega, R. J. Prill, D. M. Camacho, K. R. Allison, M. Kellis, J. J. Collins, and G. Stolovitzky. Wisdom of Crowds for Robust Gene Network Inference. *Nature Methods*, 9(8):796–804, Aug. 2012.

[44] A. Margolin, I. Nemenman, K. Basso, C. Wiggins, G. Stolovitzky, R. Favera, and A. Califano. ARACNE: An Algorithm for the Reconstruction of Gene Regulatory Networks in a Mammalian Cellular Context. *BMC Bioinformatics*, 7(Suppl 1):S7+, 2006.

[45] A. Mccallum. Instance-Based State Identification for Reinforcement Learning. In *Advances in Neural Information Processing Systems 7*, NIPS 7, pages 377–384. MIT Press, 1994.

[46] A. K. McCallum. *Reinforcement Learning with Selective Perception and Hidden State.* PhD thesis, The University of Rochester, 1996.

[47] R. A. McCallum. Overcoming Incomplete Perception with Utile Distinction Memory. In *Proceedings of the Tenth International Conference on Machine Learning*, ICML'93, 1993.

[48] R. A. Mccallum. Instance-Based Utile Distinctions for Reinforcement Learning with Hidden State. In *Proceedings of the Twelfth International Conference on Machine Learning*, ICML'95, pages 387–395, 1995.

[49] M. Mitchell. *An Introduction to Genetic Algorithms.* The MIT press, 1998.

[50] D. Ormoneit and S. Sen. Kernel-Based Reinforcement Learning. *Machine Learning*, 49:161–178, 2002.

[51] R. Pal, A. Datta, and E. R. Dougherty. Optimal Infinite Horizon Control for Probabilistic Boolean Networks. *IEEE Transactions on Signal Processing*, 54:2375–2387, 2006.

[52] W. Pan, J. Lin, and C. Le. How Many Replicates of Arrays are Required to Detect Gene Expression Changes in Microarray Experiments? A Mixture Model Approach. *Genome Biol*, 3(5):0022–1, 2002.

[53] R. Parsons, S. Forrest, and C. Burks. Genetic Algorithms for DNA Sequence Assembly. In *Proceedings of the International Conference on Intelligent Systems for Molecular Biology*, page 318. AAAI Press, 1993.

[54] J. T. Pedersen and J. Moult. Genetic Algorithms for Protein Structure Prediction. *Current Opinion in Structural Biology*, 6(2):227–231, 1996.

[55] G. Piatetsky-Shapiro, T. Khabaza, and S. Ramaswamy. Capturing Best Practice for Microarray Gene Expression Data Analysis. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 407–415, 2003.

[56] P. Poupart. *Exploiting Structure to Efficiently Solve Large Scale Partially Observable Markov Decision Processes.* PhD thesis, Toronto, Ont., Canada, Canada, 2005.

[57] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice-Hall, Englewood Cliffs, NJ, 2nd edition edition, 2003.

[58] T. Schaffter, D. Marbach, and D. Floreano. GeneNetWeaver: In Silico Benchmark Generation and Performance Profiling of Network Inference Methods. *Bioinformatics*, 27(16):2263–2270, 2011.

[59] M. Schilling, T. Maiwald, S. Bohl, M. Kollmann, C. Kreutz, J. Timmer, and U. Klingmüller. Quantitative Data Generation for Systems Biology: the Impact of Randomisation, Calibrators and Normalisers. *IEE Proceedings on Syst Biol (Stevenage)*, 152(4):193–200, 2005.

[60] I. Shmulevich, E. R. Dougherty, S. Kim, and W. Zhang. Probabilistic Boolean Networks: A Rule-Based Uncertainty Model for Gene Regulatory Networks. *Bioinformatics*, 18(2):261–274, 2002.

[61] S. P. Singh, T. Jaakkola, and M. I. Jordan. Learning Without State-Estimation in Partially Observable Markovian Decision Processes. In *Proceedings of the Eleventh International Conference on Machine Learning*, pages 284–292. Morgan Kaufmann, 1994.

[62] M. T. J. Spaan and N. Vlassis. Perseus: Randomized Point-based Value Iteration for POMDPs. *Journal of Artificial Intelligence Research*, 24(1):195–220, 2005.

[63] P. Spellman, G. Sherlock, M. Zhang, V. Iyer, K. Anders, M. Eisen, P. Brown, D. Botstein, and B. Futcher. Comprehensive Identification of Cell Cycle-regulated Genes of the Yeast Saccharomyces Cerevisiae by Microarray Hybridization. *Molecular Biology of Cell*, 9:3273–3297, 1998.

[64] R. S. Sutton. Learning to Predict by the Methods of Temporal Differences. *Machine Learning*, 3(1):9–44, 1988.

[65] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, MA, USA, 1998.

[66] G. Sywerda. Uniform Crossover in Genetic Algorithms. In *Proceedings of the International Conference on Genetic Algorithms*, pages 2–9, 1989.

[67] M. Tan, M. AlShalalfa, R. Alhajj, and F. Polat. Influence of Prior Knowledge in Constraint-Based Learning of Gene Regulatory Networks. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 8:130–142, 2011.

[68] M. van Iterson, P. 't Hoen, P. Pedotti, G. Hooiveld, J. den Dunnen, G. van Ommen, J. Boer, and R. Menezes. Relative Power and Sample Size Analysis on Gene Expression Profiling Data. *BMC Genomics*, 10(1):439, 2009.

[69] C. Watkins. *Learning from Delayed Rewards*. PhD thesis, University of Cambridge,England, 1989.

[70] J. Wildenhain and E. J. Crampin. Reconstructing Gene Regulatory Networks: from Random to Scale-free Connectivity. *IEEE Systems biology*, 153(4):247–256, July 2006.

[71] I. H. Witten. Adaptive Text Mining: Inferring Structure from Sequences. *Journal of Discrete Algorithms*, 2(2):137–159, 2004.

[72] R. Q. Yiqian Zhou and A. Sacan. Data Simulation and Regulatory Network Reconstruction from Time-series Microarray Data Using Stepwise Multiple Linear Regression. *Network Modeling Analysis in Health Informatics and Bioinformatics*, 2012.

[73] J. Yu, V. A. Smith, P. P. Wang, A. J. Hartemink, and E. D. Jarvis. Advances to Bayesian Network Inference for Generating Causal Networks from Observational Biological Data. *Bioinformatics*, 20(18):3594–3603, Dec. 2004.