Efficient inference algorithms for near-deterministic systems

Shaunak Chatterjee

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2013-219
http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-219.html

December 18, 2013

Acknowledgement

I would like to thank my advisor, Prof. Stuart Russell, for his guidance, advice and patience over the course of the last six years. I am also grateful to my committee members and fellows RUGS members for their professional help. Finally, I would like to thank my parents, my family, my fiancee, and my friends in Berkeley, without whose support, this dissertation would not have possible.

Efficient inference algorithms for near-deterministic systems

by

Shaunak Chatterjee

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Stuart Russell, Chair
Professor Jaijeet Roychowdhury
Professor Dan Klein
Professor Ian Holmes

Fall 2013

Efficient inference algorithms for near-deterministic systems

Abstract

Efficient inference algorithms for near-deterministic systems

by

Shaunak Chatterjee

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Stuart Russell, Chair

This thesis addresses the problem of performing probabilistic inference in stochastic systems where the probability mass is far from uniformly distributed among all possible outcomes. Such *near-deterministic* systems arise in several real-world applications. For example, in human physiology, the widely varying evolution rates of physiological variables make certain trajectories much more likely than others; in natural language, a very small fraction of all possible word sequences accounts for a disproportionately high amount of probability under a language model. In such settings, it is often possible to obtain significant computational savings by focusing on the outcomes where the probability mass is concentrated. This contrasts with existing algorithms in probabilistic inference—such as junction tree, sum product, and belief propagation algorithms—which are well-tuned to exploit conditional independence relations.

The first topic addressed in this thesis is the structure of discrete-time temporal graphical models of near-deterministic stochastic processes. We show how the structure depends on the ratios between the size of the time step and the effective rates of change of the variables. We also prove that accurate approximations can often be obtained by sparse structures even for very large time steps. Besides providing an intuitive reason for causal sparsity in discrete temporal models, the sparsity also speeds up inference.

The next contribution is an eigenvalue algorithm for a linear factored system (e.g., dynamic Bayesian network), where existing algorithms do not scale since the size of the system is exponential in the number of variables. Using a combination of graphical model inference algorithms and numerical methods for spectral analysis, we propose an approximate spectral algorithm which operates in the factored representation and is exponentially faster than previous algorithms.

The third contribution is a temporally abstracted Viterbi (TAV) algorithm. Starting with a spatio-temporally abstracted coarse representation of the original problem, the TAV algorithm iteratively refines the search space for the Viterbi path via spatial and temporal refinements. The algorithm is guaranteed to converge to the optimal solution with the use of admissible heuristic costs in the abstract levels and is much faster than the Viterbi algorithm for near-deterministic systems.

The fourth contribution is a hierarchical image/video segmentation algorithm, that shares some of the ideas used in the TAV algorithm. A supervoxel tree provides the abstraction hierarchy for this application. The algorithm starts working with the coarsest level supervoxels, and refines portions of the tree which are likely to have multiple labels. Several existing segmentation algorithms can be used to solve the energy minimization problem in each iteration, and admissible heuristic costs once again guarantee optimality. Since large contiguous patches exist in images and videos, this approach is more computationally efficient than solving the problem at the finest level of supervoxels.

The final contribution is a family of Markov Chain Monte Carlo (MCMC) algorithms for near-deterministic systems when there exists an efficient algorithm to sample solutions for the corresponding deterministic problem. In such a case, a generic MCMC algorithm's performance worsens as the problem becomes more deterministic despite the existence of the efficient algorithm in the deterministic limit. MCMC algorithms designed using our methodology can bridge this gap.

The computational speedups we obtain through the various new algorithms presented in this thesis show that it is indeed possible to exploit near-determinism in probabilistic systems. Near-determinism, much like conditional independence, is a potential (and promising) source of computational savings for both exact and approximate inference. It is a direction that warrants more understanding and better generalized algorithms.

Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgments

This dissertation would not have been possible without the support, encouragement and help of my advisors, colleagues, friends, and family.

First and foremost, I would like to thank my advisor, Prof. Stuart Russell. He was very patient with me during my initial years in graduate school, very generous with both his time and advice even during his years as the Chair of the Department. His guidance was crucial in identifying the key problems addressed in this dissertation. At the same time, he has always encouraged me to explore new ideas and pursue my academic interests, even if they did not conform to my current research agenda. His influence has been instrumental in improving my writing, research and thinking skills over my graduate career.

I am very grateful to my committee members — Professors Jaijeet Roychowdhury, Dan Klein and Ian Holmes. Prof. Roychowdhury has especially been very helpful with several detailed discussions on possible solutions for the eigenvalue computation problem. His ideas on numerical methods were instrumental in designing the current solution. Prof. Holmes and Prof. Klein were also helpful with their suggestions on guiding the research in this dissertation by serving on my qualifying examination committee.

I would also like to thank all the professors at Berkeley and at the Indian Institute of Technology (IIT) Kharagpur (my undergrad school), who have shaped my education and research skills. Several teachers from high school have also been very influential in this journey. Prof. Rene Vidal (Johns Hopkins University) was a key collaborator in the work on hierarchical video segmentation.

My fellow RUGS members have provided a constant reservoir of new research ideas, stimulating academic (and non-academic) discussions, useful suggestions, and the weekly dose of *Gregoire*: Norm Aleks, Nimar Arora, Emma Brunskill, Kevin Canini, Daniel Duckworth, Yusuf Bugra Erol, Nick Hay, Gregory Lawrence, Lei Li, Akihiro Matsukawa, David Moore, Rodrigo De Salvo Braz, Fei Sha, Siddharth Srivastava, Erik Sudderth, Jason Wolfe. In particular, Norm Aleks was my first collaborator in graduate school and helped me out a lot during the initial semesters. Jason Wolfe has always been the person I turned to, to discuss an idea or get some feedback. Emma, Nick, Dave, Sid, Lei and Nimar have also been very helpful on numerous instances.

I am also grateful to the anonymous reviewers at various conferences, whose feedback has helped me better organize and present my research. I would also like to thank Intel Corporation, UC Discovery Program and the NSF (grant no. IIS-0904672) for supporting and generously funding my research over the years.

Finally, none of this would have been possible without the continued support of my family and friends. My parents, Dr. Ranjana Chatterjee and Dr. Sukanta Chatterjee, and my brother Sourav, have always unconditionally stood by me and urged me to pursue my dreams. I cannot thank them enough for all that they have done for me over the last three decades. My fiancee, Aastha Jain, has been inspiring me to become a better researcher (and person) and has played more than a supporting role in this venture (she was a collaborator in the video segmentation work). To all my friends in Berkeley — Ajith, Jayakanth, Himanshu, Arka, Godhuli, Maniraj, Payel, Soumen, Anindya, Debkishore, Arka, Raj, Momo, Piyush — thank you for being around all these years and making it so much fun!

Chapter 1

Introduction

One of the most important aspects of intelligence in humans and machines is the ability to *reason about uncertainty*: to analyze the relevant available information, consider all the different possibilities and act upon the resulting conclusions. Consider the problem of deciding whether or not to arrange an outdoor picnic tomorrow (assuming that you do not have immediate access to a weather forecast). Your decision would be based on the chances of rainfall tomorrow and could incorporate factors like today's temperature, cloud cover and the current season. A more detailed model might also consider the amount of rain in the past couple of days.

Uncertainty is a result of one or more among several factors. We could have partial or noisy observation of the world. Another possible source of uncertainty is the non-deterministic relationship between variables. This non-determinism could be either innate or a result of a partial model (i.e., due to a lack of detail in representation and/or understanding). In this example, even if we include much more detailed information about yesterday's weather, we cannot get a deterministic prediction for tomorrow's rainfall (as any experienced meteorologist would tell you).

Scientists and statisticians have long recognized the need for a common framework to represent and reason about systems with uncertainty. A probabilistic model is a formalization to express the stochastic relations among variables in a system. The model defines a probability distribution over all (i.e., a set of mutually exclusive and exhaustive) possibilities, thereby facilitating reasoning based on the relative importance of various outcomes.

A probabilistic model is an example of a *declarative representation* which separates the two key aspects of *knowledge representation* and *reasoning*. The representation has its own well-defined semantics, which are independent of the algorithms that can be applied to the model. This has enabled the design of various algorithms that are applicable for any application that can be expressed as a probabilistic model.

In a probabilistic model, there are several inter-related variables, but some variables might

be independent of one another when we know about a third set of variables. For instance, the likelihood of rain today might not depend on yesterday's cloud cover if we know about today's cloud cover. Such *conditional independence* relations can be expressed in a graph, and such a representation is called a *probabilistic graphical model*. Algorithms which exploit such conditional independence are much faster than ones that do not. Graphical models are explained in greater detail in Chapter 2.

While there exists several algorithms to exploit the conditional independence relations in graphical models, these algorithms are not designed to exploit the actual nature of the dependence relations. For instance, if yesterday's cloud cover had a very weak effect on today's likelihood of rain, it might be possible to ignore that effect and still obtain the correct solution or accrue a very small error. In this thesis, we focus on a particular type of dependence relation where there is very limited (but non-zero) stochasticity (hence the dependence relation is near-deterministic). Algorithms designed for graphical models typically do well when the dependence relations are quite stochastic, but in many near-deterministic cases they are *unnecessarily slow* as they cannot (intelligently) ignore the large set of very unlikely possibilities.

In this thesis, we propose various ways to leverage near-determinism and obtain significant speedups over runtimes from existing algorithms on the same problems. Each approach generally has two components (in line with the declarative representation):

Firstly, we construct a modified representation (i.e., an abstract model) which is able to transform the original probabilistic model in a way that makes it easier to exploit the near-deterministic relations. This modified model could be approximate. The actual abstraction scheme varies based on the nature of the inference problem (maximization vs marginalization) and also on the relative amounts of near-determinism in different edges.

Secondly, we need to design modified algorithms which will work on either the original or the modified representations and perform necessary refinements to obtain the final inference objective. Often, these algorithms are slight variants of algorithms designed for the original representation, since the modified model is designed to make near-determinism more exploitable. It should be noted that for certain algorithms, one of the two components (modifying the abstraction and modifying the algorithm) might not be needed.

The ultimate goal of this line of work will be to exhaustively delineate strategies to exploit the actual nature of the dependence relations in a graphical model – something that is mostly ignored by the current algorithms which only utilize the graphical structure. This will lead to much faster inference and learning in several real-world applications as the underlying mathematical nature of many of these dependence relations are far from being completely random.

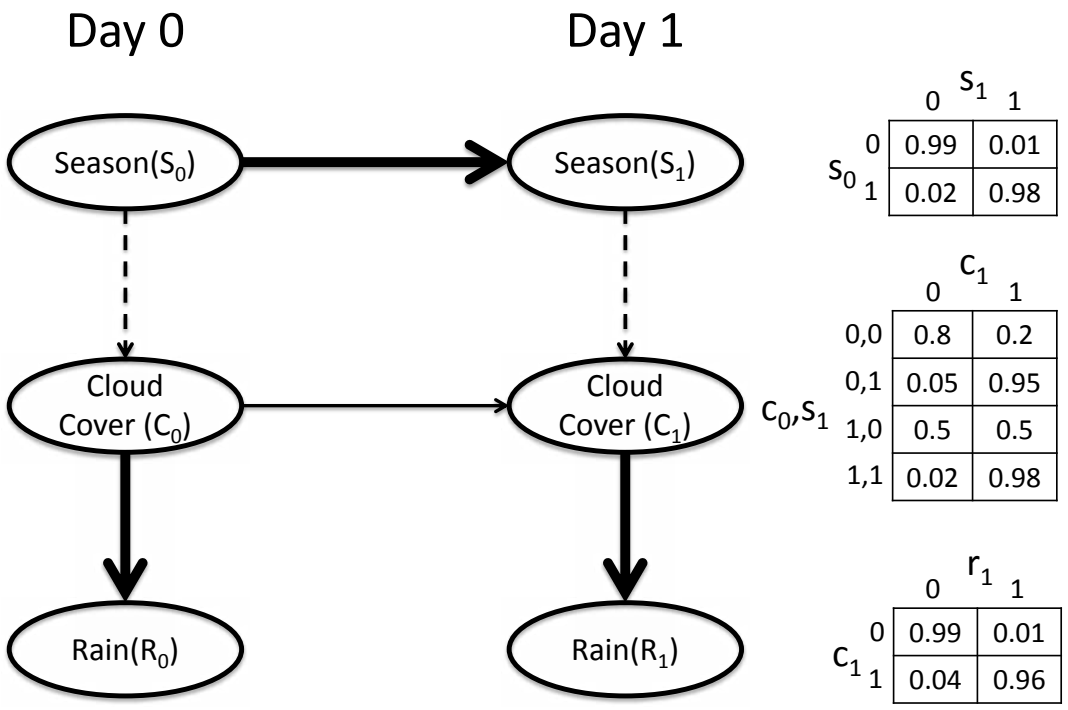


Figure 1.1: The rain prediction example. There are 3 binary variables in this model for each day. The variables for day 0 are 1. Current season (S_0): "not monsoon" or "monsoon", 2. Cloud cover (C_0): "clear" or "cloudy", and 3. Rain (R_0): "no rain" or "rain". The same variables exist for day 1 – S_1, C_1 and R_1. as well The edges denote conditional dependence relations, and the tables are conditional probability tables, where the conditioning variable(s) are on the vertical axis.

For each variable, the two possible values are denoted by 0 and 1 respectively in the table due to space constraints. For S, 0: "Not monsoon" and 1: "monsoon". For C, 0: "not cloudy" and 1: "cloudy". For R, 0: "No rain" and 1: "Rain". The bold edges mark near-deterministic conditional dependence relations, while the dotted edge marks a contextual near-deterministic relationship. The dependence of S_1 on S_0 is near-deterministic since seasons change very infrequently, while R_1 on C_1 is near-deterministic because clouds very often bring rains. During the monsoon season, the dependence of C_1 on S_1 becomes near-deterministic , but it is otherwise quite stochastic.

1.1 Near-determinism in graphical models

While there has been extensive work on designing algorithms to exploit conditional independence relations in probabilistic models, not much attention has been given to the nature of the direct dependence relations. For example, in the rain prediction example introduced before and shown in Figure 1.1, the season variable on day 1 (S_1) is almost always the same as the season variable on day 0 (S_0) — i.e., their dependence is almost deterministic or "near-deterministic." This is also true for the dependence between cloud cover and rainfall on the same day. The relationship between season and cloud cover is near-deterministic during the monsoon season, but much more stochastic during other seasons. In a near-deterministic system, the probability mass (or density in the case of continuous variables) is concentrated on a small subset (subspace) of states. The probability mass is contained in a single state in the limiting case of a deterministic system.

While we shall define the notion of near-determinism mathematically in Chapter 2, the common result of near-determinism is the concentration of probability mass (or density) in a small portion of the overall joint state space. As a result, there exists an opportunity to do efficient inference by focusing on the high probability states. Depending on how we handle the remaining low probability states, the inference can be approximate (yet still fairly accurate) or exact. The varying amounts of determinism in probabilistic systems are completely ignored by current inference algorithms for graphical models, which can only distinguish between the presence and absence of a dependence.

In linear system literature, researchers have looked at tackling some aspects of near-determinism with spectral transformations. However, these approaches were designed for the simulation task, and do not apply directly to other inference tasks that are common in probabilistic models. Secondly, the transition model in such systems is not factored and there are no conditional independence relations. Systems expressed via graphical models tend to be much larger and for such models even creating the transition model explicitly (as needed by conventional spectral analysis) is not feasible and hence these methods are not well-suited for such systems.

The primary reason to focus on near-determinism is twofold. First, the ubiquity of near-determinism in real-life applications and second, the possibility afforded by near-determinism to design much faster inference algorithms for the same problems is intriguing.

There are several instances of near-deterministic dependence relations in the everyday world. They exist in several aspects of human physiology. For instance, in a temporal model of the cardiovascular system, the elasticity of blood vessels changes very, very slowly and hence the dependence on the elasticity in the previous time step is near-deterministic (in a 1-second or 1-minute time step model).

Similarly, in natural language, any word in a valid English sentence is most likely to be followed by one among a very "small" set of words ("small" as compared to the full English vocabulary). Thus, the probability mass of a language model is concentrated on a very small subset of all possible English word sequences. The same holds for possible phonetic sequences in speech analysis.

In computer vision, large contiguous patches in an image or video are occupied by the same object. As a result, two contiguous pixels are very likely to belong to the same object which makes the dependence relation between their object labels. Thus, pixel labelings which assign the same label to large, contiguous patches are much more likely than other pixel labelings — thus the likelihood is concentrated on a small subset of the label space. Some of these examples are better described as near-static. From a computational perspective, a near-static and a near-deterministic system are analogous.

An interesting characteristic of near-determinism is its integral connection to space and time. For the same application, changing the granularity of the state space or the size of the time step (in a temporal model) can affect the degree of near-determinism in the dependence relations. For instance, in a 1-decade time step model, the elasticity of blood vessels is no longer near-deterministic . Conversely, in a 1-second model, body temperature evolution is near-deterministic (but that is not true for a 1-hour model). Similarly, a person's travel itinerary (for a 1-day time step model) can be very near-deterministic at the country-scale but quite random at the zip code scale (but still largely limited to a small set of well-connected zip codes). This characteristic hints at the possibility of using hierarchical methods (in space and/or time) in various ways to exploit the near-determinism for different inference problems.

In this dissertation, we present four primary results which use near-determinism to speed up existing inference algorithms:

- We show how the interplay of near-determinism and time step size affects the structure of causality in temporal models and how this can be used to do faster simulation and inference.
- We present an approximate eigenanalysis technique for factored systems.
- We design a spatio-temporally abstracted maximization algorithm which can exploit near-determinism at different scales of the model in both space and time.
- We describe a general method to build fast MCMC algorithms for near-deterministic problems, which are inspired by algorithms for corresponding problems in the deterministic realm.

1.2 Outline of the thesis

Chapter 2 begins with a detailed review of background material on graphical models, various instances of important graphical models, related inference algorithms, a brief overview of linear systems and a mathematical definition of near-determinism . Chapter 2 is divided into four sections: the first formally introduces graphical models and describes in a fair amount of detail some of the main inference algorithms for graphical models. These inference algorithms are heavily

drawn upon to design the new algorithms for near-deterministic systems. The second section introduces two families of graphical models – the hidden Markov model (HMM) and the dynamic Bayesian network (DBN) – and their related inference algorithms. The third section is a brief overview of linear systems which will be helpful for Chapter 4. The fourth section describes the notion of near-determinism mathematically and presents a couple of illustrative examples to show the potential for computational savings in near-deterministic systems and how this potential grows as the size of the system increases.

Next, Chapters 3, 4, 5, 6 and 7 describe the primary contribution of the thesis.

Chapter 3 studies the effect of near-determinism on graphical models in a temporal setting – namely, in dynamic Bayesian networks (DBNs). We show that near-determinism in a small-time-step model can result in very sparse large-time-step models (which are approximate but very accurate), whereas traditional graphical model wisdom suggests that the large-time-step model be fully connected. The sparse DBN models for larger time-steps lead to very fast simulation and inference using existing DBN algorithms. This chapter also provides some insights into the implicit approximations human experts make while proposing any finite-time-step model.

Chapter 4 proposes a numerical method to compute approximate eigenvalues and factored eigenvectors for the whole DBN (or any system whose transition model is presented in some factored form). While the abstraction in Chapter 3 depended upon varying levels of near-determinism in the evolution of individual variables, this approach becomes applicable to linear combinations of the variables and hence is strictly more powerful. The dominant eigenpairs (i.e., eigenvalues and their corresponding eigenvectors) can be used to perform super-fast simulation.

Next, Chapter 5 focuses on the maximization problem and proposes a modified Viterbi algorithm with spatial and temporal abstractions. The primary insight behind speeding up the maximization process is to prune away a large portion of the search space which has no chance of being the solution. In order to do this pruning, we have to create the appropriate abstractions of the state space and also define customized refinement operations.

After presenting a general maximization algorithm in Chapter 5, the next chapter describes an application of that idea to the problem of video segmentation. We design a hierarchical video segmentation that starts from a very abstract problem (very coarse/large supervoxels) and iteratively refines those parts of the video which are likely to contain more than one object category. The algorithm trivially extends to image segmentation.

Markov chain Monte Carlo (MCMC) algorithms are investigated in Chapter 7. These sampling algorithms are used ubiquitously to numerically solve marginalization problems which are analytically intractable. In the face of near-determinism , MCMC algorithms fare very poorly. In fact, their performance worsens as the degree of near-determinism in the problem increases. However, for many problems, there exist efficient algorithms which can sample from the solution set of the corresponding deterministic problem and are much more efficient than an MCMC algorithm in the near-deterministic domain. We propose a methodology to design MCMC algorithms for near-determinism systems which are guided by the insights behind these deterministic domain

algorithms and are shown to significantly outperform generic MCMC algorithms and make for a smooth performance curve in the determinism continuum.

Finally, Chapter 8 concludes the dissertation by identifying a few potential directions of research to further exploit such skewed structures in graphical models. We also summarize our contributions by highlighting both the benefits and limitations of the various algorithms presented in this dissertation.

Chapter 2

Background

In this chapter, we review concepts and literature related to the material covered in this dissertation. Additional references are also provided for readers interested in exploring the material in detail.

2.1 Graphical model

Probabilistic graphical models are graphs where the nodes represent random variables and the edges represent conditional dependence relations. The variables can be discrete, continuous or hybrid. The graphical structure provides a concise description of the joint probability of a system, and several algorithms have been designed to exploit the conditional independence relations for specific graph structures (e.g., chains, trees). There are broadly two families of graphical models — undirected (Markov random fields and factor graphs are two popular examples) and directed (these are also called Bayesian networks).

Random variables are denoted by capital letters (e.g., X and Y). The values a random variable can take are denoted by small letters (e.g., $X = x_1$ or $X = x_2$). The probability of $X = x$ conditioned on $Y = y$ is denoted by $p(X = x|Y = y)$, or (when the context is clear) by $p(x|y)$. The conditional distribution of X given Y is denoted by $p(X|Y)$. The independence between X and Y is denoted by $X \perp\!\!\!\perp Y$ while the independence between X and Y conditioned on Z (i.e., the conditional independence) is denoted by $X \perp\!\!\!\perp Y|Z$. A group of variables is denoted by a block capital letter (e.g., $\mathbf{X}$ or $\mathbf{Y}$). The parents of a node (or variable) X is denoted by $\pi(X)$.

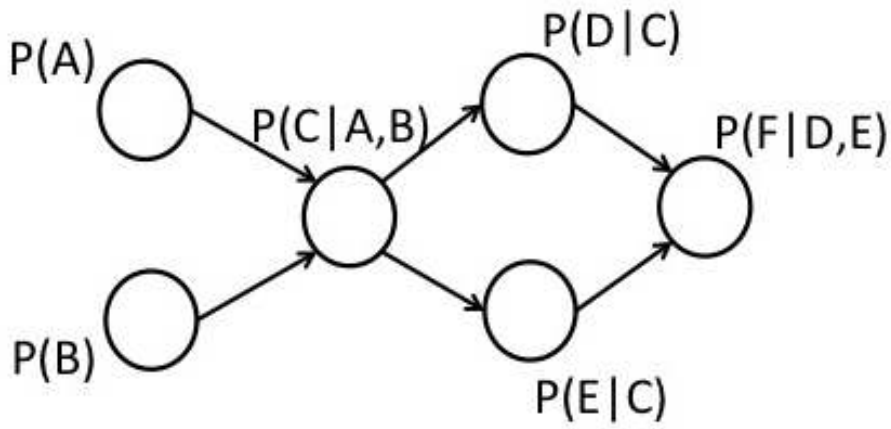


Figure 2.1: An example of a directed graphical model or Bayesian network. There are 6 random variables $\{A, B, C, D, E, F\}$. The edges denote conditional dependence relations, and the tables are conditional probability tables.

Directed graphical models

The joint probability of a directed graphical model with variables $\{X_1, \cdots, X_n\}$ is given by:

$$p(X_1, \cdots, X_n) = \prod_{i=1}^{n} p(X_i|\pi(X_i))$$

This is also called the chain rule of probability.

Hence, in order to completely specify a directed probabilistic graphical model, we need to specify not only the graphical structure, but also the parameters of each conditional probability distribution (namely the $p(X_i|\pi(X_i))$). If the variables are discrete, then this is specified in a conditional probability table (CPT) and if they are continuous, then a conditional probability density (CPD) is used. In Figure 2.1, the CPTs for each variable are placed next to the corresponding node. In this example, the joint probability is given by:

$$p(A, B, C, D, E, F) = p(A)p(B)p(C|A, B)p(D|C)p(E|C)p(F|D, E)$$

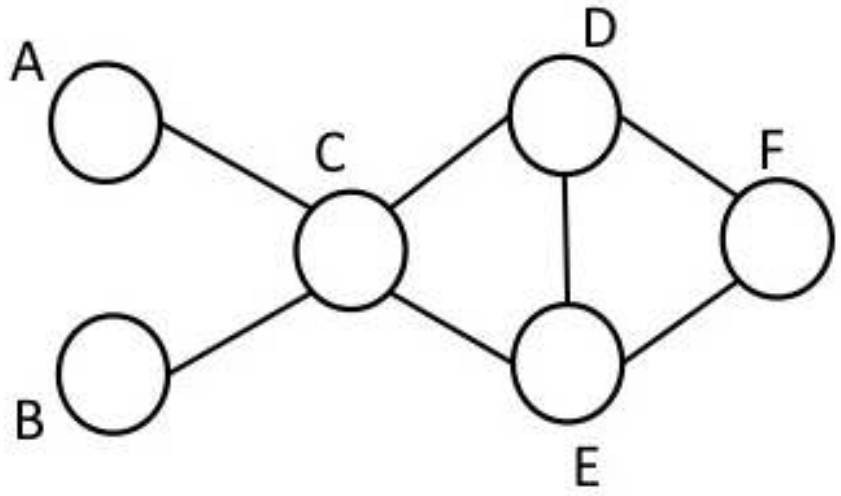


Figure 2.2: An example of an undirected graphical model or Markov random field. There are 6 random variables $\{A, B, C, D, E, F\}$. The joint probability is a normalized product of the individual potential functions.

Undirected graphical models

The joint probability of variables in an undirected graphical model is defined to be a product of the potential of the cliques in the graph. More specifically, if we consider the graphical model in Figure 2.2, the joint probability of the system is given by:

$$\phi(A, B, C, D, E, F) = \phi(A, C)\phi(B, C)\phi(C, D, E)\phi(D, E, F)$$

where $\phi(A, B)$ denotes the potential of each configuration of the variable set $\{A, B\}$. Unlike the conditional probability distributions in directed graphical model, the potential functions need not be normalized. Hence, the joint probability is the normalized product of potentials. Computing the normalization constant is generally computationally expensive, but is also avoidable for several inference problems.

Conditional independence

In a directed graphical model, the conditional independence relations between two (sets of) variables, conditioned on a third, can be algorithmically determined by the d-separation algorithm (also called the Bayes Ball algorithm) (Pearl, 1988). In an undirected model, conditional independence is equivalent to non-connectivity, i.e., $A \perp\!\!\!\perp B|C$ is true if by removing C, A and B become disconnected.

The Markov blanket of a variable X_i is the minimal set of variables, conditioned on which X_i becomes independent of every other variable. More formally, let $MV(X_i)$ denote the Markov blanket of X_i, then

$$\forall X_i \in \mathbf{X}, \quad X_i \perp\!\!\!\perp X_j | MV(X_i)$$

for any $X_j \in \mathbf{X} \setminus MV(X_i)$.

The important point to note here is that in the absence of conditional independence relations, the number of parameters needed to specify the joint probability distribution is *exponential* in the number of variables. In case of a graphical model where the maximum in-degree of a node is a constant, the number of parameters needed is *linear* in the number of variables.

2.2 Inference algorithms

In order to answer any query of the form $p(\mathbf{X}_H|\mathbf{X}_V)$, where H and V are disjoint sets of indices representing the query variables (which are generally a subset of the hidden or unobserved variables) and evidence (or observed) variables respectively. Inference, or computing the conditional probability distribution $p(X_H|X_V)$ comprises of computing the two marginals $p(\mathbf{X}_H|\mathbf{X}_V)$ and $p(\mathbf{X}_V)$ since.

$$p(\mathbf{X}_H|\mathbf{X}_V) = \frac{p(\mathbf{X}_H, \mathbf{X}_V)}{\sum_{\mathbf{x}_H} p(\mathbf{X}_H, \mathbf{X}_V)}$$

In a probabilistic system with n binary variables, the total number of possible states is $O(2^n)$. Computing one of these marginals requires exponential (in n) amount of time in the worst-case. However, there are important special cases where the computational complexity of inference is much lower.

Variable elimination

During marginalization, the order in which the variables are summed out (or "eliminated") can significantly affect the computation time. The variable elimination algorithm uses a greedy heuristic to determine the order by pushing each summation operator as far to the right in the expression as possible. In the example in Figure 2.1, if we want to compute $p(A|E)$, then we need to compute $p(A, E)$ and $p(E)$. The greedy heuristic elimination ordering to compute $p(A, E)$ would be F followed by D followed by C and finally B.

$$
\begin{aligned}
p(A,E) &= \sum_b \sum_c \sum_d \sum_f p(A)p(B)p(C|A,B)p(D|C)p(E|C)p(F|D,E) \\
&= \sum_b \sum_c \sum_d p(A)p(B)p(C|A,B)p(D|C)p(E|C) \sum_f p(F|D,E) \\
&= \sum_b \sum_c p(A)p(B)p(C|A,B)p(E|C) \sum_d p(D|C)c_f(D,E) \\
&= \sum_b p(A)p(B) \sum_c p(C|A,B)p(E|C)c_{f,d}(C,E) \\
&= p(A) \sum bp(B)c_{c,d,f}(A,B,E) \\
&= p(A)c_{b,c,d,f}(A,E)
\end{aligned}
$$

Finding the optimal variable elimination ordering is known to be an NP-hard problem. Greedy heuristics, however, often provide good solutions. $p(E)$ can be computed in a similar fashion to complete the inference process.

Belief propagation

When we wish to compute the marginal probabilities of all (or several) variables in a system, then one (expensive) alternative is to use variable elimination for each of those variables. However, this approach is very wasteful, since several computations from one variable elimination can be reused in the next iterations. A more general solution is to use dynamic programming to avoid redundant computations.

For any singly-connected Bayesian network, Pearl's polytree algorithm (Pearl, 1988) works for any singly-connected Bayesian network. Let the parents of a node X be the set of nodes $\{U_1, \cdots, U_M\}$ and its children be the set $\{Y_1, \cdots, Y_N\}$. Let the evidence variables be denoted by E, with the evidence in the ancestor nodes denoted by e^+ and evidence in the descendant nodes denoted by e^-.

Messages can be sent "down" the network (from parent to child) or "up" the network (from child to parent). Let us label these messages π_{U_i} and λ_{Y_j}. We wish to compute the marginal probability of X, denoted by $BEL(X)$, conditioned on the evidence, i.e., $p(X|E=e)$, which is given by

$$BEL(X) = p(X|e) = \alpha\lambda(X)\pi(X)$$

where α is a normalizing constant, $\lambda(X) = p(e^-|X)$ and $\pi(X) = p(X|e^+)$. These are computed as follows:

$$\lambda(X) = \prod_{i=1}^{N} \lambda_{Y_i} \pi(X) = \sum_{\vec{u}} p(X|\vec{u})\pi(\vec{u})$$

where $\vec{u}$ is a vector of possible values for $\{U_1, \cdots, U_M\}$.

Once $\lambda(X)$ and $\pi(X)$ are computed and $BEL(X)$ updated, X sends messages out to its parents and children using all the information from every node other than the one it is sending the message to. Thus, the message send to U_i and Y_i are as follows:

$$\pi(Y_i) = \frac{BEL(X)}{\lambda_{Y_i}} \lambda(U_i) = \beta \sum_{X} \lambda(X) \sum_{u_k : k \neq i} p(X|\vec{u}) \prod_{k \neq i} \pi(u_k)$$

where β is a normalizing factor. The overall algorithm runs as follows:

1. Initialize the network.
2. Update beliefs.
3. Propagate changes in belief
4. If beliefs change, then go to step 2, else terminate.

The most general version of such a message passing algorithm is the junction tree algorithm, which requires a transformation of the directed graph to an undirected one.

Junction tree algorithm

For a multiply-connected Bayesian network, belief propagation is not guaranteed to converge to the exact marginals. In order to compute the exact marginals, we need to convert the Bayes net to an equivalent singly-connected undirected graphical model—the junction tree—and then run a similar message passing protocol.

In order to convert a directed graphical model to an undirected one, we need to convert local conditional probabilities into potentials. In our running example, $p(C|A, B)$ is essentially a function of three variables. The only obstacle to using this function as a potential is that $\{A, B, C\}$ is not a clique in the directed model. Or, in general, a node and its parents are generally not in the same clique (if we just drop the directionality of the edges in a Bayes net). Thus, we need to "marry" the two parents (i.e., introduce an edge between every pair of nodes which share a common child). This process is called **m**$oralization$ as shown in Figure 2.3.

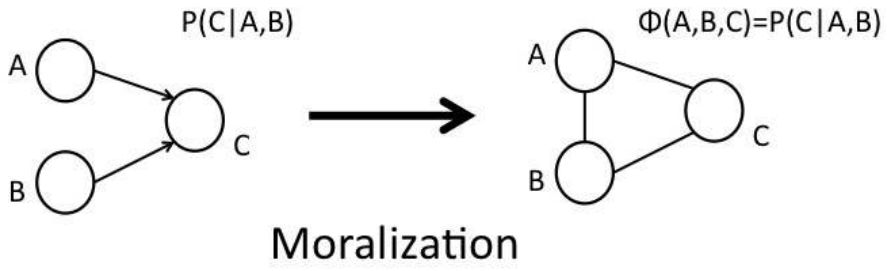


Figure 2.3: The moralization process of introducing an edge between every pair of non-connected parents of a node.

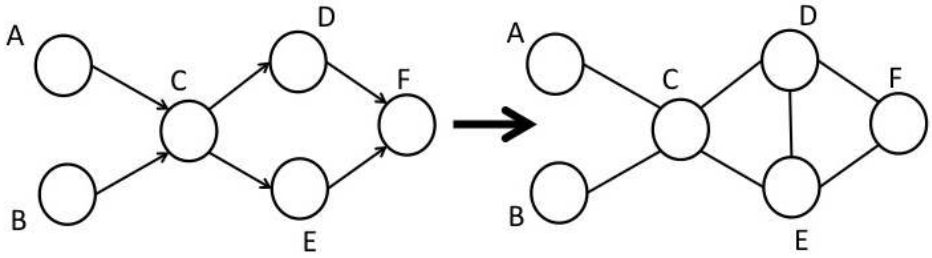


Figure 2.4: The moralization transformation on our example.

The potential on a clique is defined as the product over all the conditional probability distributions *contained* within the clique. The product of potentials defined in this manner results in the same joint probability as in the directed graphical model.

$$\begin{aligned} p(A,B,C,D,E,F) &= p(A)p(B)p(C|A,B)p(D|C)p(E|C)p(F|D,E) \\ &= \phi(A,B,C)\phi(C,D,E)\phi(D,E,F) \end{aligned}$$

where

$$\begin{aligned} \phi(A,B,C) &= p(A)p(B)p(C|A,B) \\ \phi(C,D,E) &= p(D|C)p(E|C) \\ \phi(D,E,F) &= p(F|D,E) \end{aligned}$$

Clique tree A *clique tree* is an undirected tree of cliques. The clique tree corresponding to the moralized graph in Figure 2.4, is shown in Figure 2.5.

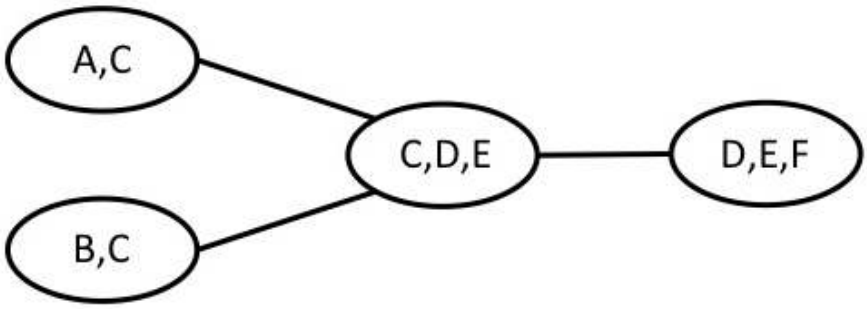


Figure 2.5: The clique tree corresponding to the moralized graph.

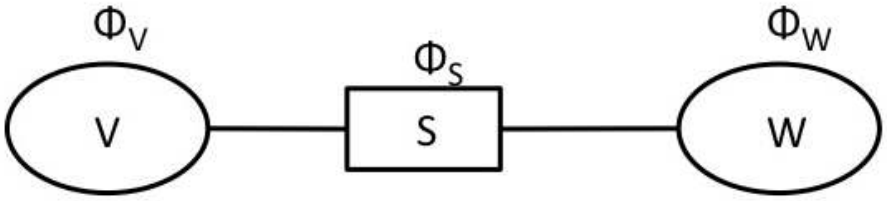


Figure 2.6: The clique tree corresponding to the moralized graph.

Neighboring cliques need to be consistent on the probability of nodes in their intersection. For instance, the neighboring cliques $\{A, C\}$ and $\{C, D, E\}$ have an overlap of $\{C\}$. A general instance is shown in Figure 2.6, where V and W denotes the two neighboring cliques and S is their intersecting set of nodes. The consistency of the marginal distribution of the intersecting set of variables is ensured by first marginalization and then re-scaling. The two steps together constitute the update step.

$$\begin{aligned} \phi^*(S) &= \textstyle\sum_{V \setminus S} \phi(V) \quad \text{Marginalization} \\ \phi^*(W) &= \phi(W)\frac{\phi^*(S)}{\phi(S)} \quad \text{Rescaling} \end{aligned}$$

A local message passing algorithm can ensure global consistency **i**ff non-neighboring clique nodes always have a null intersection set. However, moralization does not ensure this (an example is shown in Figure 2.8). Local consistency entails global consistency **i**ff the *running intersection property* is satisfied, which states that if a node appears in two cliques, then it appears in every clique on the path between those two cliques. The running intersection property is also called the *junction tree property*.

A *triangulated graph* is one in which no cycles of four or more nodes exist without a chord. We perform triangulation on the moralized graph by adding chords to every cycle of four or more

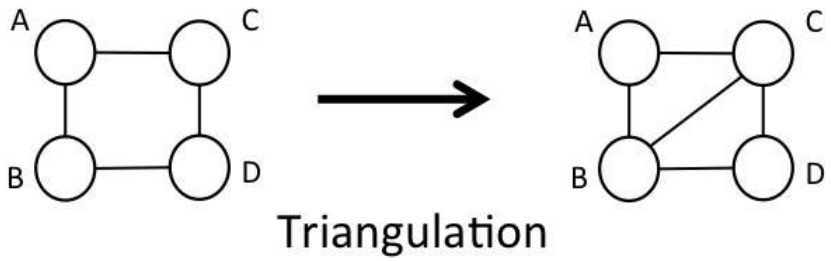


Figure 2.7: The triangulation process of introducing chords in cycles of length 4 or greater to ensure maximal clique size of 3.

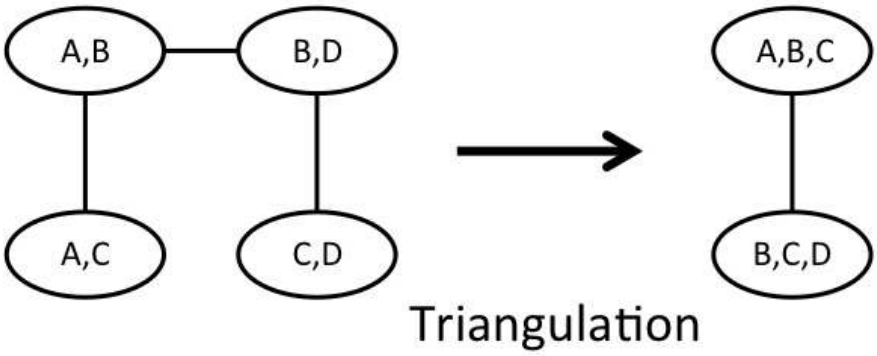


Figure 2.8: The clique tree before and after the triangulation process. The one after satisfies the running intersection property.

nodes. Triangulation ensures that the clique tree corresponding to the triangulated graph satisfies the running intersection property.

Message passing protocol After moralization and triangulation, we can perform local message passing updates (as defined previously) to perform inference. These updates need to be propagated in a particular order. Root the clique tree (obtained after triangulation) in an arbitrary node. Then, first send messages from the leaves towards the root and then from the root back towards the leaves. At the end of the two phases, all the cliques will have consistent non-normalized potentials (conditioned on the evidence variables). This algorithm is called the *Hugin algorithm*.

The moralization and triangulation phases can be done offline for a graphical model. The message passing phase is online (depending on the evidence and query variables).

Approximate inference

Variational inference The general idea is to formulate the inference problem as an optimization problem and then "relax" the problem in different ways which include approximating the function to be optimized or approximating the set over which the optimization is performed. These relaxations in turn approximate the quantity of interest. Popular variational inference techniques include mean field approximation and variational Bayes (Attias, 1999; Xing *et al.*, 2002; Wainwright & Jordan, 2008).

Sampling methods We can approximate query distributions with a set of samples using different sampling techniques like importance sampling, sequential Monte Carlo and Markov chain Monte Carlo (MCMC). Most sampling methods asymptotically converge to the correct answer in the limit of infinite samples. Although there are several computationally inexpensive sampling schemes, they do not converge quickly enough in high-dimensional problems. MCMC algorithms, which we will cover in more detail in Section 2.3, are well-suited for high-dimensional problems.

Loopy belief propagation The belief propagation algorithm for singly-connected Bayesian network (Pearl, 1988), as described previously, can also be applied to a graphical model with cycles (or which is multiply connected). In this case, the belief propagation process may not necessarily converge, and even if it does, it need not converge to the exact distribution, but the approximation quality is often found to be very good (Murphy *et al.*, 1999).

Hidden Markov model (HMM)

A hidden Markov model (HMM) is a doubly stochastic process with one underlying *discrete-valued* stochastic process that is not observable (hence hidden or latent), but can only be observed through another stochastic process which produces a sequence of observations. A popular application is speech processing, where the underlying latent stochastic process corresponds to the sentence or phrase being expressed, while the observations correspond to the acoustic signals produced in the process. For a detailed introduction to hidden Markov models, please refer to (Rabiner & Juang, 1986; Rabiner, 1989).

Consider the instance of an HMM shown in Figure 2.9, whose latent Markovian state X is in one of N discrete states $\{1, 2, \ldots, N\}$. Let the state variable at time t be denoted by X_t. The transition matrix $A = \{a_{ij} : i, j = 1, 2, \ldots, N\}$ defines the state transition probabilities where $a_{ij} = p(X_{t+1} = j \mid X_t = i)$. The Markov chain is assumed to be stationary, so a_{ij} is independent of t. Let the discrete observation space be the set $\{1, 2, \ldots, M\}$. Let Y_t be the observation symbol at time t. The observation matrix $B = \{b_{ik} : i = 1, 2, \ldots, N; k = 1, 2, \ldots, M\}$ defines the emission probabilities where $b_{ik} = p(Y_t = k \mid X_t = i)$. The definition extends naturally to continuous observations. The initial state distribution is given by $\Pi = \{\pi_1, \ldots, \pi_N\}$ where $\pi_i = p(X_0 = i)$.

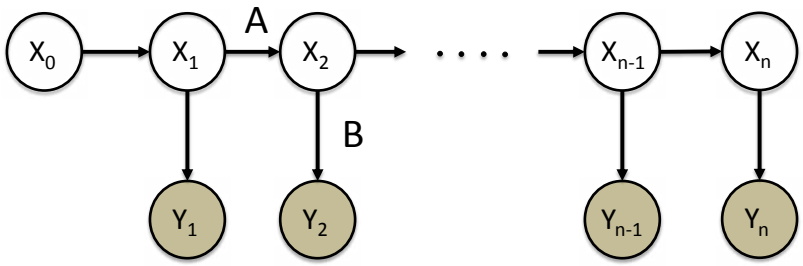


Figure 2.9: The hidden Markov model (HMM). X_t is the latent (or unobserved) variable at time-step t, and its transition dynamics are Markovian. Y_t is the observed variable at time t, and is conditionally independent of all other variables given X_t.

The Viterbi algorithm

The Viterbi algorithm (Viterbi, 1967; Forney, 1973) finds the most likely sequence of hidden states, called the "Viterbi path," conditioned on a sequence of observations in a hidden Markov model (HMM). If the HMM has N states and the sequence is of length T, there are N^T possible state sequences, but, because it uses dynamic programming (DP), the Viterbi algorithm's time complexity is just $O(N^2T)$. It is one of the most important and basic algorithms in the entire field of information technology; its original application was in signal decoding but has since been used in numerous other applications including speech recognition (Rabiner, 1989), language parsing (Klein & Manning, 2003), and bioinformatics (Lytynoja & Milinkovitch, 2003).

Following Rabiner (1989), we define

$$\delta_t(i) = \max_{X_{0:t-1}} p(X_{0:t-1}, X_t = i, Y_{1:t} \mid A, B, \Pi) \,,$$

i.e., the likelihood score of the optimal (most likely) sequence of hidden states (ending in state i) and the first t observations. By induction on t, we have:

$$\delta_{t+1}(j) = [\max_i \delta_t(i) a_{ij}] b_{jY_{t+1}} \,.$$

The actual state sequence is retrieved by tracking the transitions that maximize the $\delta(.)$ scores for each t and j. This is done via an array of back pointers $\psi_t(j)$. The complete procedure (Rabiner, 1989) is as follows:

1. Initialization $\begin{array}{rcl}\delta_1(i) & = & \pi_i\, b_{iY_1}, \quad 1 \leq i \leq N \\ \psi_1(i) & = & 0\end{array}$

2. Recursion

$$\begin{array}{rl}\delta_t(j) = & \max_{1\leq i\leq N}[\delta_{t-1}(i)a_{ij}]b_{jY_t}, \quad 2 \leq t \leq T \\ \psi_t(j) = & \operatorname{argmax}_{1\leq i\leq N}[\delta_{t-1}(i)a_{ij}], \quad 2 \leq t \leq T\end{array}$$

3. Termination

$$\begin{array}{rcl}P^* & = & \max\limits_{1\leq i\leq N}[\delta_T(i)] \\ X_T^* & = & \operatorname{argmax}\limits_{1\leq i\leq N}[\delta_T(i)]\end{array}$$

4. Path backtracking

$$X_t^* \;=\; \psi_{t+1}(X_{t+1}^*), \quad t = T-1, T-2, \ldots, 1$$

The time complexity of this algorithm is $O(N^2T)$ and the space complexity is $O(N^2 + NT)$.

Dynamic Bayesian network (DBN)

A dynamic Bayesian network (DBN) (Dean & Kanazawa, 1989) is a discrete-time model of a stochastic dynamical system. The system's state is represented by a set of variables, $\mathbf{X}_t$ for each time $t \in \mathbb{Z}^*$ and the DBN represents the joint distribution over the variables $\bigcup_{t=0}^{\infty} \mathbf{X}_t$. Typically we assume that the system's dynamics do not change over time, so the joint distribution is captured by a 2-TBN (2-Timeslice Bayesian Network), which is a compact graphical representation of the state prior $P(\mathbf{X}_0)$ and the stochastic dynamics $P(\mathbf{X}_{t+1}|\mathbf{X}_t)$. In turn, the dynamics are represented in factored form via a collection of local conditional models $P(X_{t+1}^i|\pi(X_{t+1}^i))$, where $\pi(X_{t+1}^i)$ are the parent variables of X_{t+1}^i in slice t or $t+1$. Henceforth, we will consider all X_t^i to be discrete.

An instance of a DBN is shown in Figure 2.10. Since their introduction, DBNs have proved to be a flexible and effective tool for representing and reasoning about stochastic systems that evolve over time. DBNs include as special cases hidden Markov models (HMMs) (Baum & Petrie, 1966), factorial HMMs (Ghahramani & Jordan, 1997), hierarchical HMMs (Fine *et al.*, 1998), discrete-time Kalman filters (Kalman, 1960), and several other families of discrete-time models. For a detailed survey of inference algorithms for DBNs, please refer to (Srkk, 2013).

In this dissertation, we will mainly focus on directed graphical models, but the algorithms and ideas presented are equally applicable to undirected models.

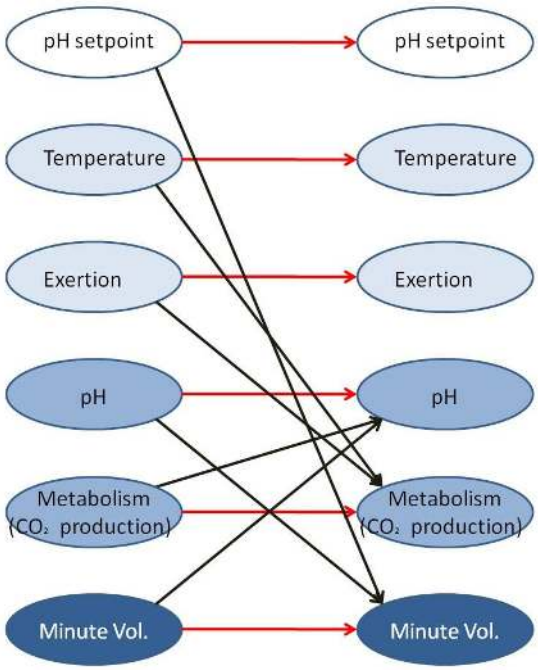


Figure 2.10: A sample dynamic Bayesian network (DBN) model of the blood acidity regulation mechanism in humans.

2.3 Markov chain Monte Carlo

Markov chain Monte Carlo (MCMC) methods are a family of algorithms for sampling from probability distributions, which are difficult to sample from directly. The methods construct a Markov chain that has the desired distribution as its equilibrium distribution. The state of the chain gradually approaches the equilibrium distribution, and can then be used as samples.

While it is easy to design an MCMC algorithm for any desired distribution, the critical aspect is to ensure that the equilibrium distribution is reached quickly. The number of steps taken to reach the equilibrium distribution, starting from any arbitrary state, is called the *mixing time*. It should also be noted that an MCMC algorithm can only approximate the desired distribution, as there is always some residual effect of the starting state. Most practitioners discard a number of samples – the *burn-in* – at the start (this number depends on the mixing time). Successive samples are often highly correlated and there are several application-specific methods to counter that.

The most common application for MCMC algorithms is numerically calculating multi-dimensional integrals. Each sample generated by the Markov chain is used to generate the integrand value and that counts towards the integral. Let $\mathbf{x}^{(1)}, \cdots, \mathbf{x}^{(k)}$ be k samples generated using an MCMC algorithm (after accounting for the burn-in) and let $\pi(.)$ be the desired distribution. Then we can

approximate the expected value of a function $f(.)$ under the distribution $\pi(.)$ by:

$$\mathbb{E}_{\pi}(f) = \frac{1}{k}\sum_{i=1}^{k} f(\mathbf{x}^{(i)}) \tag{2.1}$$

We will now briefly review some popular MCMC algorithms, which we use later in this dissertation. To read more about MCMC algorithms, please refer to (Robert & Casella, 2005) and (Andrieu *et al.*, 2003).

Metropolis Hastings algorithm

The Metropolis-Hastings (hereafter MH) algorithm (Hastings, 1970) is possibly the most popular MCMC algorithm. Let us say we want to generate samples from the probability distribution $\pi(\mathbf{x})$ (hereafter, also referred to as the target distribution). Let $q(.|\mathbf{x})$ denote the proposal distribution given the current state $\mathbf{x}$. The MH algorithm states that a newly proposed state $\mathbf{x}'$ using $q(.|\mathbf{x})$ will be accepted with probability:

$$\alpha(\mathbf{x}, \mathbf{x}') = \min\left(1, \frac{\pi(\mathbf{x}')q(\mathbf{x}|\mathbf{x}')}{\pi(\mathbf{x})q(\mathbf{x}'|\mathbf{x})}\right) \tag{2.2}$$

If the proposed state $\mathbf{x}'$ is accepted, then it becomes the next sample, else $\mathbf{x}$ is replicated as the next sample. It can be shown that the Markov chain thus constructed has $\pi(.)$ as its invariant distribution, if the support of $q(.|.)$ includes the support of $\pi(.)$ (Tierney, 1994).

Gibbs sampling

Gibbs sampling (Geman & Geman, 1984) is another very popular MCMC algorithm which, in its very basic form, is a special case of the MH algorithm. Named after the physicist Josiah Willard Gibbs, the point of Gibbs sampling is that it is easier to sample from a conditional distribution than to sample from a joint distribution. Let us focus on a problem of generating k samples of $\mathbf{x} = \{x_1, x_2, \cdots, x_n\}$ from a joint distribution $p(x_1, x_2, \cdots, x_n)$. The i^{th} sample is denoted by $\mathbf{x}^{(i)}$.

1. Start with some initial value $\mathbf{x}^{(0)}$
2. For each sample $i = \{1, 2, \cdots, k\}$, sample each variable ${x^{(i)}}_j$ from the conditional distribution $p(x_j|{x^{(i)}}_1, \cdots, {x^{(i)}}_{j-1}, {x^{(i-1)}}_{j+1}, \cdots, {x^{(i-1)}}_n)$

In each iteration, we sample each variable conditioned on the current value of all the other variables. The samples then approximate the joint distribution. One of the great things about the

Gibbs sampler is that it does not require any parameter tuning and hence is often the first MCMC algorithm that practitioners resort to.

Important variations of the Gibbs sampler include the *blocked Gibbs* sampler (Jensen *et al.*, 1995) – where instead of sampling a single variable, we sample a *block* of variables conditioned on all the other variables – and the *collapsed Gibbs* sampler (Liu, 1994) – where a subset of variables are marginalized out when sampling for some other variable.

2.4 A^* algorithm

A^* is an algorithm that is widely used in graph search problems. It is a best-first search algorithm that finds a minimum-cost path from an initial node to a goal node (which could be one among many goal nodes). As the algorithm proceeds, it explores the optimal path given the currently available information while maintaining a priority queue of the alternate paths.

The A^* algorithm uses a knowledge-plus-heuristic cost function of a node x. This overall cost is generally denoted by $f(x)$. The cost function is a sum of two functions:

1. The path cost to reach x from the initial node. This is an exact cost and is denoted by $g(x)$.
2. The estimated cost to reach a goal state from x, denoted by $h(x)$. This is an estimate and is called an *admissible heuristic* if it is a lower bound of the actual cost.

Thus, $f(x) = g(x) + h(x)$. It is critical that $h(x)$ is an admissible heuristic, since the algorithm explores the node x^* with the lowest $f(.)$ value and terminates when it picks a goal state to explore. If the heuristic is not admissible, then it might be possible to have a shorter path to a goal state remaining unexplored, upon termination, which would in turn affect the correctness of the algorithm. A good heuristic function is very application-specific and often requires extensive research and domain expertise. A simple example of a heuristic function in a shortest route-finding problem is the straight line distance between two cities. The full details of the algorithm are presented in Algorithm 2.1, where $e[x, y]$ is a function that returns the edge distance between neighboring nodes x and y while $h(x, y)$ is a function that returns an admissible heuristic distance between nodes x and y. The $Return\_Shortest\_Path$ method is a simple backtracking procedure which can return the optimal path to the $goal$ state. For more details about some of the subtleties, please refer to (Russell & Norvig, 2010).

Application to MAP inference

The A^* search algorithm can be used in other kinds of search problems too – one such application is maximum a posteriori (MAP) inference. Given a joint probability distribution $Pr(\mathbf{x}, \mathbf{y})$, we

Algorithm 2.1 A^*(start, goal)

```
1: closedset ← ∅
2: openset ← {start}
3: previous ← empty map
4: g[start] ← 0
5: f[start] ← g[start] + h(start, goal)
6: while openset ≠ ∅ do
7:    current ← argmin_{x∈openset} f[x]
8:    if current = goal then
9:       Return_Shortest_Path(previous, goal)
10:   end if
11:   openset ← openset \ current
12:   closedset ← closedset ∪ current
13:   for all neighbor of current do
14:      temp ← g[current] + e[current, neighbor]
15:      if neighbor ∈ closedset and temp ≥ g[neighbor] then
16:         continue
17:      end if
18:      if neighbor ∉ openset or temp < g[neighbor] then
19:         previous[neighbor] ← current
20:         g[neighbor] ← temp
21:         f[neighbor] ← g[neighbor] + h(neighbor, goal)
22:         if neighbor ∉ openset then
23:            openset ← openset ∪ neighbor
24:         end if
25:      end if
26:   end for
27: end while
28: No path found.
```

wish to find $\text{argmax}_{\mathbf{x}} Pr(\mathbf{x} \mid \mathbf{y})$. We can start off with an abstract model for $Pr(.)$, say $q(.)$, where $\forall \mathbf{x}, q(. \mid \mathbf{y}) \leq Pr(. \mid \mathbf{y})$. Thus, $q(. \mid \mathbf{y})$ is an admissible heuristic function for $Pr(. \mid \mathbf{y})$.

As we shall see in Chapters 5 and 6, we can use a succession of abstract models ending in the actual probability distribution. Each abstract model is an admissible heuristic of the next abstract model and this set of models can be used to define a hierarchical MAP inference algorithm.

2.5 Matrices and vectors

In this section, we will review some concepts and algorithms related to matrices. For more details about any of these topics, please refer to (Golub & Van Loan, 2012).

Definitions

A matrix is a rectangular array of numbers, symbols or expressions, arranged in rows and columns. In this dissertation, the matrix elements will be either numbers or symbols. Let A be a matrix with m rows and n columns – an $m \times n$ matrix. $a_{i,j}$ denotes the element of the matrix at the i^{th} row and j^{th} column.

$$\mathbf{A} = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & & \ddots & \vdots \\ a_{m,1} & \cdots & a_{m,n-1} & a_{m,n} \end{bmatrix} \tag{2.3}$$

A vector is a special matrix which has only 1 column. For example, $\mathbf{u}$ is a vector with n elements

$$\mathbf{u} = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{bmatrix} \tag{2.4}$$

A *square* matrix is one where the number of rows and columns are equal.

The *diagonal* of a square matrix is the set of elements whose row and column indices are identical (i.e., $a_{i,i}$ for $1 \leq i \leq n$, in an $n \times n$ matrix).

A *diagonal matrix* is a square matrix whose all non-diagonal entries are zeros.

An *identity matrix* is a diagonal matrix whose diagonal elements are all 1. Any square matrix A multiplied with the identity matrix (of the same dimensions) results in A.

The *transpose* of an $m \times n$ matrix A is an $n \times m$ matrix B, where $b_{i,j} = a_{j,i}$ for $1 \leq i \leq n, 1 \leq j \leq m$. The transpose of A is generally denoted by A^T.

The *conjugate transpose* of a matrix A, denoted by A^*, is the transpose of A with each entry replaced by its complex conjugate entry.

The *inverse* of an $n \times n$ square matrix A, denoted by A^{-1}, is the matrix which satisfies

$$AA^{-1} = I$$

where I is the $n \times n$ identity matrix.

A matrix is said to be *invertible* if its inverse exists.

For more information about standard matrix operations and their properties, please refer to (Golub & Van Loan, 2012).

Special matrices

Triangular matrix

A *triangular matrix* is a square matrix where all the entries above or below the diagonal are zero. If all the entries above the diagonal are zero (i.e., $a_{i,j} = 0$ for $i < j$), then it is an *lower* triangular matrix, while it is an *upper* triangular matrix if all the entries below the diagonal are zero (i.e., $a_{i,j} = 0$ for $i > j$). An example of a 4×4 upper triangular matrix is

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & -1 & 0 \\ 0 & 8 & -3 & 2 \\ 0 & 0 & 4 & -1 \\ 0 & 0 & 0 & 6 \end{bmatrix}$$

Hessenberg matrix

A *Hessenberg matrix* is a square matrix that is "almost triangular". If all the entries above the first subdiagonal are zeros (i.e., $a_{i,j} = 0$ for $i < j - 1$), then it is an *lower* Hessenberg matrix, while it is an *upper* Hessenberg matrix if all the entries below the first subdiagonal are zero (i.e., $a_{i,j} = 0$ for $i - 1 > j$). An example of a 4×4 upper Hessenberg matrix is

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & -1 & 0 \\ 6 & 8 & -3 & 2 \\ 0 & -3 & 4 & -1 \\ 0 & 0 & 1 & 6 \end{bmatrix}$$

Similar matrices

Two matrices A and B are said to be *similar* if

$$A = P^{-1}BP \tag{2.5}$$

where P is an $n \times n$ invertible matrix.

Unitary matrix

A square matrix is *unitary*, if its conjugate transpose is also its inverse, i.e., if it satisfies:

$$AA^* = A^*A = I$$

Orthogonal matrix

A square matrix with real entries is *orthogonal*, if its transpose is also its inverse, i.e., if it satisfies:

$$AA^T = A^T A = I$$

Orthonormal vectors

Two non-zero vectors $\mathbf{x}$ and $\mathbf{y}$ are said to be *orthonormal* if each have a norm of 1 and satisfy $\mathbf{x}^T\mathbf{y} = 0$. If they only satisfy the latter criterion, then they are called *orthogonal* vectors.

A set of vectors is said to be orthonormal if each pair is orthonormal.

Kronecker product

The *Kronecker product*, denoted by $\otimes$, is an operation on two matrices. If A is an $m \times n$ matrix and B is a $p \times q$ matrix, then $A \otimes B$ is an $mp \times nq$ matrix of the form:

$$A \otimes B = \begin{bmatrix} a_{1,1}B & a_{1,2}B & \cdots & a_{1,n}B \\ a_{2,1}B & a_{2,2}B & \cdots & a_{2,n}B \\ \vdots & \ddots & \ddots & \vdots \\ a_{m,1}B & \cdots & \cdots & a_{m,n}B \end{bmatrix} \tag{2.6}$$

It is a generalization of the *outer product* from vectors to matrices.

2.6 Eigenvalues and eigenvectors

An *eigenvector* of a square matrix A is a non-zero vector $\mathbf{v}$, which when multiplied by A, results in the eigenvector multiplied by a scalar λ

$$A\mathbf{v} = \lambda\mathbf{v} \tag{2.7}$$

The scalar λ is called the *eigenvalue* associated with the eigenvector $\mathbf{v}$. The 2-tuple $(\lambda, \mathbf{v})$ is called an *eigenpair*.

The *characteristic polynomial* for a matrix A is given by:

$$A\mathbf{v} - \lambda\mathbf{v} = 0 \tag{2.8}$$

This is equivalent to

$$(A - \lambda I)\mathbf{v} = 0 \tag{2.9}$$

For an equation of the form $B\mathbf{v} = 0$ to have a non-zero solution $\mathbf{v}$, the determinant of B must be 0. Therefore, the eigenvalues of A are the roots of the equation

$$det(A - \lambda I) = 0 \tag{2.10}$$

We know from the Abel–Rufini theorem that there are no algebraic formulae for roots of a general polynomial of degree greater than 4. As a result, numerical methods are the only way for eigenvalue computation for general matrices.

QR algorithm

Similar matrices have the same eigenvalues. The QR algorithm is an eigenvalue algorithm that is used to calculate the eigenvalues and eigenvectors of a matrix. This is an iterative algorithm, where in each iteration, the current matrix is factored into a product of an orthogonal matrix and an upper triangular matrix (a QR decomposition) and then the factors are multiplied in reverse order.

Let A_k be the matrix in the k^{th} iteration, so $A_0 = A$. At the k^{th} step, we compute the QR decomposition $A_k = Q_k R_k$ where Q_k is an orthogonal matrix and R_k is an upper triangular matrix. We then form $A_{k+1} = R_k Q_k$.

$$A_{k+1} = R_k Q_k = {Q_k}^T Q_k R_k Q_k = {Q_k}^T A_k Q_k = {Q_k}^{-1} A_k Q_k \tag{2.11}$$

Thus, A_{k+1} and A_k are similar matrices and have the same eigenvalues and each iteration is a similarity transformation. Under certain conditions, the matrix A_k converges to a triangular matrix, the *Schur form* of A. The eigenvalues of A are listed on the diagonal of the resultant triangular matrix. It is impractical to iterate till we get perfect zeros in the left bottom half of A_k, but we can bound the error with the *Gershgorin circle theorem* (Golub & Van Loan, 2012).

The cost of *each* iteration, in its current form, is $O(n^3)$. However, it can be sped up using a *Householder reduction* to first bring A to an upper Hessenberg form. This reduction is an $O(n^3)$ operation. After that, each iteration is $O(n^2)$ and it also reduces the number of iterations required. This little detail will not be essential in our later analysis. Suffice to say that computational complexity of the QR decomposition is $O(n^3)$. Upon termination, we have all the n eigenpairs of the matrix A.

Sometimes though,we only need are a few eigenpairs (and not all of them). The QR algorithm is not suitable for such an application since it only terminates upon the formation of the upper

triangular matrix (which contains all the eigenvalues) and hence is too expensive. This resulted in a large body of work focused on finding a few (extreme) eigenvalues and their corresponding eigenvectors.

Also, when A is a sparse matrix (also a very important subclass of applications), the QR algorithm is not suitable since it cannot exploit sparsity. Similarity transformations are not sparsity-preserving and the whole process still costs $O(n^3)$.

Krylov subspaces

The *order-r Krylov subspace* generated by an $n \times n$ matrix A and a vector $\mathbf{x}$ is the linear subspace spanned by the images of $\mathbf{x}$ under the first r powers of A ($A^0 = I$).

$$\mathcal{K}_r = span\{\mathbf{x}, A\mathbf{x}, A^2\mathbf{x}, \cdots, A^{r-1}\mathbf{x}\} \tag{2.12}$$

Modern iterative methods for finding eigenvalues of large sparse matrices utilize the Krylov subspace to avoid large matrix-matrix operations. Computing the different images of $\mathbf{x}$ require only matrix-vector multiplications which can fully exploit any sparsity in A. Starting with $\mathbf{x}$, we first compute $A\mathbf{x}$, then multiply A with the resultant vector to obtain $A^2\mathbf{x}$, and so on. Since the images of $\mathbf{x}$ soon become linearly dependent, all Kyrlov subspace methods require some form of *orthogonalization scheme*. The method of orthogonalization distinguishes one algorithm from another.

All Krylov subspace methods can yield partial results. If we only want k extreme eigenvalues of a matrix, then we can work on a Krylov subspace whose order is $O(k)$. If we assume the sparse matrix A has $O(n)$ non-zero entries, then the computational complexity of computing each basis vector of the Krylov subspace is $O(n^2)$.

Lanczos method

Power iteration

One way to find the largest eigenvalue of a matrix A is the power iteration. Starting with an initial random vector $\mathbf{x}$, this method computes $A\mathbf{x}, A^2\mathbf{x}, A^3\mathbf{x}, \cdots$, with the result being iteratively normalized and stored in $\mathbf{x}$. This sequence converges to the eigenvector corresponding to the largest eigenvalue λ_1.

However, this method does not utilize a lot of the information in the intermediate vectors that are computed, since the power iteration only uses the final vector. If instead, we use an orthogonalized basis of the vectors $\{\mathbf{x}, A\mathbf{x}, A^2\mathbf{x}, \cdots, A^{r-1}\mathbf{x}\}$, this orthogonal basis gives good approxi-

mations of the eigenvectors corresponding to the r largest eigenvalues, just like the final vector in power iteration converges to the eigenvector corresponding to the largest eigenvalue.

Arnoldi iteration

Using Gram-Schmidt orthogonalization in the algorithm mentioned previously is numerically unstable. The Arnoldi iteration is an eigenvalue algorithm which fixes this numerical instability.

The Arnoldi iteration uses a variant of the Gram-Schmidt process to produce a series of orthonormal vectors $\{\mathbf{q}_1, \mathbf{q}_2, \cdots\}$ where $\mathbf{q}_1$ is the random initial vector (previously being referred to as **x**). This set of vectors, called the *Arnoldi vectors* span the Krylov subspace $\mathcal{K}_r$, after the r^{th} iteration. The exact steps are enumerated in Algorithm 2.2.

Algorithm 2.2 Arnoldi iteration(A, $\mathbf{q}_1$)

Start with an arbitrary vector $\mathbf{q}_1$ with norm 1
for i = 2 to r **do**
  $\mathbf{q}_i \leftarrow A\mathbf{q}_{i-1}$
  for j = 1 to i-1 **do**
    $h_{j,i-1} \leftarrow \mathbf{q}_j^* \mathbf{q}_i$
    $\mathbf{q}_i \leftarrow \mathbf{q}_i - h_{j,i-1}\mathbf{q}_j$
  end for
  $h_{i,i-1} \leftarrow ||q_i||$
  $\mathbf{q}_i \leftarrow \frac{\mathbf{q}_i}{h_{i,i-1}}$
end for

The i-loop computes the next orthonormal basis vector while the j-loop subtracts the projections of $\mathbf{q}_i$ in the directions of $\mathbf{q}_1, \cdots, \mathbf{q}_{i-1}$.

After executing the Arnoldi iteration algorithm, we have an $r \times r$ upper Hessenberg matrix formed by the coefficients $h_{i,j}$.

$$\mathbf{H_r} = \begin{bmatrix} h_{1,1} & h_{1,2} & h_{1,3} & \cdots & h_{1,r} \\ h_{2,1} & h_{2,2} & h_{2,3} & \cdots & h_{2,r} \\ 0 & h_{3,2} & h_{3,3} & \cdots & h_{3,r} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & h_{r,r-1} & h_{r,r} \end{bmatrix}$$

The eigenvalues of this coefficient matrix can be easily computed since it is pretty small ($r \ll n$) and is also already in Hessenberg form, which makes it easy to use the QR algorithm. The eigenvalues of H_r, called *Ritz values*, are good approximations of the eignenvalues of A (typically of the largest r values). Let $\mathbf{u}_i$ be an eigenvector of H_r corresponding to the eigenvalue λ_i. The

Ritz eigenvector of A corresponding to λ_i, $\mathbf{v}_i$, can be obtained by transforming the eigenvectors of H using Q_r:

$$\mathbf{v}_i = Q_r \mathbf{u}_i \tag{2.13}$$

where Q_r is the transformation matrix whose column vectors are $\mathbf{q}_1, \cdots, \mathbf{q}_r$.

2.7 Linear systems

A linear system is a mathematical model of a system whose evolution is defined by a linear operator. Specific examples of linear systems discussed previously include hidden Markov models (HMMs) and dynamic Bayesian networks (DBNs). In such systems, the output $\mathbf{u}'$ is related to the input $\mathbf{u}$ by the following relation:

$$\mathbf{u}' = A\mathbf{u} \tag{2.14}$$

where A is an $m \times n$ matrix and the input and output are $n \times 1$ vectors.

In temporal systems, $\mathbf{u}$ corresponds to the belief vector at the current time step t (i.e., $\mathbf{u}_t$) while $\mathbf{u}'$ corresponds to the belief vector at the next time step $t+1$ (i.e., $\mathbf{u}_{t+1}$). Let $\mathbf{x}_t$ denote the state of the system at time t – this is a value between 1 and n. Since the state space is generally identical in subsequent time steps, A is a square matrix. The entry $a_{i,j}$ is the probability of transitioning to state j if the current state is i (i.e., $a_{i,j} = Pr(\mathbf{x}_{t+1} = j | \mathbf{x}_t = i)$). Such a matrix is called a *stochastic matrix*. Each column of a stochastic matrix represents a probability distribution and hence the sum of the elements in each column is 1. Also, all its entries are non-negative.

We will revisit the specific properties of stochastic matrices and belief vectors along with the eigenvalue problem in Chapter 4.

Chapter 3

Why are DBNs sparse?

Real stochastic processes operating in continuous time can be modeled by sets of stochastic differential equations. On the other hand, several popular model families, including hidden Markov models and dynamic Bayesian networks (DBNs), use discrete time steps. This chapter explores methods for converting DBNs with infinitesimal time steps into DBNs with finite time steps, to enable efficient simulation and filtering over long periods. An exact conversion—summing out all intervening time slices between two steps—results in a completely connected DBN, yet nearly all human-constructed DBNs are sparse. We show how this sparsity arises from well-founded approximations resulting from differences among the natural time scales of the near-deterministic or "slow" and the stochastic or "fast" variables in the DBN. We define an automated procedure for constructing an approximate DBN model for any desired time step and prove error bounds for the approximation. We illustrate the method by generating a series of approximations to a simple pH model for the human body, demonstrating speedups of several orders of magnitude compared to the original model.

3.1 Introduction

As explained in detail in Section 2.2, a DBN represents the state of a system by the values of a set of variables in a *time slice*, with connections between slices representing the stochastic evolution of the system. Of particular importance is the fact that DBNs are often *sparse*—each variable in a given slice includes among its parents only a small subset of variables from the preceding slice. Thus, a DBN may require exponentially fewer parameters than an equivalent HMM.

Although there have been some attempts at DBN structure learning (Friedman *et al.*, 1998), for the most part DBNs are built by hand. As with ordinary (non-temporal) Bayesian networks, this is a somewhat opaque process fraught with errors; but for DBNs, there is the additional issue

of *choosing the size of the time step* Δ that separates the time slices. As we will see, the choice of Δ has a dramatic effect on both the computational cost of the model and the proper topology of the DBN. Folk wisdom in the field—borrowed perhaps from standard practice in simulation of differential equations—suggests that Δ needs to be small enough so that the fastest-changing variable in the model has only a small probability of changing its state in time Δ. Unfortunately, in many systems this results in gross inefficiency. For example, the body's pH setpoint changes on a timescale of days or weeks, while breathing rate (which affects pH) changes on a timescale of seconds; hence, a system that models both is forced to perform inference over millions of time steps in order to track the pH setpoint over an extended period. This issue motivated the development of continuous-time Bayes nets (CTBNs) (Nodelman *et al.*, 2002), which avoid committing to any fixed time step. Another approach, appropriate for regular but widely separated observations and for certain restricted classes of models, is to convert a natural small-Δ model into an equivalent model whose Δ matches the observation frequency (Aleks *et al.*, 2009).

The approach we take in this chapter is to think about how one might convert a continuous-time model—a CTBN or a set of stochastic differential equations (SDEs)—into an equivalent, or approximately equivalent, discrete-time DBN for a given Δ. This provides some insight into why DBNs have the structures that they do, and also yields an automatic procedure for choosing time steps and DBN structures, such that simulation over long periods can be both efficient and provably (approximately) accurate.

Let us assume that the system can be modeled exactly by a set of n coupled SDEs that is sparse; we can think of this model as a sparse DBN with a very small time step δ. Now, if we increase the time step to, say, $n\delta$ by summing out $n-1$ intervening steps in the model, the resulting model will be completely connected (unless the original model has disjoint components). This presents a puzzle, since most human-designed DBNs are sparse even with very large Δ. Such models must implicitly be making approximations. In this chapter, we will show how these approximations are a natural outcome when the variables have widely different timescales (rates of evolution).

In a deterministic dynamic model, the idea of using a wide separation of timescales to simplify the model goes back at least to work by Michaelis and Menten (1913); see Iwasaki and Simon (1994), Gómez-Uribe *et al.* (2008) for more recent surveys. The general analysis involves finding gaps in the eigenspectrum of the coefficient matrix of the system of differential equations. Here, we provide the simplest possible example: a system of two variables, s and f, where s (the "slow" variable) influences f (the "fast" variable) but not vice versa:

$$\frac{ds}{dt} = a_1 s \; ; \qquad \frac{df}{dt} = b_1 s + b_2 f \tag{3.1}$$

where we assume $|a_1| \ll |b_2|$ and both negative. Viewed as a (deterministic) DBN, this looks like Figure 3.1(a). The exact solution for some time t is

$$s = S_0 e^{a_1 t} \; ; f = \left(\frac{b_1 S_0}{a_1 - b_2} \right) e^{a_1 t} + \left(F_0 - \frac{b_1 S_0}{a_1 - b_2} \right) e^{b_2 t} \tag{3.2}$$

where S_0 and F_0 are initial values for s and f. This is represented by the DBN structure shown in Figure 3.1(b) for a large finite time step Δ. Although f is nominally a "fast" variable, the solution shows that, for $t \gg 1/|b_2|$, f follows a slowly changing equilibrium value that depends on s. Thus, we need model only the dynamics of s and can compute $f(t)$ directly from $s(t)$. This corresponds to the DBN structure in Figure 3.1(c). With this structure, we can use a large Δ because s changes very slowly.

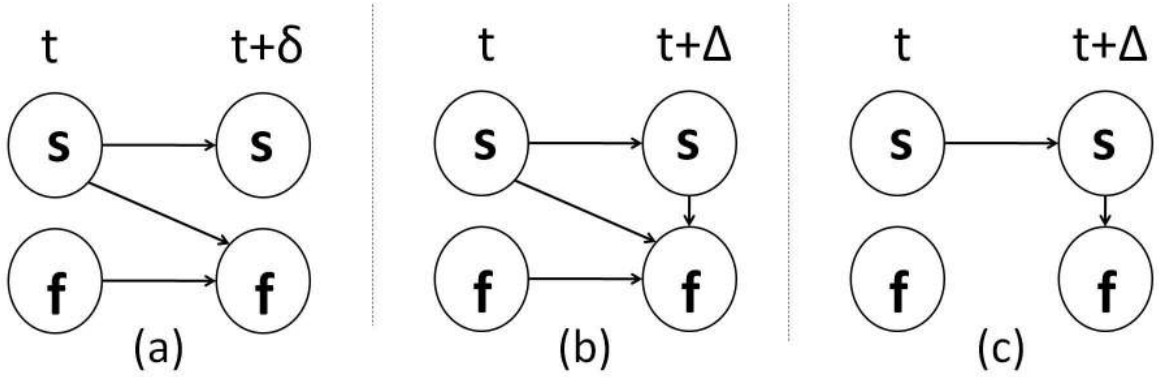


Figure 3.1: Two variable DBN: The slow variable s is independent of the fast variable f. (a) Exact model for small time-step δ. (b) Exact model for large time-step Δ. (c) Approximate model for large time-step Δ.

Effective *model reduction* methods have been developed in the dynamical systems literature for deriving such simplified deterministic models in a semi-automated fashion. Moreover, any discrete-state DBN model can be converted into an equivalent deterministic dynamical system whose variables are the occupancy probabilities of individual states, and model-reduction techniques can be applied to this system. Unfortunately, this approach involves an *exponential blowup* in the model size; furthermore, even if it can be computed, the reduced version would not necessarily correspond to a meaningful sparse model in terms of the original variables.

Instead, we work directly with the DBN, beginning with a very-small-time-step model, identifying time steps Δ that nicely separate the model's time scales, and deriving the corresponding reduced DBN for each such Δ. A salient feature of the algorithm is that it avoids building the intractably large full transition matrix. For large Δ, accurate simulation over very long time periods becomes possible; moreover, the per-time-step inference cost for the reduced models can be much less than for the original models, since the models become sparser as Δ becomes larger. The larger time-step combined with the simpler model result in speed-ups of several orders of magnitude compared to the original model.

Section 3.2 presents *timescales* and other relevant definitions. Section 3.3 introduces an example DBN that models the body's pH control system. Section 3.4 presents the approximation scheme, a proof of its correctness and an analysis of the associated error. Section 3.5 extends these ideas to obtain a general set of rules to construct an approximate DBN for a large time-step. Sec-

tion 3.6 presents results on the accuracy and computational cost of the approximate DBNs of the pH control mechanism.

3.2 Definitions

Consider a simple one-variable DBN, where the variable (say X) can be in one of k discrete states. Let $p_{i,j} = Pr(X_{t+1} = j | X_t = i)$. We define the *timescale of X for state i*, T_X^i, as the expected number of time-steps that the variable spends in state i before changing state, given that it is currently in state i. It can be shown that $T_X^i = 1/(1 - p_{i,i})$. Thus, the overall timescale of variable X is actually a range of timescales from $\min_i T_X^i$ to $\max_i T_X^i$.

In a general DBN, let $\hat{\pi}(X_{t+1})$ denote the parents of variable X_{t+1} in the 2-TBN representation *other than* X_t. Then the conditional probability table (CPT) for X_{t+1} is given by $p_{i,j}^k = Pr(X_{t+1} = j | X_t = i, \hat{\pi}(X_{t+1})_t = k)$. We also define $T_X^{i,k} = 1/(1 - p_{i,i}^k)$ to be the timescale of variable X in state i when its parents are in state k, where k is any state in the joint state space of $\hat{\pi}(X_{t+1})$.

Now, consider two variables X_1 and X_2 in a general DBN. Let $l_{X_1} = \min_{i,k} T_{X_1}^{i,k}$ and $h_{X_2} = \max_{i,k} T_{X_2}^{i,k}$. If $l_{X_1} \gg h_{X_2}$, then X_1 and X_2 are said to be *slow* and *fast* variables (respectively) with respect to each other. Their *timescale separation* is defined as the ratio l_{X_1}/h_{X_2}. For a set of variables $C = \{X_1, \ldots, X_n\}$, the lower timescale bound is defined as $l_C = \min_{X_i \in C} l_{X_i}$, with h_C also defined in a similar fashion. The existence of significant timescale separation in a DBN is crucial in allowing accuracy-preserving model simplifications.

A continuous-time analog of the DBN is the continuous-time Markov chain. Formally, it is defined as a Markov stochastic process $\{\mathbf{x}_t\}_{t \in \mathbb{R}^+}$ with state space $\mathbb{I}$. Let $\mathbb{I} = \{1, 2, \ldots, k\}$. The transition matrix for the interval from 0 to t, $P_{ij}(t) = Pr(\mathbf{X}_t = j | \mathbf{X}_0 = i), (i, j) \in \mathbb{I} \times \mathbb{I}$, is given by $P(t) = e^{Lt}$, where the matrix L is called the *generator* of the Markov chain. L has the following properties: (i) $\sum_j l_{ij} = 0, \forall i \in \mathbb{I}$ (this makes L *conservative*); (ii) $l_{ij} \in [0, \infty), \forall (i, j) \in \mathbb{I} \times \mathbb{I}$ with $i \neq j$. L can be computed by:

$$L = \lim_{t \to 0} \frac{P(t) - I}{t} \tag{3.3}$$

We will assume that the transition matrix $P(t)$ is *standard* (i.e., $\lim_{t \to 0} P_{ii}(t) = 1, \forall i \in \mathbb{I}$).

3.3 A Motivating Example: Human pH Regulation System

pH is a measure of the concentration of hydrogen ions in a solution or substance. Measured on a log scale of 0–14, higher numbers represent alkaline nature and lower numbers are characteristic of acids. The pH balance of the human bloodstream is one of the most important biochemical balances

in the human body since it controls the speed of our body's biochemical reactions (Guyton & Hall, 1997).

The human body has a complex system to maintain body pH around a setpoint ($\simeq 7.4$) under normal circumstances. Generally, metabolism leads to CO_2 production, thereby producing carbonic acid, which lowers the pH of blood (an abnormal lowering leads to "acidosis"). On the other hand, respiration brings O_2 into the system and also removes CO_2, which neutralizes the acid in the blood, thus raising the pH. These are the two main compensatory mechanisms that we will consider in our model. Chemical acid–base buffer systems of the body fluids (which provide the first line of defense against fluctuations in blood pH) are not modeled. Metabolism rate increases with increasing temperature and higher levels of exertion. The respiratory rate (measured as "Minute Volume", which is the volume of air inhaled in a minute) is raised by a lower pH value, if the pH setpoint is normal. However, an overdose of certain narcotics might lower the pH setpoint of the body. In this case, a low pH will not trigger a rise in the Minute Volume, causing the blood to become more and more acidic and possibly resulting in death. Figure 3.2 shows the DBN model of the system which controls the body pH. Variables with similar shades have comparable timescales. The darkest shaded variable (i.e., "Minute Volume") has the fastest dynamics, while the lightest shaded variable ("pH setpoint") has the slowest dynamics. Details of each variable in the model are provided in Table 3.1.

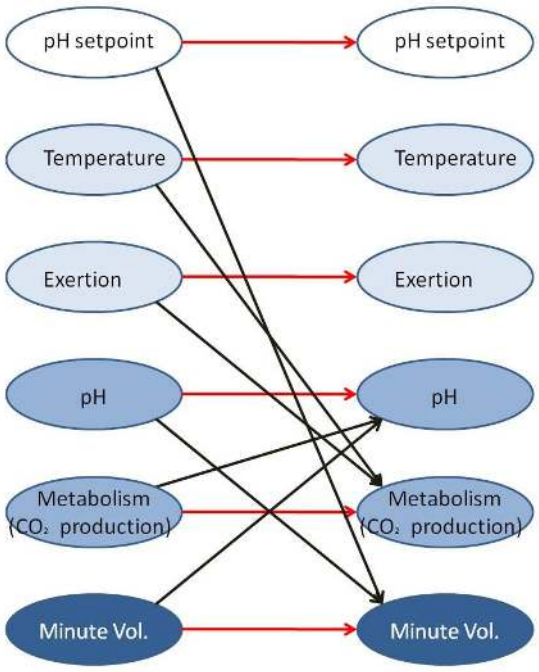


Figure 3.2: Exact model for the pH control system for a small time-step δ.

Table 3.1: Information about the variables in the DBN (including their state space and timescales)

Details of the pH control system DBN		
Variable Name	State Space	Timescale (Seconds)
pH setpoint (pHst)	{ Normal, Low }	l_{pHst} = $3.3e6$ h_{pHst} = $1e7$
Temperature (Temp)	{ Hot, Warm, Cool, Cold }	l_{Temp} = $8e3$ h_{Temp} = $1e4$
Exertion (Ex)	{ High, Normal, Low}	l_{Ex} = $5e3$ h_{Ex} = $1e4$
pH (pH)	{ Acid, Neutral, Alkaline }	l_{pH} = 100 h_{pH} = 300
Metabolism (Meta)	{ High, Normal, Low }	l_{Meta} = 70 h_{Meta} = 150
Minute Volume (MV)	{ High, Normal, Low }	l_{MV} = 1.1 h_{MV} = 5

We chose this model as a motivating example, since there are interacting variables in this system which evolve at very different timescales. We will construct approximate, sparsely connected models for this system over large time-steps in Section 3.6.

3.4 Approximation scheme

Let us consider the general 2-variable DBN in Figure 3.3(a). (Although one might expect a link between $s_{t+\delta}$ and $f_{t+\delta}$, we can reduce δ appropriately to drop the intra-time-slice links since *any differential equation system can be represented without contemporaneous edges*.) For simplicity of presentation, we will assume that s and f are binary random variables (although all the results presented in this section are generally applicable to any finite discrete state space for the two variables). Since we assume there is a timescale separation in the dynamics of s and f, their CPTs should have the following structure:

$$\mathbf{p}(\mathbf{s_{t+1}}|\mathbf{s_t}, \mathbf{f_t}) = \begin{bmatrix} 1-\epsilon x_1 & \epsilon x_1 \\ 1-\epsilon x_2 & \epsilon x_2 \\ \epsilon x_3 & 1-\epsilon x_3 \\ \epsilon x_4 & 1-\epsilon x_4 \end{bmatrix} \tag{3.4}$$

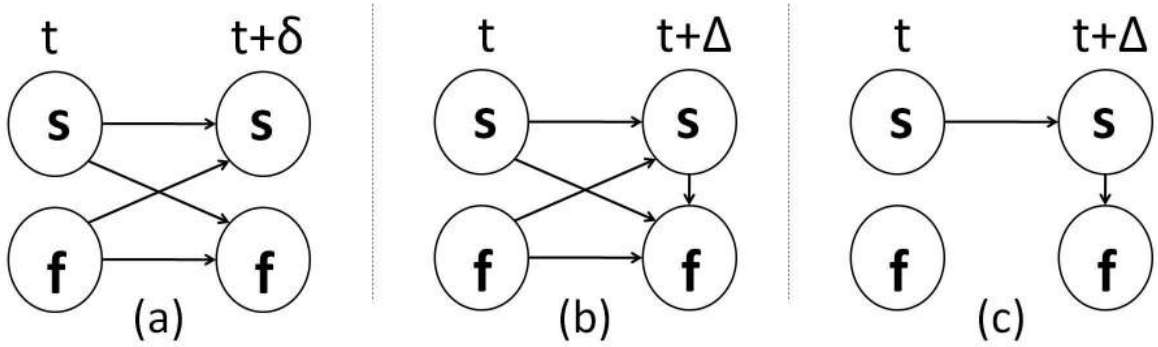


Figure 3.3: Two variable general DBN: The slow variable s is also dependent on the fast variable f. (a) Exact model for small time-step δ. (b) Exact model for large time-step Δ. (c) Approximate model for large time-step Δ.

$$\mathbf{p}(\mathbf{f_{t+1}}|\mathbf{s_t}, \mathbf{f_t}) = \begin{bmatrix} 1-a_1 & a_1 \\ a_2 & 1-a_2 \\ 1-a_3 & a_3 \\ a_4 & 1-a_4 \end{bmatrix} \tag{3.5}$$

where $\epsilon \ll 1$, $0 < a_i, x_i < 1$ and $a_i, x_i \gg \epsilon$. The rows correspond to (s_t, f_t) = $\{(0,0)$, $(0,1)$, $(1,0)$, $(1,1)\}$ respectively. The first column corresponds to s_{t+1} (or f_{t+1}) = 0. Using the definitions in Section 3.2, l_s = $\min_i 1/\epsilon x_i$ and h_f = $\max_i 1/a_i$. Therefore $l_s/h_f = O(1/\epsilon)$. It should be noted that ϵ is a redundant parameter in this specification—hence we have an option to choose an ϵ. This choice has to be made such that the order of the x_i's and a_i's are similar and they also satisfy the previous constraints.

The exact transition model for a larger time-step Δ is shown in Figure 3.3(b). Without loss of generality, let us assume $\delta = 1$. As shown in Figure 3.3(c), for the large time-step Δ, the distribution of $f_{t+\Delta}$ becomes (approximately) independent of the value of s_t and f_t. The key observation is that irrespective of the value of f_t, the distribution of f_{t+i}, for i $\in [1, \Delta]$ will exponentially converge to the equilibrium distribution of f for the current value of s. Once it does so (approximately), we can compute an exact expression for $\hat{P}(s_{t+(N+1)}|s_{t+N})$, where N is large enough for f to approximately reach its equilibrium distribution. This expression will be equal to

$$\hat{\mathbf{P}}(\mathbf{s_{t+1}}|\mathbf{s_t}) = \sum_{\mathbf{f_t}} \mathbf{p}(\mathbf{s_{t+1}}|\mathbf{s_t}, \mathbf{f_t}) \times \mathbf{P}_\infty(\mathbf{f_t}|\mathbf{s_t}) \tag{3.6}$$

where $P_\infty(f_t|s_t)$ is the equilibrium distribution of f. Since f (nearly) reaches its equilibrium in a short fraction of Δ, we ignore that portion of f's trajectory, and simply assign $(\hat{P}(s_{t+(N+1)}|s_{t+N}))^\Delta$ to be the CPT of s for the large time-step Δ. Equation 3.6 is analogous to Forward Euler integration since we use s_t only to determine the equilibrium distribution of f.

The CPT $\hat{P}(f_{t+\Delta}|s_{t+\Delta})$ is simply the invariant distribution of f for the fixed value of $s_{t+\Delta}$.

Correctness of the approximation scheme

The above approximation heuristic is analogous to elimination of the fast variable through averaging for the continuous-time Markov chain. (For a complete treatment of the continuous-time case, see Pavliotis and Stuart (2007).) In particular, let the state spaces of s and f be $\mathbb{I}_s$ and $\mathbb{I}_f$ respectively. Let $q((i,k),(j,l))$ denote the element of the generator matrix associated with transition from $(i,k) \in \mathbb{I}_s \times \mathbb{I}_f$ to $(j,l) \in \mathbb{I}_s \times \mathbb{I}_f$. $B_0(i)$ is a generator with entries $b_0(k,l;i)$ where the indices indicate transition from $k \in \mathbb{I}_f$ to $l \in \mathbb{I}_f$ for given fixed $i \in \mathbb{I}_s$. For each $i \in \mathbb{I}_s$, $B_0(i)$ generates an ergodic Markov chain on $\mathbb{I}_f$. Let $\rho^B_\infty(k;i)_{k\in\mathbb{I}_f}$ be the invariant distribution of a Markov chain on $\mathbb{I}_f$, indexed by $\mathbb{I}_s$. Similarly, let $B_1(k)$ with indices $b_1(i,j;k)$ denote transition from $i \in \mathbb{I}_s$ to $j \in \mathbb{I}_s$, for each fixed $k \in \mathbb{I}_f$. Let us introduce generators Q_0 and Q_1 of Markov chains on $\mathbb{I}_s \times \mathbb{I}_f$ by:

$$q_0((i,k),(j,l)) = b_0(k,l;i)\delta_{ij},$$

$$q_1((i,k),(j,l)) = b_1(i,j;k)\delta_{kl},$$

where δ_{ij} and δ_{kl} are Kronecker delta functions. Now, let us define another generator $\bar{Q}$ of a Markov chain on $\mathbb{I}_s$ by $\bar{q}(i,j) = \sum_k \rho^B_\infty(k;i) b_1(i,j;k)$.

Lemma 1

If Q, the generator of a Markov chain, takes the form $Q = \frac{1}{\epsilon}Q_0 + Q_1$, then for $\epsilon \ll 1$ and times t up to $O(1)$, the finite-dimensional distribution of $s \in \mathbb{I}_s$ is approximated by a Markov chain with generator $\bar{Q}$ with an error of $O(\epsilon)$.

The proof (Pavliotis & Stuart, 2007, Section 9.4) bounds the error on a vector v, such that $v_i(t) = \mathbb{E}(\phi_{x(t)}|x(0) = i)$, where $\phi : \mathbb{I} \mapsto \mathbb{R}$. $v(t)$ satisfies the backward Kolmogorov equation (i.e., $dv/dt = Lv$; $v(0) = \phi$). Thus $v(t) = P(t)\phi$, where $P(t)$ is the transition matrix for the interval from 0 to t. Since this is true for any mapping ϕ, the approximation error in any element of $P(t)$ is necessarily bounded by $O(\epsilon)$ too.

We now show how our approximation scheme for the discrete time-step is equivalent to their solution for continuous time and thereby shares the same order of approximation error. The first step towards proving equivalence is to show that the generator matrix corresponding to the discrete-time process also has a similar structure (i.e., $Q = \frac{1}{\epsilon}Q_0 + Q_1$). Firstly, for $\delta \ll 1$ and an integer $n > 0$,

$$\begin{bmatrix} 1-\delta x & \delta x \\ \delta y & 1-\delta y \end{bmatrix}^n \approx \begin{bmatrix} 1-n\delta x & n\delta x \\ n\delta y & 1-n\delta y \end{bmatrix} \tag{3.7}$$

when we ignore δ^2 terms. If we consider $1/\delta$ to be a large integer, Equation 3.7 implies

$$\begin{bmatrix} 1-x & x \\ y & 1-y \end{bmatrix}^{\delta} \approx \begin{bmatrix} 1-\delta x & \delta x \\ \delta y & 1-\delta y \end{bmatrix} \tag{3.8}$$

Let P_f^δ (similarly P_s^δ) be the CPT for the dynamics of f (s) over an infinitesimal time-step δ when we freeze s (f). Using Equation 3.8, we get:

$$P_s^\delta \approx \begin{bmatrix} 1-\delta\epsilon x_1 & \delta\epsilon x_1 \\ 1-\delta\epsilon x_2 & \delta\epsilon x_2 \\ \delta\epsilon x_3 & 1-\delta\epsilon x_3 \\ \delta\epsilon x_4 & 1-\delta\epsilon x_4 \end{bmatrix}; P_f^\delta \approx \begin{bmatrix} 1-\delta a_1 & \delta a_1 \\ \delta a_2 & 1-\delta a_2 \\ 1-\delta a_3 & \delta a_3 \\ \delta a_4 & 1-\delta a_4 \end{bmatrix}$$

We now combine P_s^δ and P_f^δ to form $P_{s,f}^\delta$.

$$P_{s,f}^\delta \approx \begin{bmatrix} (1-\delta\epsilon x_1)(1-\delta a_1) & \cdots & \cdots & \delta\epsilon x_1 \delta a_1 \\ \vdots & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ \delta\epsilon x_4 \delta a_4 & \cdots & \cdots & (1-\delta\epsilon x_4)(1-\delta a_4) \end{bmatrix}$$

The generator corresponding to this discrete-time process can be computed by the formula $L = \lim_{\delta\to 0}(P_{s,f}^\delta - I)/\delta$. Since we divide by δ when taking the limit, ignoring the higher order terms of δ in the previous step(s) becomes inconsequential. The generator matrix L thus obtained is:

$$L = \begin{bmatrix} -a_1 & a_1 & 0 & 0 \\ a_2 & -a_2 & 0 & 0 \\ 0 & 0 & -a_3 & a_3 \\ 0 & 0 & a_4 & -a_4 \end{bmatrix} + \epsilon \begin{bmatrix} -x_1 & 0 & x_1 & 0 \\ 0 & -x_2 & 0 & x_2 \\ x_3 & 0 & -x_3 & 0 \\ 0 & x_4 & 0 & -x_4 \end{bmatrix}$$

Hence, the generator has the form $L_0 + \epsilon L_1$. Thus, $L = O(\epsilon Q)$. Since $P(t) = e^{Lt}$, the behavior of L at time T/ϵ is similar to the behavior of Q at time T. Thus we can use Lemma 3.4 to say the following:

If L, the generator of a Markov chain, takes the form $L = L_0 + \epsilon L_1$, then for $\epsilon \ll 1$ and times t up to $O(1/\epsilon)$, the finite-dimensional distribution of $s \in I_s$ is approximated by a Markov chain with generator $\bar{L}$ with an error of $O(\epsilon)$, where $\bar{L}(i,j) = \sum_k \rho_\infty^L(k;i) l_1(i,j;k)$.

It is easy to see that the generator corresponding to the transition matrix $\hat{P}(s_{t+1}|s_t)$ (Equation 3.6) is exactly equal to the generator $\bar{L}$, and hence we arrive at the following result.

Result 1

If a discrete time Markov process has conditional probability tables given by Equations 3.4 and 3.5, then for $\epsilon \ll 1$ and times Δ up to $O(1/\epsilon)$, the finite-dimensional distribution of $s \in \mathbb{I}_s$ is approximated by $\hat{P}(s_{t+1}|s_t)^{\Delta}$ (given by Equation 3.6), with error $O(\epsilon)$.

For very small values of Δ (like $O(1)$), the error for f decays exponentially ($O(|\lambda|^{\Delta})$) where λ is the maximum singular value of $p(f_{t+1}|s_t, f_t)$. However, for $\Delta = O(1/\epsilon)$, the error for f essentially replicates the error for s, since the fast ergodic dynamics of f has almost reached a quasi-static equilibrium.

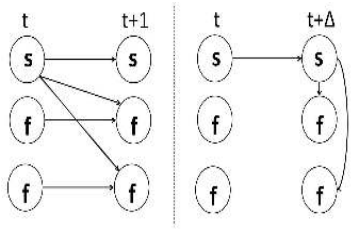


Figure 3.4: Structural transformation in the large time-step model when f_1 and f_2 have **no** cross links in the small time-step model

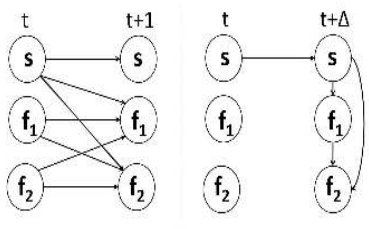


Figure 3.5: Structural transformation in the large time-step model when f_1 and f_2 have cross links in the small time-step model

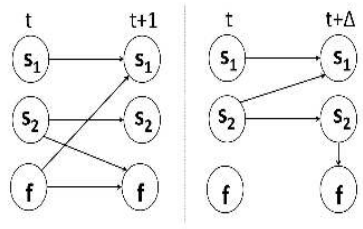


Figure 3.6: A slow cluster s_1 has a new parent s_2 in the larger time-step model when s_2 is a parent of f in the smaller time-step model

Special case

In the exact model, if $p(s_{t+1}|s_t, f_t) = p(s_{t+1}|s_t)$ (i.e., the dynamics of s is independent of f as shown in Figure 3.1), then the dynamics of s is tracked exactly by the above scheme. In Equation 3.6, the $p(s_{t+1}|s_t, f_t)$ term goes outside the summation, and the invariant distribution of f sums to 1.

Other approaches

There is another line of work (Yin & Zhang, 2004) where discrete time transition models of the form $P_\epsilon = P + \epsilon Q$ are considered. Here, P is a stochastic matrix and Q is a generator matrix. The approximation error for P_ϵ^k can be made $O(\epsilon^{n+1})$ by constructing a series of approximation functions. While this approach has the benefit of a potentially much smaller error, the functions are much more expensive to compute and the approximate model has no simple or intuitive mapping to the original model.

3.5 General Rules of Construction

This section describes the general algorithm for constructing an approximate DBN for a larger time-step when given the exact DBN for a small time-step δ. We will use the approximate CPT construction discussed in section 3.4. Let the DBN have n variables X_1, X_2, ...,X_n. This algorithm will produce a sequence of approximate DBNs for various increasing values of Δ (the larger time-step). The algorithm is as follows:

1. For each variable X_i, determine l_{X_i} and h_{X_i}.
2. Cluster the variables into $\{C_1, C_2, \ldots, C_m\}$ ($m \leq n$), such that $\epsilon_i \approx \frac{h_{C_i}}{l_{C_{i+1}}} \ll 1$, $\forall i \in \{1, 2, \ldots, m-1\}$, i.e., there is a significant timescale separation between successive clusters. C_1 is the cluster of fastest variables, while C_m is the cluster of slowest variables. In the worst case, m can be 1, when all variables have very comparable timescales.
3. Repeat the following steps for $i = \{1, \ldots, m-1\}$. Let $\Delta_0 = 1$.
   a) Select $\Delta_i = \Delta_{i-1} \times O(1/\epsilon_i)$
   b) For each configuration of the slower parents of C_i, compute the stationary point (equilibrium distribution) of C_i to fill in the CPT of $p((C_i)_{t+\Delta_i} | \pi(C_i)_{t+\Delta_i})$ in the approximate model for time-step Δ_i. If the fast variables in C_i are conditionally independent given the slow parents C_j ($j > i$), then the individual equilibrium are calculated (see Figure 3.4). However, if the variables in C_i are not conditionally independent given C_j ($j > i$), we have to compute the joint equilibrium of C_i (as in Figure 3.5).
   c) While C_i only has parents in the same time-slice in the approximate model, C_j ($j > i$) will have parents from the previous time-slice. If there were no links to C_j from C_i in the exact model, then the CPT of C_j is exactly equal to $C\hat{P}T_{C_j}^{(\Delta_i/\Delta_{i-1})}$ (as mentioned in section 3.4), where $C\hat{P}T_{C_j}$ is the joint CPT of C_j and its parents for time-step Δ_{i-1}. In the worst case, all C_j's ($j > i$) can become fully connected.
   However, if there are links from the fast cluster C_i to the slow cluster C_j, then we have to use Equation 3.6 to compute the CPT of C_j for time-step Δ_i. *Since the equilibrium distribution of C_i is parameterized by the parents of C_i, these variables now become additional parents of C_j* (see Figure 3.6).
   d) Now we have an approximate model for time-step Δ_i. This model only has links across time-slices for C_j ($j > i$). This approximate model is used as the exact model for the next iteration (since using the exact model would result in the same approximations).

The sequence of DBNs produced by this algorithm become more and more sparse. This makes exact inference on these approximate models much more feasible than the fully connected exact model for the same time-step. Let D_s be the number of variables in the slow clusters and D_f be

the number of variables in the fast clusters and let all variables be binary. Then, the complexity of exact inference (per time step) in the fully connected, exact model is $O(2^{2\times(D_s+D_f)})$ while that in the approximate model is $O(2^{2\times D_s} + 2^{D_s+D_f})$. This complexity is for the projection of the joint state space distribution vector. Also, particle filters will run much faster on these approximate models because particles are only needed to estimate the joint state space of D_s and not $D_s \cup D_f$ (as is the case in the exact model). In the next section, we return to the pH regulation model from Section 3.3 and create approximate models for appropriate values of Δ.

3.6 Experiment

As mentioned previously, the pH regulation model exhibits a wide range of timescales. Minute Volume can potentially change every second, depending on the current needs of the body. The pH of the body and the rate of metabolism have much slower dynamics relative to Minute Volume. Temperature and Exertion are even slower, while pH setpoint (which only changes upon a heavy overdose of narcotics, a very rare event) is the slowest of all. (Timescales are specified in Table 3.1.) Thus, there are four separate clusters of variables on which we can apply the algorithm of Section 3.5.

The three approximate models created for $\Delta = 20$, $\Delta = 1000$ and $\Delta = 50000$ are shown in Figure 3.7. For the first approximate model M_{20}, only Minute Volume is the fast variable. Hence, its parents are pH and pH-setpoint from the same time-slice. Also, since pH-setpoint determines the equilibrium distribution of Minute Volume, which in turn is a parent of pH (in the exact model), pH now has an additional parent, pH-setpoint (according to step 3.(c) in Section 3.5). For the second approximate model M_{1000}, pH and Metabolism are the new fast variables. Since Metabolism is a parent of pH, we have to consider the joint equilibrium of the two variables, given each configuration of their slow parents (i.e., pH-setpoint, Temperature and Exertion). For the third approximate model M_{50000}, Temperature and Exertion also become fast variables. Since these were independently evolving variables, they do not have any parents in M_{50000}.

For evaluation purposes, we implemented the exact model and the three approximate models in MATLAB. We chose 10 random starting configurations of the variables and simulated the exact trajectory of the belief vector for each of these initial configurations for 1,000,000 time-steps. Then we used the same starting configurations and M_{20} to simulate the trajectories at regular intervals of 20 steps over 1 million time-steps. We repeated the same procedure with M_{1000} and M_{50000} to simulate the trajectories at intervals of 1000 and 50000 steps respectively. The L2-error (per time step) of the joint state space belief vector was averaged over all 10 instances and plotted in Figure 3.8. As expected, the error increases with the level of approximation—although all three models perform well.

The performance speed-up details of the different models are summarized in Table 3.2. The speed-up factor for M_{20} was less than 20 because the exact model required only matrix multi-

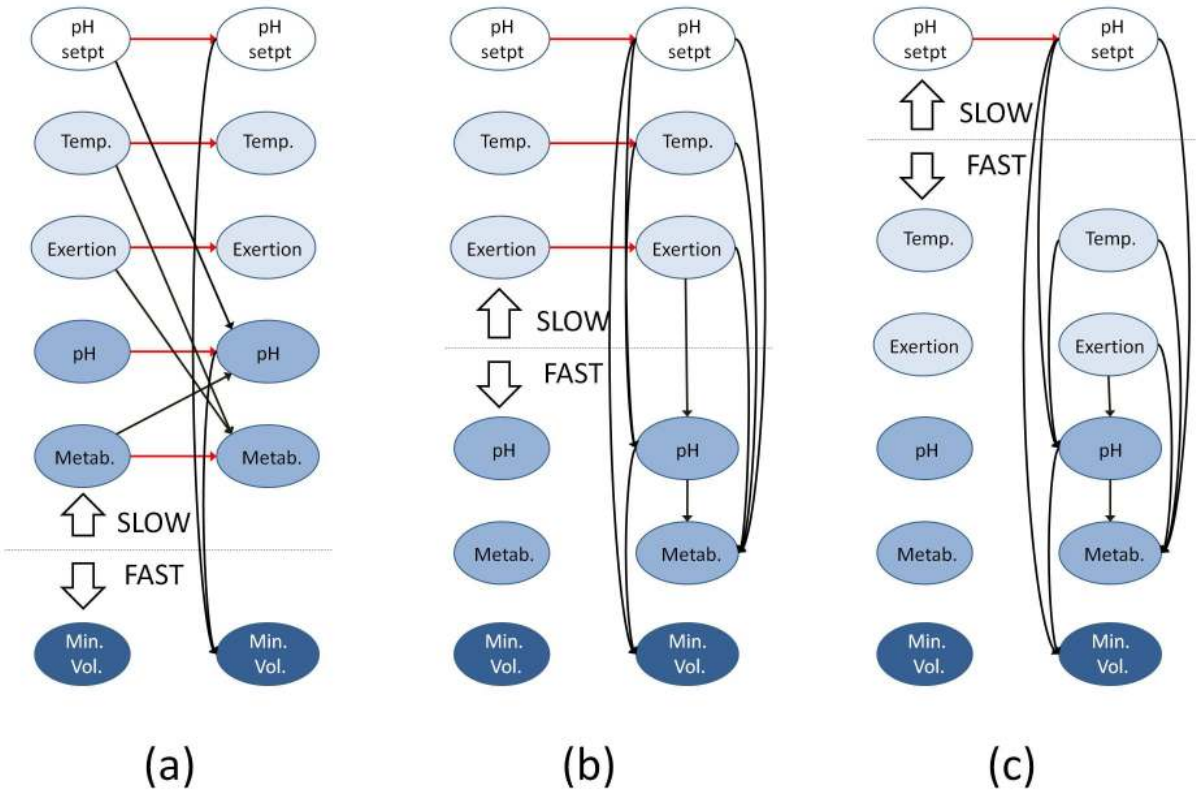


Figure 3.7: Approximate models of the pH regulation system of the human body. (a) Approximate model for $\Delta = 20$. (b) Approximate model for $\Delta = 1000$. (c) Approximate model for $\Delta = 50000$

Table 3.2: Computational speed-up in different models

Model	Avg. Simulation Time (sec)	Speedup Factor
Exact	385.44	1
$\Delta = 20$	24.87	15.5
$\Delta = 1000$	.0889	4300
$\Delta = 50000$	.0006327	> 600000

plication (very efficient in MATLAB), while M_{20} needed some indexing work to compute the distribution of the fast variable Minute Volume even though its matrix multiplication requirements were less. The benefit of a much simpler (sparser) model was evident for both M_{1000} and M_{50000}, as the speed-up factor exceeded the size of the time-step (Δ).

Since this is a model for pH regulation, we also decided to check the performance of the approximate models on the marginal distribution of pH. Since pH is a slow variable in M_{20}, it is

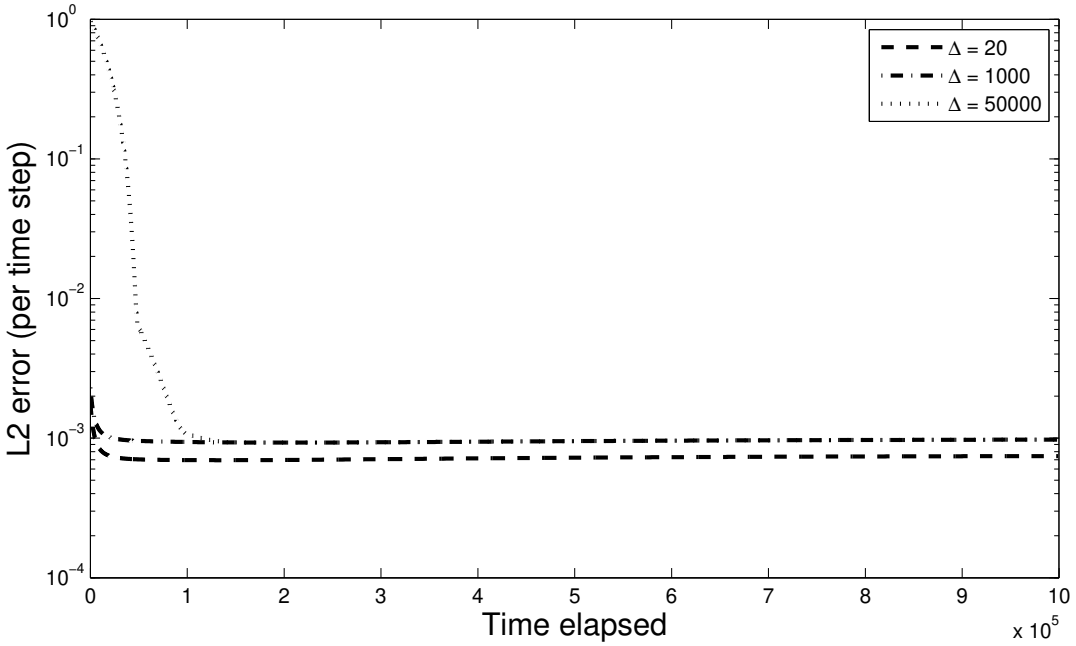


Figure 3.8: Comparison of the average L2-error(per time-step) of the belief vector of the joint state space for M_{20}, M_{1000} and M_{50000}.

simulated exactly in that model and hence is not relevant to this experiment. As we can see from Figure 3.9, both M_{1000} and M_{50000} perform very well with an error of less than 0.04%.

3.7 Conclusion

We have shown how DBNs that are naturally sparse for a small time step may be converted to (different) sparse DBNs for large time steps, even though an exact conversion methods would yield a fully connected model. The sparse approximation becomes more and more accurate with increasing separation of timescales among variables. Our error analysis also supports a quantitative trade-off between accuracy and speed-up. The methods accommodate models with widely varying timescales and/or intermittent observations and should be applicable to a broad range of chemical, biological, and social systems with these properties.

Aleks *et al.* (2009) noted that specifying a DBN may be quite easy for a small time-step but much harder for a larger time-step. This construction automates the conversion. Thus, it allows the user to build a DBN at a natural time-step, yet run it at much larger time-steps to reduce computational cost.

Further work along these lines includes extending the results to handle continuous variables;

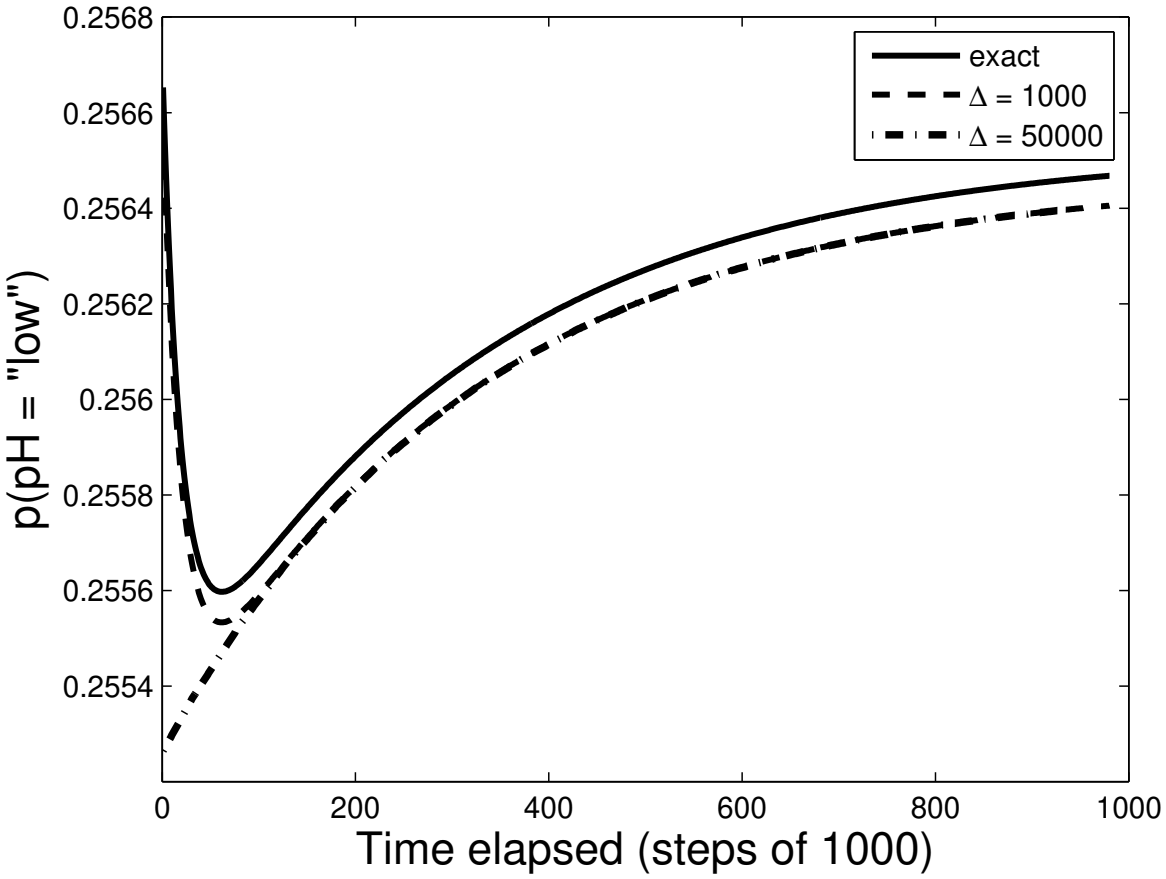


Figure 3.9: Accuracy of M_{1000} and M_{50000} in tracking the marginal distribution of pH

adding the possibility of replacing a state variable by another variable corresponding to its long-term average value (e.g., replacing instantaneous blood pressure by its one-minute average); and adding the possibility of replacing a set of variables by functions of those variables (e.g., replacing two Boolean variables by their XOR, or two continuous variables by linear combinations thereof). These two latter ideas both create additional scope for clean separation of timescales.

Chapter 4

Eigencomputation for factored systems

In the previous chapter, we limited ourselves to considering temporal systems where individual variables were evolving at very different rates. However, we would like to ease this constraint and instead generalize our technique to systems where different linear combinations of the variables evolve at widely separated rates. In such a system, a spectral gap exists between the first few eigenvalues and all the remaining ones. Accurate approximate inference on such a system can be performed by only tracking the eigenvectors corresponding to the eigenvalues before (i.e., larger than) the spectral gap. However, identifying these eigenpairs requires a spectral analysis, which is computationally very expensive if we use conventional methods. This chapter presents an algorithm that combines a numerical eigenpair algorithm (namely the Arnoldi iteration) with factored representations of belief states to perform an approximate eigenanalysis of a DBN (or any factored linear system).

4.1 Linear systems

A linear system is a mathematical model of a system whose transformation from input to output is defined by a linear operator. Specific examples of linear systems discussed previously include hidden Markov models (HMMs) and dynamic Bayesian networks (DBNs). In such systems, the output $\mathbf{u}'$ is related to the input $\mathbf{u}$ by the following relation:

$$\mathbf{u}' = A\mathbf{u} \tag{4.1}$$

where A is an $m \times n$ matrix and the input and output are $n \times 1$ vectors.

In temporal systems, $\mathbf{u}$ corresponds to the belief vector at the current time step t (i.e., $\mathbf{u}_t$) while

$\mathbf{u}'$ corresponds to the belief vector at the next time step $t+1$ (i.e., $\mathbf{u}_{t+1}$). Let $\mathbf{x}_t$ denote the state of the system at time t – this is a value between 1 and n. Since the state space is generally identical in subsequent time steps, A is a square matrix. The entry $a_{i,j}$ is the probability of transitioning to state i if the current state is j (i.e., $a_{i,j} = Pr(\mathbf{x}_{t+1} = i | \mathbf{x}_t = j)$). Such a matrix is called a *stochastic matrix*. Each column of a stochastic matrix represents a probability distribution and hence the sum of the elements in each column is 1. Also, all its entries are non-negative.

Factored representation

For clarity of exposition, we will focus on systems with binary variables in the rest of this chapter. However, the algorithms presented here extend trivially to any system of discrete-valued variables. The number of independent parameters required to specify an HMM with n discrete states is $n^2 - n$ (since each column has one dependent parameter). Since the number of possible states for a system increases exponentially with the number of variables, this specification is too expensive. Instead, practitioners use conditional independence relations to specify the linear evolution of a temporal system using DBNs (which were introduced in Chapter 2 and discussed further in Chapter 3.

The number of independent parameters required to specify a DBN with K binary variables (K is $O(log_2\ n)$) is $O(K)$, as long as the maximum number of parents of a variable is $O(1)$. This makes the specification of large systems feasible.

4.2 Computational complexity

In this chapter, we study the problem of computing the eigenvalues and eigenvectors of a linear system which is specified in a factored form. The naive way to solve this problem would be to convert the factored representation into the equivalent dense matrix (i.e., convert the DBN to its equivalent HMM), and use the *QR algorithm* on the resultant dense matrix. The computational complexity of this process would be $O(n^3)$ or $O(2^{3K})$.

If we are only interested in a few extreme eigenvalues, then we can use *Arnoldi iteration* on the dense HMM representation. The computational complexity to compute r eigenvalues is $O(r2^{2K})$. This is still *too expensive* since it is exponential in the size of the problem specification (i.e., the input, which is $O(K)$). The exponential blowup is a result of converting the factored form to the dense HMM.

However, it is possible to work with the original factored form and still perform an Arnoldi iteration, although certain approximations need to be made, primarily to keep every representation from suffering an exponential blowup.

4.3 Factored belief vector and forward projection

The key step in the Arnoldi iteration to find the r most extreme eigenvalues of a matrix A is to compute the vectors $\mathbf{x}, A\mathbf{x}, \cdots, A^{r-1}\mathbf{x}$, for a random initial vector $\mathbf{x}$. Since in a DBN, $\mathbf{x}$ is the belief vector over the state space, its uncompressed size would be $n \times 1$. However, this is too large. So, instead we have to resort to some factored representation for the belief vector as well.

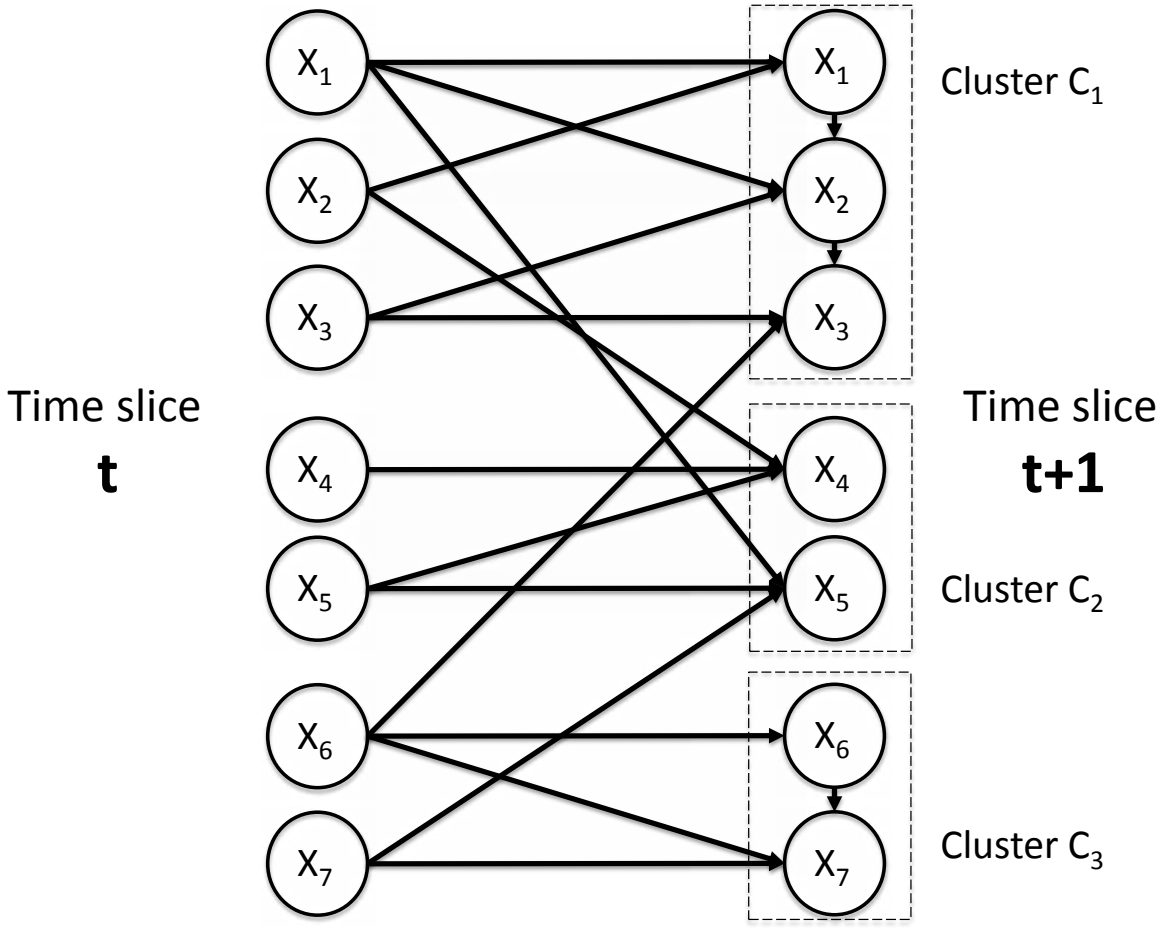


Figure 4.1: A 2-TBN with 7 binary variables in each time slice. The belief vector is maintained as a Kronecker product of belief vectors over clusters of variables – $C_1 = \{X_1, X_2, X_3\}$, $C_2 = \{X_4, X_5\}$ and $C_3 = \{X_6, X_7\}$

One such possibility is to assume that there are variable clusters within the system which are independent of one another. For instance, as shown in Figure 4.1, variables X_1, X_2, X_3 form cluster C_1 whereas X_4, X_5 form C_2 and X_6, X_7 form C_3. Therefore

$$Pr(X_1, X_2, X_3, X_4, X_5, X_6, X_7) = Pr(X_1, X_2, X_3)Pr(X_4, X_5)Pr(X_6, X_7) \tag{4.2}$$

The joint belief vector, denoted by $\mathbf{x}$, is the *Kronecker product* of the belief vectors of the

individual clusters

$$\mathbf{x} = \mathbf{x}_{C_1} \otimes \mathbf{x}_{C_2} \otimes \mathbf{x}_{C_3} \tag{4.3}$$

Henceforth, we will refer to such vectors as *Kronecker product vectors*. In this example, $\mathbf{x}$ is a 128×1 vector, while $\mathbf{x}_{C_1}$, $\mathbf{x}_{C_2}$ and $\mathbf{x}_{C_3}$ are 8×1, 4×1 and 4×1 vectors respectively. In general, if the maximum size of a cluster is $O(1)$, then the number of parameters required to specify a Kronecker product vector is $O(K)$.

The first thing to consider is that such a representation is an approximation. As we project the initial random vector (which might be chosen as one that does factorize), it will no longer be factored in the same manner as before. Due to the infeasibility of storing the full-blown joint belief vector that it will eventually become, we store only the cluster marginals after every iteration. This allows us to keep the representation size constant. The Boyen-Koller algorithm (Boyen & Koller, 1998) shows that such factored belief vectors have bounded error in tracking the evolution of a factored system.

4.4 Revisiting Arnoldi

Let us now revisit the Arnoldi iteration algorithm that was introduced in Chapter 2 and identify the modifications required to run the Arnoldi iteration for a factored system. The first modification, of course, is the factored nature of the initial vector $\mathbf{x}$.

Step 1: Forward projection through DBN

This step can be performed via the junction tree algorithm (see Chapter 2 for more details). The cost of this operation is $O(K)$ if the largest clique in the junction tree is $O(1)$. The second part of this step is to project back the resultant vector into the constituent factors – C_1, C_2 and C_3 in this case. (Boyen & Koller, 1998) showed that this back projection into a set of factors enables us to keep the size of the representation of $\mathbf{x}_t$ constant as t increases, while keeping the approximation error bounded. In the absence of this projection step, the representation size would grow quickly to $O(2^K)$. Another option is the *factored frontier* algorithm (Murphy & Weiss, 2001).

An important approximation we have to make is to pre-compute all the projected belief vectors at the outset. After each Arnoldi iteration, the resultant vector will not necessarily have all non-negative components. Since the forward projection algorithms only work with belief vectors, these vectors can no longer be projected using the existing algorithms. Hence, we choose to compute the projected belief vectors before performing the iterative orthogonalization process.

Step 2: Orthogonalize

In order to find a vector which is orthogonal to the existing vectors, we will follow the standard steps of the Arnoldi iteration algorithm. We now present the methods and computational complexities of the various operations required for this orthogonalization.

Dot product

Let two vectors $\mathbf{x}$ and $\mathbf{y}$ be Kronecker product vectors $\mathbf{x}_{C_1} \otimes \mathbf{x}_{C_2} \otimes \mathbf{x}_{C_3}$ and $\mathbf{y}_{C_1} \otimes \mathbf{y}_{C_2} \otimes \mathbf{y}_{C_3}$ respectively. We only consider the case where the clusters are the same since that in ensured by the factorization step mentioned above. We need to compute the dot product of two such Kronecker product vectors in Step 7 of Algorithm 4.1. This can be broken down as follows:

$$\begin{aligned}
\mathbf{x}^T\mathbf{y} &= \left(\mathbf{x}_{C_1} \otimes \mathbf{x}_{C_2} \otimes \mathbf{x}_{C_3}\right)^T \left(\mathbf{y}_{C_1} \otimes \mathbf{y}_{C_2} \otimes \mathbf{y}_{C_3}\right) \\
&= \left({\mathbf{x}_{C_1}}^T \otimes {\mathbf{x}_{C_2}}^T \otimes {\mathbf{x}_{C_3}}^T\right) \left(\mathbf{y}_{C_1} \otimes \mathbf{y}_{C_2} \otimes \mathbf{y}_{C_3}\right) \\
&= \left({\mathbf{x}_{C_1}}^T \otimes \left({\mathbf{x}_{C_2}}^T \otimes {\mathbf{x}_{C_3}}^T\right)\right) \left(\mathbf{y}_{C_1} \otimes \left(\mathbf{y}_{C_2} \otimes \mathbf{y}_{C_3}\right)\right) \\
&= {\mathbf{x}_{C_1}}^T\mathbf{y}_{C_1} \otimes \left({\mathbf{x}_{C_2}}^T \otimes {\mathbf{x}_{C_3}}^T\right) \left(\mathbf{y}_{C_2} \otimes \mathbf{y}_{C_3}\right) \\
&= \left({\mathbf{x}_{C_1}}^T\mathbf{y}_{C_1}\right) \otimes \left({\mathbf{x}_{C_2}}^T\mathbf{y}_{C_2}\right) \otimes \left({\mathbf{x}_{C_3}}^T\mathbf{y}_{C_3}\right) \qquad = \left({\mathbf{x}_{C_1}}^T\mathbf{y}_{C_1}\right) \left({\mathbf{x}_{C_2}}^T\mathbf{y}_{C_2}\right) \left({\mathbf{x}_{C_3}}^T\mathbf{y}_{C_3}\right)
\end{aligned} \tag{4.4}$$

Thus, the computational complexity of the dot product is $O(K)$ given the previous assumptions hold. This also allows us to calculate the norm of a factored vector which is required in Step 11 of the algorithm.

Algorithm 4.1 Factored Arnoldi iteration(A, $\mathbf{q}_1$)

1: Start with an arbitrary *factored* vector $\mathbf{q}_1$ with norm 1
2: **for** i = 2 to r **do**
3: $\quad \mathbf{q}_i \leftarrow Project\&Factorize(D, \mathbf{q}_{i-1})$
4: **end for**
5: **for** i = 2 to r **do**
6: $\quad$ **for** j = 1 to i-1 **do**
7: $\quad\quad h_{j,i-1} \leftarrow \mathbf{q}_j^* \mathbf{q}_i$
8: $\quad\quad \mathbf{q}_i \leftarrow \mathbf{q}_i - h_{j,i-1}\mathbf{q}_j$
9: $\quad$ **end for**
10: $\quad \mathbf{q}_i \leftarrow Compress\,(\mathbf{q}_i)$
11: $\quad h_{i,i-1} \leftarrow ||q_i||$
12: $\quad \mathbf{q}_i \leftarrow \frac{\mathbf{q}_i}{h_{i,i-1}}$
13: **end for**

Sum of two Kronecker product vectors

The next important operation is the summation (or subtraction) of two Kronecker product vectors, as required in Step 11 of Algorithm 4.1. Firstly, it is not possible to perform this addition exactly without creating the full vector (which requires $O(2^K)$ operations). Secondly, the exact result of this operation cannot, in the general case, be factorized into the same clusters. As a result, we have to resort to some sort of approximation.

The problem of approximating the sum of two (or more) Kronecker product vectors (or matrices) has been studied in great detail (Loan & Pitsianis, 1992). The problem is posed as minimizing an objective function of the form:

$$\phi_{\mathbf{q}}(\mathbf{x}^{1:r}, \mathbf{y}^{1:r}) = ||\mathbf{q}_1 \otimes \mathbf{q}_2 - \sum_{i=1}^{r} \mathbf{x}^i \otimes \mathbf{y}^i||^2 \tag{4.5}$$

where $\mathbf{x}^i$ and $\mathbf{y}^i$ are $m_1 \times 1$ and $m_2 \times 1$ vectors (for all i), while $\mathbf{q}$ is a Kronecker product of $\mathbf{q}^1$ and $\mathbf{q}^2$, which have the same dimensions as $\mathbf{x}^i$ and $\mathbf{y}^i$ respectively.

The authors prove that the optimal form of $\mathbf{q}^1$ is a linear combination of the $\mathbf{x}^i$ vectors:

$$\mathbf{q}^1 = \sum_{i=1}^{r} \alpha_i \mathbf{x}^i \tag{4.6}$$

The same relationship holds true for $\mathbf{q}^2$ and the $\mathbf{y}^i$ vectors as well.

This minimization problem can now be solved using a gradient descent method. Let us break down the objective function:

$$\phi_{\mathbf{q}}(\mathbf{x}^{1:r}, \mathbf{y}^{1:r}) = \sum_{j=1}^{m_1} \sum_{k=1}^{m_2} \left(q_j^1 q_k^2 - \sum_{i=1}^{r} x_j^i y_k^i \right)^2 \tag{4.7}$$

Substituting $\mathbf{q}^1$ and $\mathbf{q}^2$ as linear combinations of $\mathbf{x}^{1:r}$ and $\mathbf{y}^{1:r}$ respectively, the objective function reduces to:

$$\phi_{\alpha,\beta}(\mathbf{x}^{1:r}, \mathbf{y}^{1:r}) = \sum_{j=1}^{m_1} \sum_{k=1}^{m_2} \left(\left(\sum_{i=1}^{r} \alpha_i x_j^i \sum_{i=1}^{r} \beta_i y_k^i \right) - \sum_{i=1}^{r} x_j^i y_k^i \right)^2 \tag{4.8}$$

Computing the exact gradient of the objective function incurs a computational complexity of $O(m_1 m_2 r)$, which is infeasible as it is equivalent to $O(nr)$ in our setting. Instead, we can resort to a stochastic gradient method, where in each iteration, we approximate the gradient of the objective function using a random subsample of $O(m_1 + m_2)$ of the possible summation terms in the RHS

of Equation 4.8. Computing the approximate gradient via such a method requires $O((m_1 + m_2)r)$ computations (i.e., $O(rK)$ computations in our setting).

Algorithm 4.2 Stochastic gradient descent(α, β, γ, $x^{1:r}$, $y^{1:r}$)

1: Choose an initial vector $\langle \alpha, \beta \rangle$ and a learning rate γ
2: **while** an approximate minimum is not obtained **do**
3: $\quad \mathcal{P} \leftarrow$ random $m_1 + m_2$ pairs of (j, k) s.t. $1 \leq j \leq m_1$, $1 \leq k \leq m_2$
4: $\quad \hat{\phi}(\mathcal{P}) \leftarrow \sum_{(j,k)\in\mathcal{P}} \left(\left(\sum_{i=1}^{r} \alpha_i x_j^i \sum_{i=1}^{r} \beta_i y_k^i \right) - \sum_{i=1}^{r} x_j^i y_k^i \right)^2$
5: $\quad$ Compute $\bigtriangledown_{\alpha,\beta} \hat{\phi}(\mathcal{P})$
6: $\quad \langle \alpha, \beta \rangle \leftarrow \langle \alpha, \beta \rangle - \gamma \bigtriangledown_{\alpha,\beta} \hat{\phi}$
7: **end while**

However, it is not necessary to apply this approximation as soon as we encounter an addition (or subtraction) operation. One possibility is to maintain the multiple Kronecker product vectors and their coefficients (and perform the summation later). Thus, we do not perform any addition in Step 8 – instead, we save the coefficients of each of the $\mathbf{q}_j$ vectors in the current value of $\mathbf{q}_i$. The reason for this delayed addition, is to achieve a higher level of accuracy in the orthogonalization process. As a result, the complexity of step 7 becomes $O(jK)$ in the j^{th} iteration and the complexity of each iteration of the i loop becomes $O(i^2K)$.

Once, we have computed q_i in this manner, we can apply the approximate sum operation to reduce the sum to a single Kronecker product vector (with the same set of clusters).

Step 3: Find eigenvalues and eigenvectors

The r most extreme eigenvalues of A can be approximated with the eigenvalues of the upper Hessenberg coefficient matrix H from algorithm 4.1. We can use the QR algorithm or one of its variants to find the eigenvalues and corresponding eigenvectors in $O(r^3)$, which is very tractable since $r \ll n$. The eigenvector of A corresponding to the eigenvalue λ_i can be computed in their factored form by:

$$\mathbf{v}_i = Q_r \mathbf{u}_i \tag{4.9}$$

where $\mathbf{u}_i$ is the corresponding eigenvector of λ_i in H_r and Q_r is the transformation matrix whose column vectors are $\mathbf{q}_1, \cdots, \mathbf{q}_r$. Hence,

$$\mathbf{v}_i = u_{i,1}\mathbf{q}_1 + u_{i,2}\mathbf{q}_2 + \cdots + u_{i,r}\mathbf{q}_r \tag{4.10}$$

This is again another summation of Kronecker product vectors and can be approximated as described in Section 4.4. This completes our overall eigenvalue algorithm.

4.5 Experiments

In this section, we will outline a set of experiments that we performed to demonstrate the accuracy and effectiveness of our approximate eigen-pair algorithm. More specifically, we want to show:

1. Empirical convergence rates of the stochastic gradient descent with increasing problem size
2. Quality of approximation with increasing problem size
3. Quality of approximation with increasing amount of near-determinism

Data generation

Without any loss of generalization, we will assume that all variables are binary. We generate a 2-TBN with n variables in each time-slice such that there are no cycles. We set the maximum number of parents for any node to be 3 to ensure sparse factors. The CPTs are generated with varying degrees of near-determinism, based on the requirements of a specific experiment. The amount of near-determinism is quantified by a parameter ϵ, where $p(X_i^t|\pi(X_i^t))$ is $o(\epsilon)$ when $X_i^t \neq X_i^{t-1}$.

Implementation details

We used the junction tree implementation of the LIBDAI library (Mooij, 2010). We implemented the rest of the algorithm (the Arnoldi iteraton framework, the stochastic gradient descent) in Python. For the QR algorithm and to compute the exact eigen-pairs for problem instances, we used the *linalg* module of the *numpy* library in Python.

Results

We define the problem size as the number of binary variables in each time slice of the DBN. Hence for a problem size of n, there are 2^n states in the system. We solve for k eigen-pairs using our approximate algorithm and compare them with the k dominant (maximum norm) eigenvalues and their corresponding eigenvectors solved by using a regular eigen-analysis algorithm (we use the *linalg* module of the *numpy* library in Python.

Stochastic gradient descent

We implemented a stochastic gradient descent with restarts (where the restart was triggered if the error did not reduce by at least $0.1 \times threshold$ over 100 iterations). The threshold is set based upon the norm of the subsampled vector (comprising of the coordinates sampled in the current iteration).

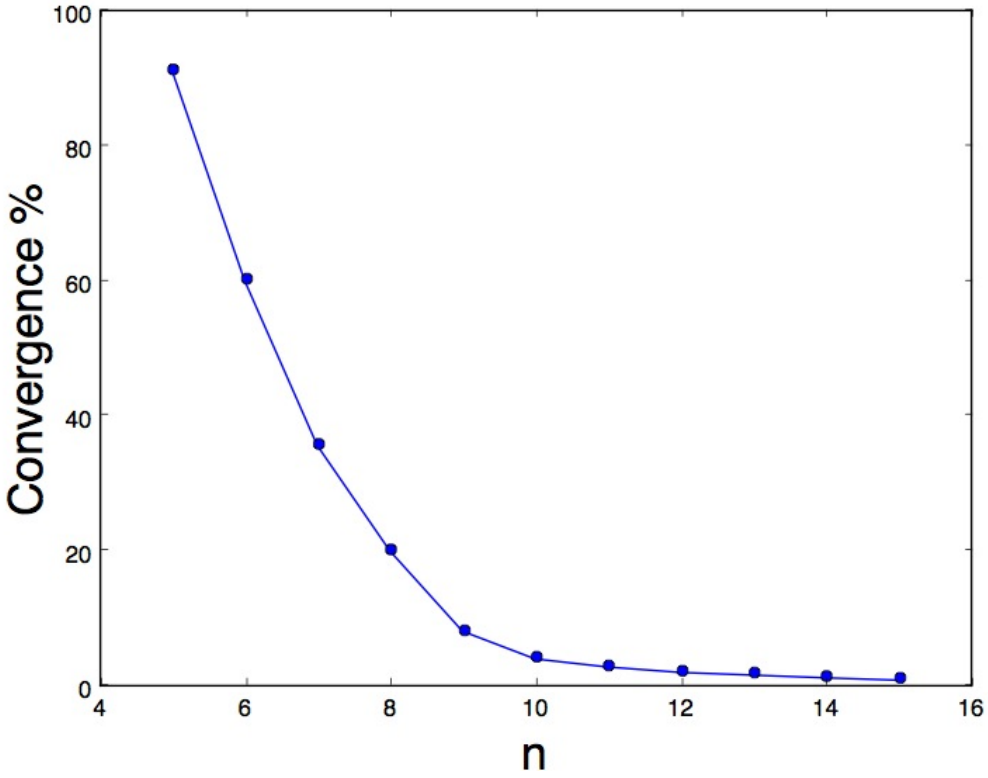


Figure 4.2: The percentage of examples where the stochastic gradient converged. For more deterministic examples, a greater percentage of examples converged.

Unfortunately, the stochastic gradient descent often did not converge to an error which was below the desired threshold. We set a time-out of 5 minutes. The percentage of cases where the stochastic gradient converged falls sharply with increase in the size of the problem as shown in Figure 4.2.

The rest of the results are for instances where the stochastic gradient descent converged in every iteration. We will also focus on the first 5 iterations (and therefore, 5 eigen-pairs).

Quality of approximation

In order to determine the accuracy of the approximate eigenvalues and eigenvectors, we first match the top k approximate eigenvalues to their nearest exact eigenvalue. A sample matching is shown in Figure 4.3, where the circles correspond to the exact eigenvalues and triangles correspond to approximate eigenvalues computed using our algorithm. Then, we look at the L2-norm of the difference vector for the corresponding eigenvectors.

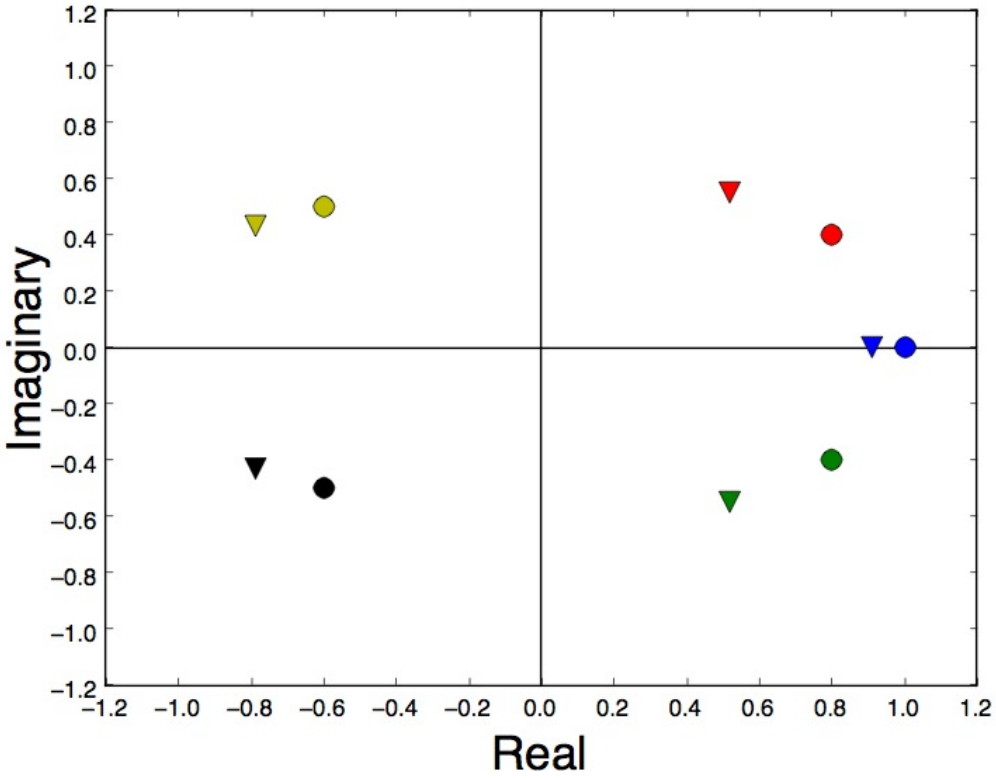


Figure 4.3: Matching of exact and approximate eigenvalues.

Eigenvalues Figure 4.4 shows the root mean squared error (RMSE) between the exact eigenvalues and their matched approximate eigenvalues. The exact eigenvalues are computed using the *linalg* module in the *scipy* package in Python. As the figure shows, the approximation quality improves as the problem size decreases and also as the amount of near-determinism increases.

Eigenvectors Figure 4.5 shows the average L2-norm of the difference vector between the exact and approximate eigenvectors. The average is taken over 10 different runs. As the amount of near-determinism increases, the accuracy increases and the average L2-norm decreases. Conversely, with decreasing near-determinism, the accuracy decreases.

4.6 Discussion

The current approach has two main drawbacks:

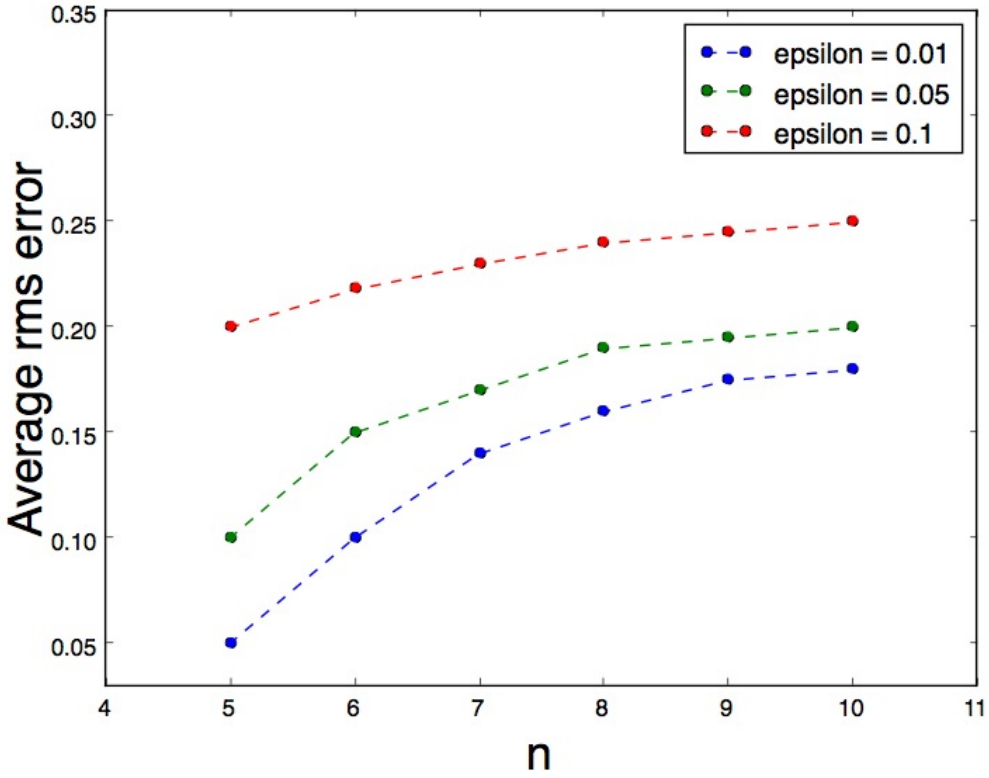


Figure 4.4: The RMSE of the approximate eigenvalues.

Linear projection of factored belief vector In the current setup, we project the factored belief vectors at the outset using the junction tree algorithm, since the orthogonalization process (e.g., the Arnoldi iteration) results in negative elements in the factored belief vector (or the belief vector in general, even if we are working in a non-factored setting).

Stochastic gradient descent The exact summation of the Kronecker product vectors is not feasible. Hence, we use the stochastic gradient descent to find an approximation to the sum. However, our experiments suggest that it is often hard to converge to a good solution with an error threshold small enough to maintain accuracy of the eigenvalue algorithm.

Because of the two above sources of error, the approximate eigenvalues and eigenvectors computed by our algorithm are often not very accurate. Removing or alleviating either error source can lead to much better approximations.

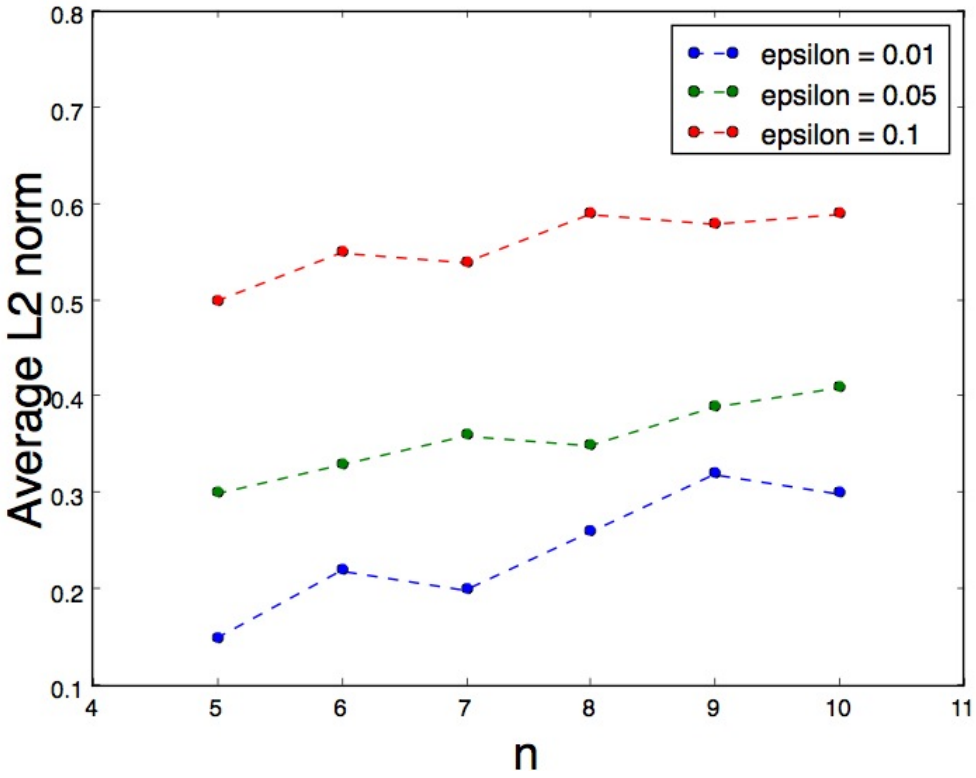


Figure 4.5: The L2 norm of the difference vector between the normalized approximate and exact eigenvector.

Chapter 5

A temporally abstracted Viterbi algorithm

The previous two chapters were focused on creating an abstract model that makes the general inference task easier for temporal near-deterministic systems. We now focus on the maximization problem and devise both an abstraction scheme and an associated algorithm to find the most likely solution in a probabilistic model.

Hierarchical problem abstraction, when applicable, may offer exponential reductions in computational complexity. Previous work on coarse-to-fine dynamic programming (CFDP) has demonstrated this possibility using *state abstraction* to speed up the Viterbi algorithm. In this chapter, we show how to apply *temporal abstraction* to the Viterbi problem. Our algorithm uses bounds derived from analysis of coarse timescales to prune large parts of the state trellis at finer timescales. We demonstrate improvements of several orders of magnitude over the standard Viterbi algorithm, as well as significant speedups over CFDP, for problems whose state variables evolve at widely differing rates.

5.1 Introduction

The Viterbi algorithm (Viterbi, 1967; Forney, 1973), introduced in Section 2.2, finds the most likely sequence of hidden states, called the "Viterbi path," conditioned on a sequence of observations in a hidden Markov model (HMM).

Finding a most-likely state sequence in an HMM is isomorphic to finding a minimum cost path through a state–time *trellis graph* (see Figure 5.1) whose link cost is the negative log probability of the corresponding transition–observation pair in the HMM. Thus, the cost of finding an optimal path can be reduced further using an admissible (lower bound) heuristic and A* graph search.

Even with this improvement, the time and space cost can be prohibitive when N and T are

very large; for example, with a state space defined by 30 Boolean variables, running Viterbi for a million time steps requires 10^{24} computations. One possible approach to handle such problems is to use a *state abstraction*: a mapping $\phi : S_0 \mapsto S_1$ from the original state space S_0 to a coarser state space S_1. For stochastic models (such as an HMM), the parameters of the model in S_1 are often chosen to be the maximum of the corresponding constituent parameters in S_0. Although these parameters do not define a valid probability measure, they serve as admissible heuristics for an A* search. The same idea can be applied to produce a a *hierarchy* of abstractions $S_0, S_1, \ldots, S_L$. Coarse-to-fine dynamic programming or CFDP (Raphael, 2001a) begins with S_L and iteratively finds the shortest path in the current (abstracted) version of the graph and refines along it until the current shortest path is completely refined. Several algorithms—e.g., hierarchical A* (Holte *et al.*, 1996) and HA*LD (Felzenszwalb & McAllester, 2007)—are able to refine only the necessary part of the hierarchy tree and compute heuristics only when needed.

The Viterbi algorithm devotes equal effort to every link in the state–time trellis. CFDP and its relatives can determine that an entire set of states need not be explored in detail, based on bounding the cost of paths through that set; *but they do so separately for each time step*. In this paper, we show how to use temporal abstraction to eliminate large sets of states from consideration *over large periods of time*.

To motivate our algorithm, consider the following problem. We observe Judy's daily tweets describing what she eats for lunch, and wish to infer the city in which she is staying on each day. The state space S_0 is the set of all cities in the world. The abstract space S_1 is the set of countries, and S_2 is the continents. (Figure 5.1 shows a small example.) The transition model suggests that on any given day Judy is unlikely to leave the city she is in, even less likely to leave the country she is in, and very unlikely indeed to travel to another continent. Thus, if Judy had Tandoori chicken on a Thursday but the rest of the week was all hamburgers, then it is most likely that she was in some American city *for the entire week*. However, if she had Tandoori chicken and/or biryani for an entire week, then it is quite possible that she is in India. Our algorithm, temporally abstracted Viterbi (henceforth TAV), facilitates *reasoning over a temporal interval* (like a week or month or longer) and *localized search within those intervals*. Neither of these is possible with Viterbi or state abstraction algorithms like CFDP. The computational savings of TAV on an instance of this problem can be seen in Figure 5.2.

Temporal abstractions have been well-studied in the context of planning (Sutton *et al.*, 1999) and inference in dynamic Bayesian networks (Chatterjee & Russell, 2010). An excellent survey of temporal abstraction for dynamical systems can be found in (Pavliotis & Stuart, 2007). To the best of our knowledge, TAV is the first algorithm to use temporal abstraction for general shortest-path problems. TAV is guaranteed to find the Viterbi path, and does so (for certain problem instances) several orders of magnitude faster than the Viterbi algorithm and one to two orders of magnitude faster than CFDP.

The rest of the paper is organized as follows. Section 5.2 reviews the Viterbi algorithm and CFDP and introduces the notations and definitions used in the rest of the paper. Section 5.3 provides a detailed description of the main algorithm and establishes its correctness. Section 5.4

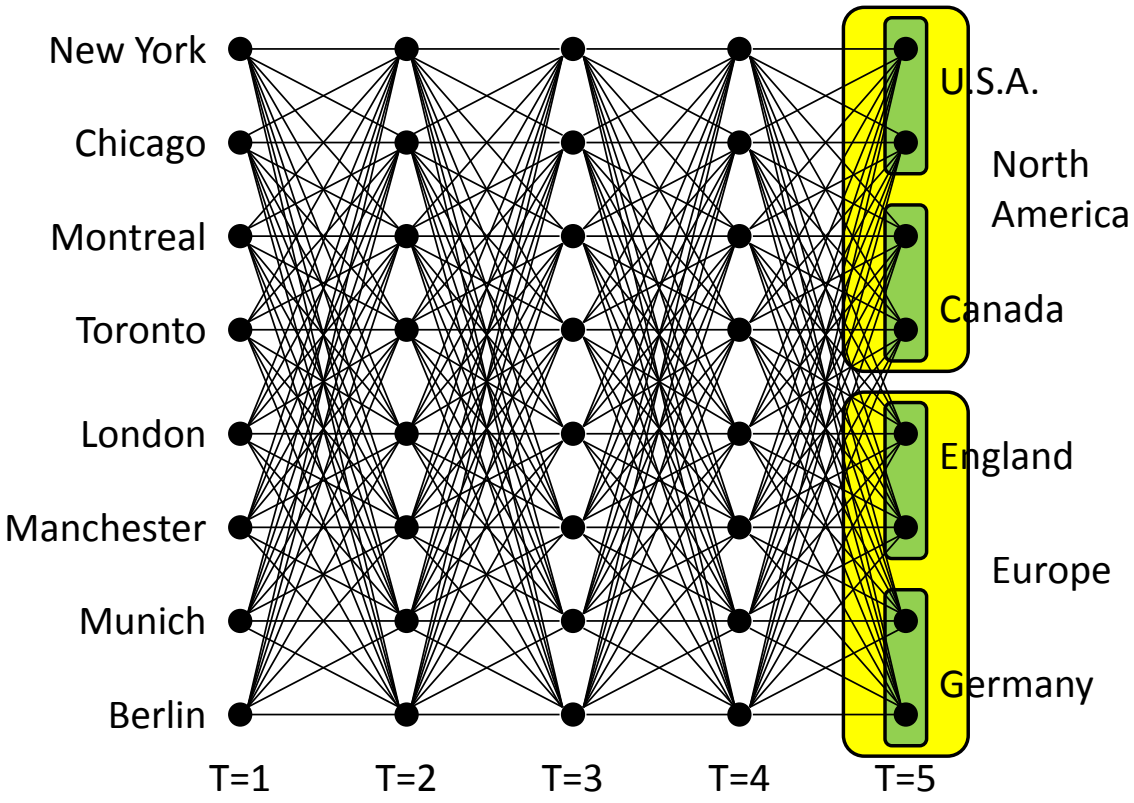


Figure 5.1: The state–time trellis for a small version of the tracking problem. The links have weights denoting probabilities of going from a city A to a city B in a day. The abstract state spaces S_1 (countries depicted in green) and S_2 (continents in yellow) are only shown for T=5 to maintain clarity. The observation links are also omitted for the same reason.

discusses the computation of temporal abstraction heuristics. Section 5.5 presents some empirical results to demonstrate the benefits of TAV while section 5.6 provides some guidance on how to induce abstraction hierarchies.

5.2 Problem Formulation

In coarse-to-fine (a.k.a. hierarchical) approaches, inference is performed in the coarser models to reduce the amount of computation needed in the finer models. Typically, a set of abstract state spaces $\mathcal{S} = \{S_0, S_1, \ldots, S_L\}$ and abstract models $\mathcal{M} = \{M_0, M_1, \ldots, M_L\}$ are defined where S_0

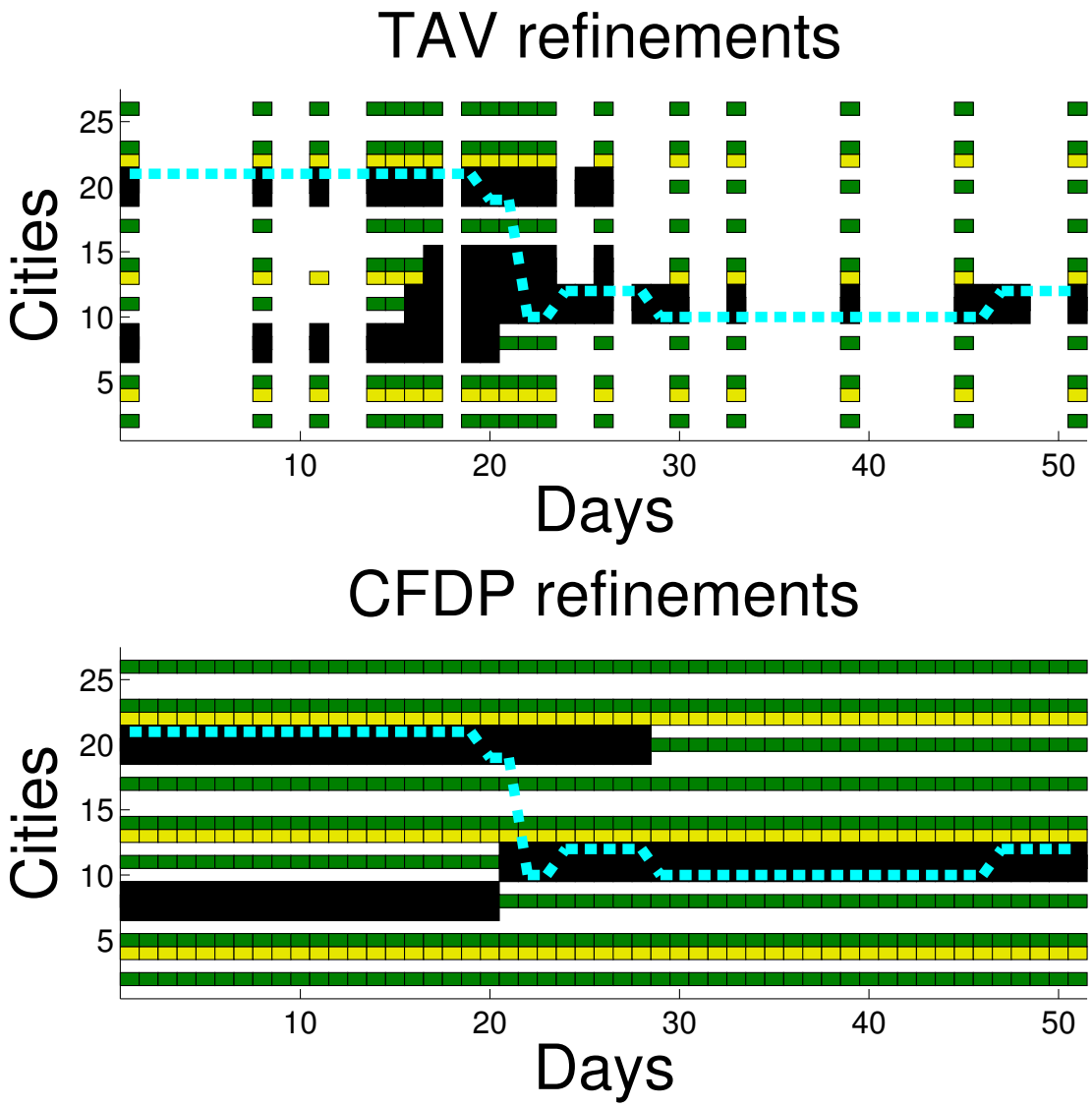


Figure 5.2: A comparison of the performance of CFDP and TAV on the city tracking problem with 27 cities, 9 countries and 3 continents over 50 days. The plots indicate portions of the state–time trellis each algorithm explored. Black, green and yellow squares denote the cities, countries and continents considered during search. The cyan dotted line is the optimal trajectory.

(M_0) is the original state space (model) and S_L (M_L) is the coarsest abstract state space (model). Let the parameters of M_l be denoted by the set $\{A^l, B^l, \Pi^l\}$.

A state in this hierarchy is denoted by s_i^l, where l is the abstraction level and i is its index within level l. N_l denotes the number of states in level l. Let $\phi : S_l \mapsto S_{l+1}$ denote the mapping from any level l to its immediate abstract level $l + 1$. The parameters at level $l + 1$ are defined by taking the

maximum of the component parameters at level l. Thus, $A^{l+1} = \{a_{ij}^{l+1}\}$, where

$$a_{ij}^{l+1} = \max_{p,q} a_{pq}^{l} \qquad s.t. \quad \phi(s_p^l) = s_i^{l+1}, \phi(s_q^l) = s_j^{l+1} \, .$$

B^{l+1} and Π^{l+1} are defined similarly in terms of B^l and Π^l. Any transition/emission probability in an abstract model is a *tight* upper bound on the corresponding probability in its immediate refinement. Hence, the cost (negative log probability) of an abstract trajectory can serve as an admissible heuristic to guide search in a more refined state space.

CFDP works by starting with only the coarsest states $s_{1:N_L}^L$ at every time step from 1 to T. The states in t and $t+1$ are connected by transition links whose values are given by A^L. B^L and Π^L define the other starting parameters. It then iterates between computing the optimal path in the current trellis graph and refining the states (and thereby the associated links) along the current optimal path. The algorithm terminates when the current optimal path contains only completely refined states (i.e., states in S_0). A graphical depiction of how CFDP works is shown in Figure 5.5. The TAV algorithm, which also has an iterative structure, is described in the next section. We will be reusing notation and definitions from this section throughout the rest of the paper.

5.3 Main algorithm

The distinguishing feature of TAV is its ability to reason with *temporally abstract links*. A link in the Viterbi algorithm and CFDP-like approaches describes the transition probability between states over a single time step. A temporally abstract link starting in state s_1 at time t_1 and ending in state s_2 at time $t_2 > t_1$ represents all trajectories having those end points and is denoted by a 4-tuple—$((s_1, t_1), (s_2, t_2))$. $Links((s, t))$ is the set of incoming links to state s at time t. $Children(s^l)$ = $\{s' : s' \in S^{l-1}, \phi(s') = s^l\}$ is the set of children of state s^l in the abstraction hierarchy. We define three different kinds of temporally abstract links:

1. Direct links: $d(s, t_1, t_2)$ represents the set of trajectories that start at (s, t_1) and end in (s, t_2) and *stay within* s for the entire time interval (t_1, t_2).
2. Cross links: $c((s_1, t_1), (s_2, t_2))$ represents the set of trajectories from (s_1, t_1) to (s_2, t_2), when $s_1 \neq s_2$.
3. Re–entry links: $r((s, t_1), (s, t_2))$ represents the set of trajectories that start at (s, t_1) and end in (s, t_2) but *move outside* s *at least once* in the time interval (t_1, t_2). $r((s_1, t_1), (s_1, t_2))$ = $\emptyset$ when $t_2 - t_1 \leq 1$.

The direct and cross links are denoted graphically by straight lines, whereas the re-entry links are represented by curved lines as shown in Figure 5.3. A generic link is denoted by the symbol k. The (heuristic) score of a temporally abstract link has to be an upper bound on the probability of all

trajectories in the set of trajectories it represents. Computing admissible and monotone heuristics will be discussed in Section 5.4.

Our algorithm's computational savings over spatial abstraction schemes come from two avenues—first, fewer time points to consider using temporal abstraction; second, fewer states to reason about by considering *constrained trajectories* using direct links. Although the general flow of the algorithm is similar to CFDP, the refinement constructions are different. The algorithm descriptions provided omit details about observation matrix computations since they are standard. However, the issue is revisited in Section 5.4 to focus on some subtleties. The correctness of the algorithm depends only on the admissibility of the heuristics.

Refinement constructions

A refinement of a temporally abstract link replaces the original link with a set of refined links that represent a *partition*—a mutually exclusive and exhaustive decomposition—of the set of trajectories represented by the original link. The refinement allows us to reason about subsets of the original set of trajectories separately and thereby potentially narrow down on a single optimal trajectory. There are two different kinds of refinement constructions.

Spatial refinement

When a direct link, $d(s^l, t_1, t_2)$, lies on the optimal path, the natural thing to do is to refine (partition) the set of trajectories it represents. The original direct link is *replaced* with all possible cross, direct and re-entry links between $Children(s^l)$ at t_1 and t_2. This is depicted graphically in Figure 5.3. A link is also refined spatially if its time span $(t_2 - t_1)$ is 1 time step since temporal refinement is not a possibility. The pseudocode for spatial refinement (see Algorithm 5.1) provides all the necessary details. It is trivial to show that the new links constitute a partition of the trajectories represented by the original link.

Temporal refinement

When a cross link $c((s_1, t_1), (s_2, t_2))$ or a re-entry link $r((s_1, t_1), (s_1, t_2))$ is selected for refinement, we are faced with the task of refining a set of trajectories that *do not stay in the same state* for the abstraction interval. This is a case where temporal abstraction is not helping (not at the current resolution at least).

An example of temporal refinement, which is only invoked when $t_2 - t_1 > 1$, is shown in Figure 5.4. It results in splitting the time interval (t_1, t_2) into two sub-intervals that together span the original interval. Let us assume that we select the (rounded off) mid-point of the interval. When a link is temporally refined , we temporally split all cross, re-entry and direct links spanning the interval (t_1, t_2) between states in $Children(\phi(s_1))$. We will show later that for any link longer

Algorithm 5.1 Spatial Refinement((p_1, t_1, p_2, t_2))

$C \leftarrow Children(p_1); D \leftarrow Children(p_2)$
if $t_2 - t_1 > 1$ **then**
  $Links(p_1, t_2) \leftarrow Links(p_1, t_2) \setminus d(p_1, t_1, t_2)$
else
  $Links(p_2, t_2) \leftarrow Links(p_2, t_2) \setminus k((p_1, t_1), (p_2, t_2))$
end if
$usedStates(t_1) \leftarrow usedStates(t_1) \cup C$
$usedStates(t_2) \leftarrow usedStates(t_2) \cup D$
for all $s \in D$ **do**
  $Links(s, t_2) \leftarrow Links(s, t_2) \cup d(s, t_1, t_2)$
  for all $s' \in C$ **do**
    if $s = s'$ **then**
      $Links(s, t_2) \leftarrow Links(s, t_2) \cup r((s, t_1), (s, t_2))$
    else
      $Links(s, t_2) \leftarrow Links(s, t_2) \cup c((s', t_1), (s, t_2))$
    end if
  end for
end for

than 1 time step, $\phi(s_1) = \phi(s_2)$. It should be noted that re-entry links are only added when the sub–interval length is longer than 1 time step. Also, if $t_2 - t_1 = 1$, then a cross link is *spatially refined* (analogous to CFDP).

One possibility is that some of the direct links for states in $Children(\phi(s_1))$ between (t_1, t_2) were already spatially refined. In that case, we apply temporal refinement recursively to the spatially refined links of those direct links. *The choice of the splitting point does not affect the correctness of the algorithm as long as the split is replicated in the instantiated portion of the state space tree rooted at* $\phi(s_1)$. The pseudocode (shown in Algorithm 5.2) provides details of this procedure.

Lemma 5.3.1 *The sets of trajectories represented by links before and after any spatial or temporal refinement are the same. Also, every trajectory is represented by exactly one temporally abstract path.*

Lemma 5.3.2 *Any link created by TAV will always be between two states at the same level of abstraction. If the time span of the link is greater than* 1 *time step, then those two states will also have the same parent at all coarser levels of abstraction.*

Proof The original links are all between states of level L. Both refinement constructions add links only between states at the same abstraction level. This proves the first statement. Moreover, upon

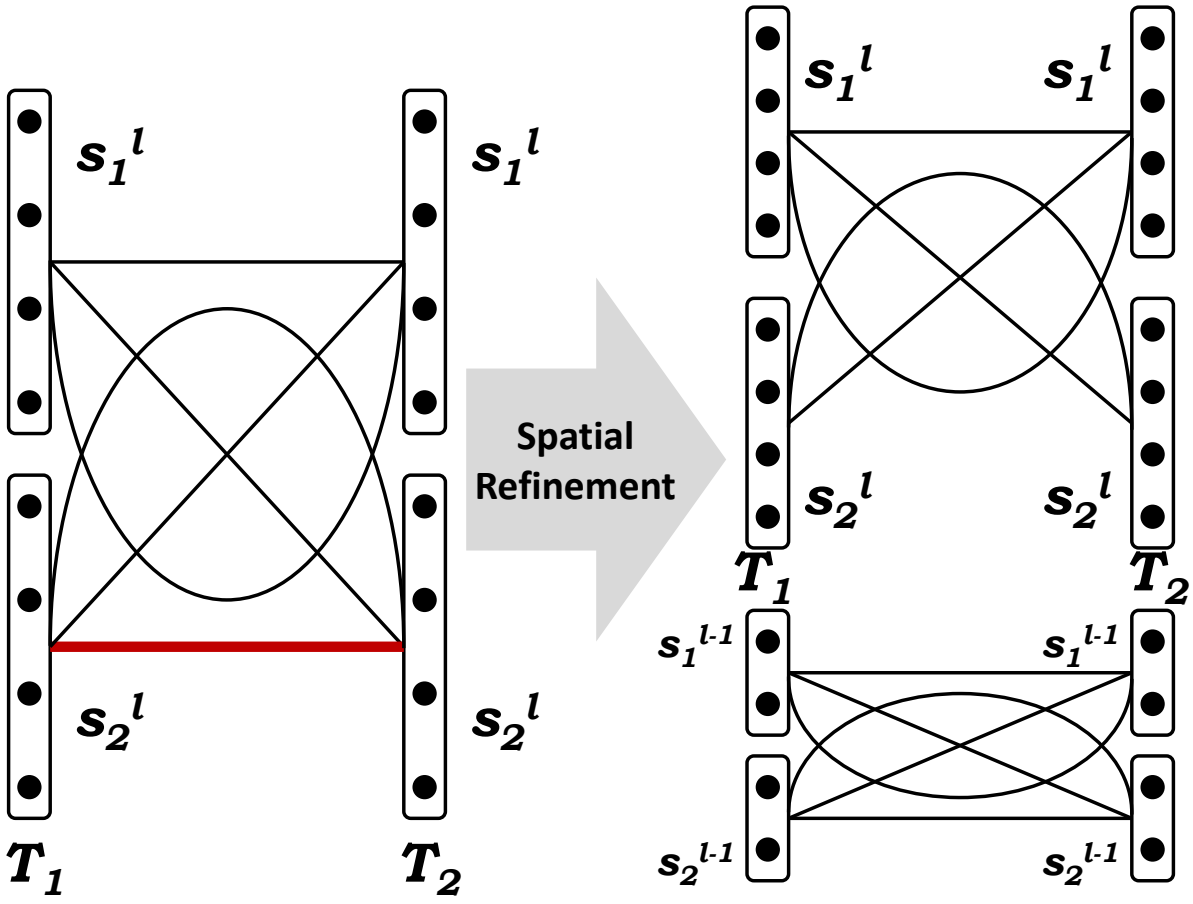


Figure 5.3: Spatial refinement: The optimal link, shown in bright red, is a direct link and is replaced with all possible links between its children.

initialization, there is no coarser level of abstraction—hence the second part of the statement is vacuously true. Temporal refinement always considers links between descendants of the $parent$ node. Spatial refinement also adds links between $Children(s_l)$ of a state s^l except when the time step is 1. This proves the second statement. □

Modified Viterbi algorithm

It is possible to have links to a state s and to its abstraction $\phi(s)$ at the same time step t (see Figure 5.5b). This was not possible in CFDP. Hence, we need a slightly modified scoring and backtracking scheme. $\delta_t(s)$ is the best score of a trajectory ending in state s at time t and $\psi_t(s)$ contains the temporally abstract link's information which connects (s, t) to its predecessor. $usedTimes$ is a sorted list of time steps which have links to or from it. $usedStates(t)$ is the set of

Algorithm 5.2 Temporal Refinement($(parent, t_1, t_2)$)

$nT \leftarrow \lceil (t_1 + t_2)/2 \rceil$
if $parent \in usedStates(nT)$ **then**
  return
end if
$usedTimes \leftarrow usedTimes \cup nT$
$C \leftarrow Children(parent)$
$usedStates(nT) \leftarrow usedStates(nT) \cup C$
for all $s \in C$ **do**
  if $d(s, t_1, t_2) \notin Links(s, t_2)$ **then**
    $Temporal\_Refinement(s, t_1, t_2)$
    $Links(s, t_2) \leftarrow \emptyset$
    $Links(s, nT) \leftarrow \emptyset$
  else
    $Links(s, t_2) \leftarrow \{d(s, nT, t_2)\}$
    $Links(s, nT) \leftarrow \{d(s, t_1, nT)\}$
  end if
  for all $s' \in C$ **do**
    $Links(s, t_2) \leftarrow Links(s, t_2) \setminus k((s', t_1), (s, t_2))$
    $Links(s, t_2) \leftarrow Links(s, t_2) \cup k((s', nT), (s, t_2))$
    $Links(s, nT) \leftarrow Links(s, nT) \cup k((s', t_1), (s, nT))$
  end for
end for

nodes at time t which have incoming or outgoing links. The score computation algorithm moves forward in time like the normal Viterbi algorithm. The score computation (at each used time step t) is done in 3 phases. The pseudocode is given in Algorithm 5.3.

1. $\delta_t(s)$ is computed using the best of its incoming links, $Links(s, t)$ and $\psi_t(s)$ points to that link.
2. Starting at level $L - 1$ and going down to level 0, a state s gets its parent's ($\phi(s)$) score and backpointer if $\phi(s)$ has a higher score.
3. Starting from level 0 and going up to level $L - 1$, a state s's parent $\phi(s)$ gets its child's score and backpointer if s has a higher score

Theorem 5.3.3 *The score $\delta_t(s)$ computed by the $BestPath$ procedure is a* strict upper bound *on all trajectories ending in state s at time t given the current abstracted version of the state-time trellis.*

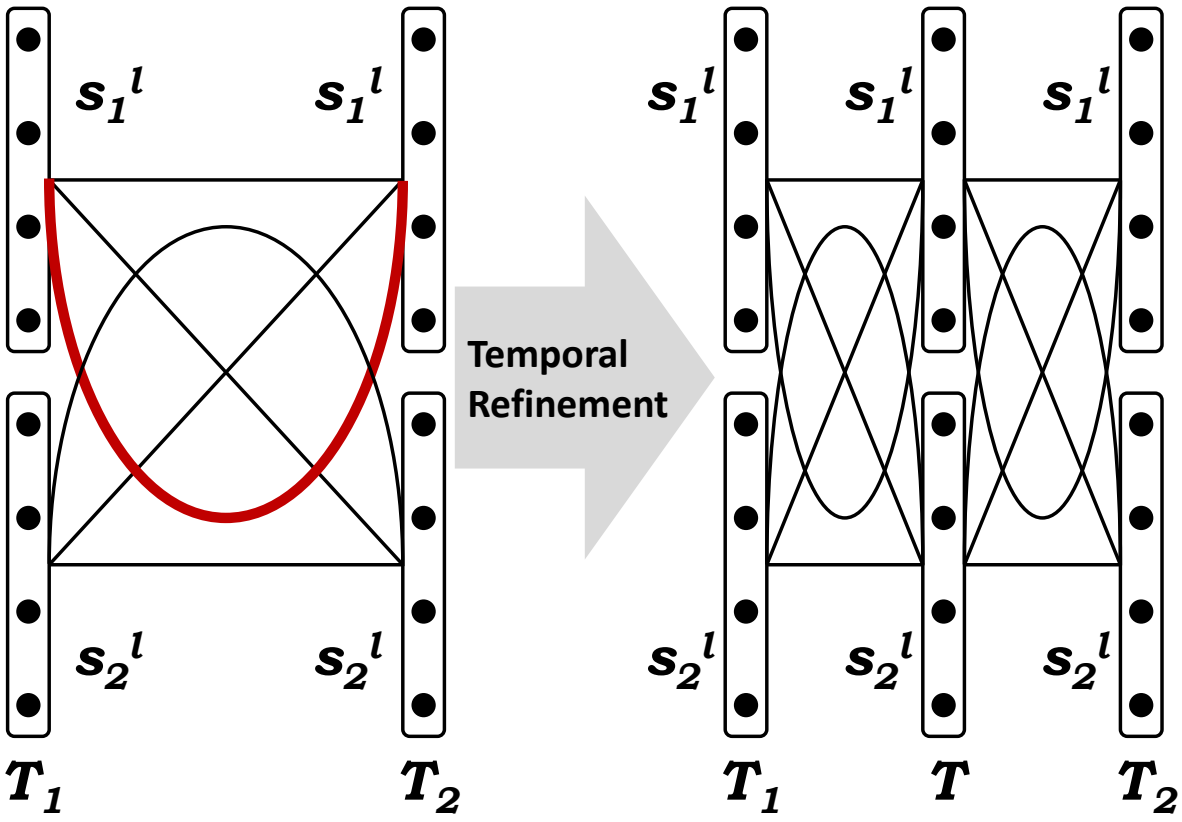


Figure 5.4: Temporal refinement: When refining a cross or re-entry link, refine all links between nodes that have the same parent as the nodes of the selected link.

Proof Any trajectory ending in a state $\hat{s}$, which is neither an ancestor nor a descendant of s, does not include any trajectory to s. Hence, $BestPath$ computes an upper bound on the score of the best trajectory ending in state s at time t.

For the bound to be strict, it is sufficient to show that each phase of $BestPath$ only considers scores of such nodes where every incoming link includes at least one trajectory to s. The first phase accounts for all the incoming links to node s itself. Let $\phi^*(s)$ denote an ancestor of s. Any incoming link (direct, cross or re–entry) to $\phi^*(s)$ includes at least one trajectory to s. This necessitates taking the maximum over $\delta_t(\phi^*(s))$ (step 2). Finally, any trajectory ending in state s', where s' is a descendant of s, is by definition, a trajectory ending in s. Hence, the upper bound is strict. □

The ordering of the phases is important to perform the desired computation correctly and effi-

Algorithm 5.3 BestPath($Links, usedStates, usedTimes$)

```
    curTime ← 1
    while curTime < T do
      curTime ← nextUsedTime(curTime, usedTimes)
      for all s ∈ UsedStates(curTime) do
 5:     δ_t(s) ← MaxOverLinks(Links((s, curTime)), δ)
        ψ_t(s) ← ArgMaxOverLinks(Links((s, curTime)), δ)
      end for
      for level = L − 1 to 0 do
        for all s ∈ S^level && s ∈ usedStates(curTime) do
10:       if δ_t(φ(s)) > δ_t(s) then
            δ_t(s) ← δ_t(φ(s))
            ψ_t(s) ← ψ_t(φ(s))
          end if
        end for
15:   end for
      for level = 0 to L-1 do
        for all s ∈ S^level && s ∈ usedStates(curTime) do
          if δ_t(φ(s)) ≤ δ_t(s) then
            δ_t(φ(s)) ← δ_t(s)
20:         ψ_t(φ(s)) ← ψ_t(s)
          end if
        end for
      end for
    end while
25: s* ← argmax_{s∈usedStates(T)} δ_T(s)
    curTime ← 1;   Path ← ∅
    while curTime > 1 do
      Path ← Path ∪ ψ_curTime(s*)
      (s*, curTime) ← ψ_curTime(s*)
30: end while
    return Path
```

ciently.

Complete algorithm

The algorithmic structure of TAV and CFDP are quite similar. The complete specification of TAV is presented in Algorithm 5.4. CFDP has a different initialization and refinement is node–based (TAV is link–based) which introduces links between states at different levels of abstraction. The two initializations are shown in Figure 5.5a. CFDP's initial configuration has no temporally abstract links. The algorithm iterates between two stages: computing the optimal path in the current graph and refining links along the current optimal path. A few steps of execution of the two algorithms are shown on an example in Figure 5.5.

The correctness of the algorithm follows from the optimality of A* search and Lemma 5.3.1

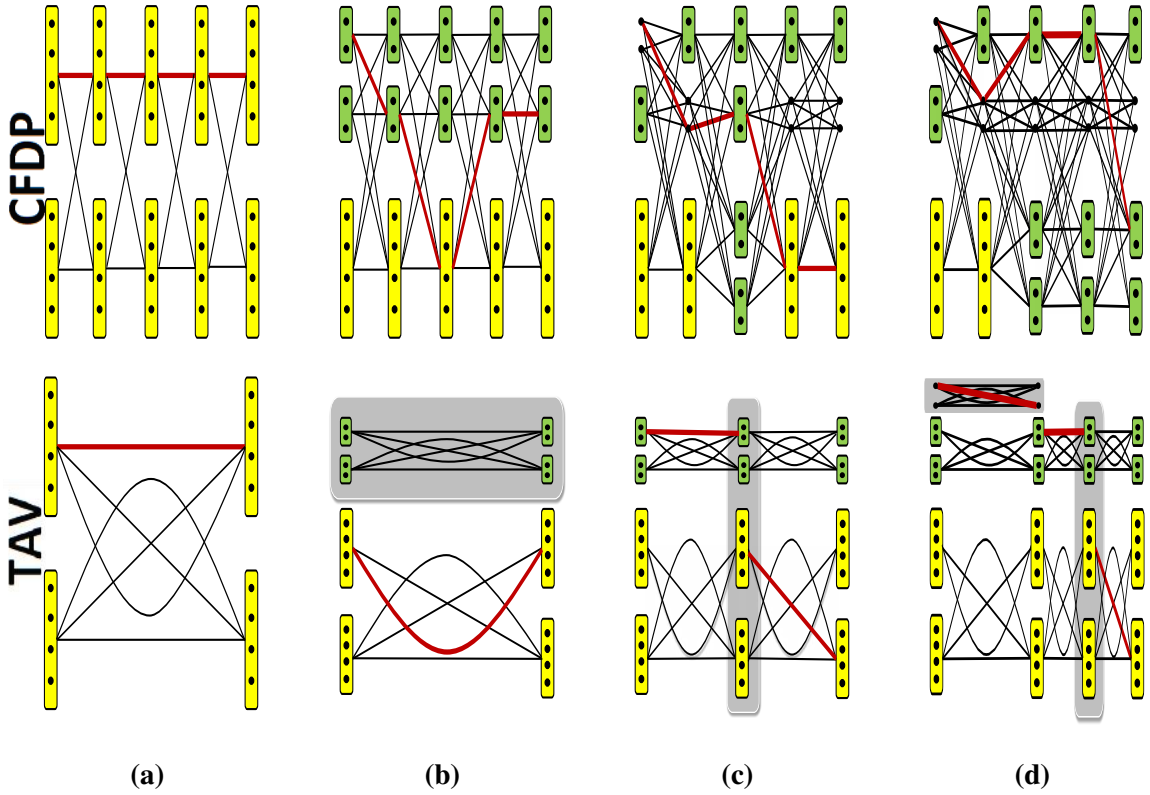


Figure 5.5: Sample run: TAV: a Initialization. The optimal path is a direct link—hence spatial refinement. The new additions are shadowed. b A re-entry link is optimal—hence temporal refinement. Since one direct link among siblings was already refined in Step 1, we also temporally refine the spatially refined component. c The optimal path has links at different levels of abstraction. Such scenarios necessitate the $BestPath$ procedure. d More recursive temporal refinement is performed. Note the difference in the numbers of links in the two graphs after 3 iterations.

and Theorem 5.3.3.

5.4 Heuristics for temporal abstraction

In hierarchical state abstraction schemes, computing heuristics involves taking the maximum of a set of single time step transition probabilities. As mentioned in Section 5.2, this can be done by hierarchically constructing A_l, B_l and Π_l. For temporal abstractions however, there are more design choices to be made when it comes to computing heuristic scores of links. There is a more significant tradeoff between cost of computation and quality of heuristic.

The heuristic score of a direct link is very easy to compute. We do not have to select between possible state transitions. Thus, the heuristic for a link spanning the interval (t_1, t_2) can be done in $O(t_2 - t_1)$, since we still need to account for all the observations in that interval. If the score is

Algorithm 5.4 TAV($A, B, \Pi, \phi, Y_{1:T}$)

$\delta_1(.) \leftarrow ScoreInitialization(\Pi)$
$usedStates(1) = usedStates(T) = S^L$
$usedTimes = \{1, T\}$
for all $s \in S^L$ **do**
  $Links(s, T) \leftarrow d(s, 1, T)$
  for all $s' \in S^L$ **do**
    $Links(s, T) \leftarrow Links(s, T) \cup k((s', 1), (s, T))$
  end for
end for
$ViterbiPathFound \leftarrow 0$
while $ViterbiPathFound = 0$ **do**
  $Path = BestPath(Links, usedStates, usedTimes)$
  $ViterbiPathFound = 1$
  for all $k \in Path$ **do**
    $((s_1, t_1), (s_2, t_2)) \leftarrow details(k)$
    if $level(s_1) > 1 \,||\, \neg isDirect(k)$ **then**
      $ViterbiPathFound = 0$
    end if
    if $isDirect(k) \,||\, t_2 - t_1 = 1$ **then**
      $Spatial\ Refinement(k)$
    else
      $Temporal\ Refinement(k)$
    end if
  end for
end while

cached, the score for any sub-interval is computable in $O(1)$ time.

Cross links and re–entry links require further consideration. A somewhat expensive option is to compute the Viterbi path in the restricted scope. As Lemma 5.3.2 shows, a cross or a re–entry link represents trajectories that can switch between sibling states (the ones which map to the same parent via ϕ). In an abstraction hierarchy, if the cardinality of $Children(s)$ for any state s is restricted to some constant C, then computing this heuristic will require $O(C^2(t_2 - t_1))$ time.

A computationally cheaper but relatively loose heuristic is the following:

$$h((s_i, t_1), (s_j, t_2)) = \max_k \hat{A}_{ik} \max_{p,q} \hat{A}_{pq}^{t_2 - t_1 - 2} \max_k \hat{A}_{kj} \prod_t \max_k \hat{B}_{kY_t}$$

$\hat{A}$ and $\hat{B}$ represent the transition matrices for the set $Children(\phi(s_i))$. This heuristic chooses the best possible transition at every time step other than the two end points and also the best possible observation probability. Its computational complexity is $O(C^2 + (t_2 - t_1))$. Caching values help in both cases. The Viterbi heuristic, being tighter, leads to fewer iterations but needs more computation time. We will compare the two heuristics in our experiments.

5.5 Experiments

The simulations we performed were aimed at showing the benefit of TAV over Viterbi and CFDP. The benefits are magnified in systems where variables evolve at widely varying timescales. The timescale of a random variable is the expected number of time steps in which it changes state. A person's continental location would have a very large timescale, whereas his zip code location would have a much smaller timescale.

A natural way to generate transition matrices with timescale separation is to use a dynamic Bayesian network (DBN). We consider a DBN with n variables, each of which has a cardinality of k. Hence, the state space size N is k^n. We used fully connected DBNs in our simulations. Our observation matrix was multimodal (hence somewhat informative). A DBN with a parameter ϵ means that the timescales of successive variables have a ratio of ϵ. The fastest variable's timescale is $1/\epsilon$ and the slowest variable's $(1/\epsilon)^k$.

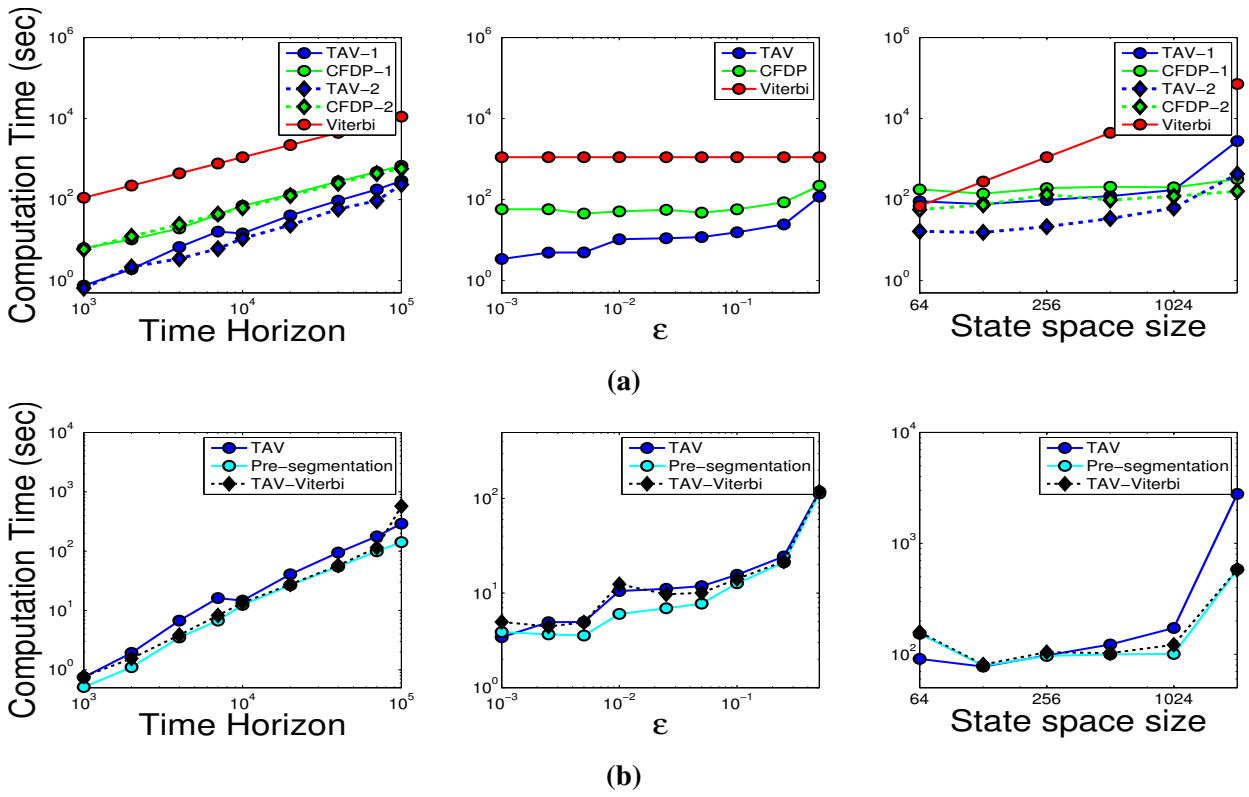


Figure 5.6: Simulation results: a The computation time of Viterbi, CFDP and TAV with varying T (left), ϵ (middle) and N (right). b The computation time of TAV and its two extensions—pre-segmentation and using the Viterbi heuristic—with varying T (left), ϵ (middle) and N (right).

The state space hierarchy at the most abstract level arises from the branching of the slowest variable. In each subtree, we branch on the next slowest variable. In the experiments in this section we assume that the abstraction hierarchy is given to us.

Varying T, N and ϵ

To study the effect of increasing T on the computation time, we generated 2 sequences of length 100000 with $\epsilon = .1$ (case 1) and $.05$ (case 2). N was 256 and the abstraction hierarchy was a binary tree. For each sequence, we found the Viterbi path for the first T timesteps using TAV, CFDP and the Viterbi algorithm. The results are shown in Figure 5.6a. TAV's computational complexity is marginally super-linear. This is because TAV might need to search in the interval $[0, T]$ even if it had found the Viterbi sequence in that interval as new observations (after time T) come in. For the ϵ values used, TAV is more than 2 orders of magnitude faster than Viterbi and 1 order of magnitude faster than CFDP.

It is intuitive that TAV will benefit more from a smaller ϵ (i.e., a larger timescale). As Figure 5.6a shows, CFDP also benefits from the timescale artifact. However, TAV's gains are larger and also grow more quickly with diminishing ϵ. This set of experiments had $N = 256$ and $T = 10000$.

Finally, to check the effect of the state space size, we chose $\epsilon = .5$ (case 1) and $.25$ (case 2). We chose fast timescales to show the limitations of TAV (having already demonstrated its benefits for small ϵ). As Figure 5.6a shows, TAV is 3x to 10x faster than CFDP for $N < 1024$. However, CFDP is about 6x faster than TAV for $N = 2048$. In this case, TAV performs poorly because of its initialization as a single interval. For fast timescales, the first few refinements in this setting are invariably temporal and these refinements can be computationally very expensive.

$A\ priori$ temporal refinement

If T is comparable to the timescale of the slowest variable, then one or more temporal refinements at the very outset of TAV is very likely. Performing these refinements $a\ priori$ will be beneficial if TAV actually had to perform those refinements. The benefit would be proportional to the cost of deciding whether to refine or not. This decision cost increases with T and N (but does not depend on ϵ).

Figure 5.6b shows the computation time for cases where we initialized TAV with 20 equal segments. The $a\ priori$ refinement time was also included in the "Pre-Segmentation" time. Speedups of 2x to 6x were obtained for varying values of N and T. When only ϵ was varied, the benefit was approximately constant (between 3 to 6 seconds of computation time). This resulted in effective speedups only for the smaller values of ϵ, which had small computation times.

Impact of heuristics

As discussed in Section 5.4, there is a trade-off in computing heuristics between accuracy and computation time. Figure 5.6b compares the effect of using the Viterbi heuristic instead of the cheap heuristic described previously. With increasing T, there was a small improvement in computation

time, although the speedup was never greater than 2x. The two computation times were virtually the same with increasing ϵ. For large state spaces, the Viterbi heuristic produced more than 5x speedup (which made it comparable to CFDP).

The main reason for the lack of improvement (in computation time) is the randomness of the data generation process. The Viterbi heuristic can significantly outperform the cheap heuristic only if the most likely state sequences according to the transition model receive very poor support from the observations. In that case, the cheap heuristic will provide very inaccurate bounds and mislead the search. In randomly generated models however, the two heuristics demonstrate comparable performance.

5.6 Hierarchy induction

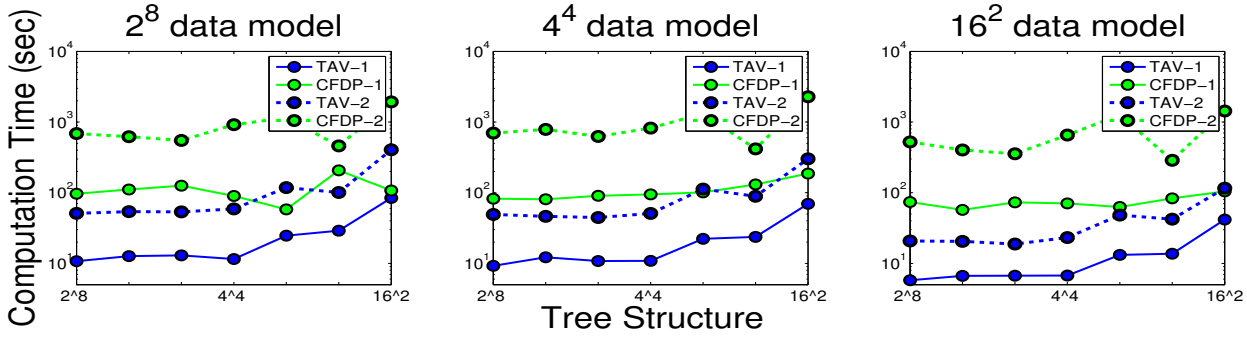


Figure 5.7: Effect of abstraction hierarchy: For different underlying models (2^8, 4^4 and 16^2), deep hierarchies outperform shallow hierarchies. Cases 1 and 2 have $\epsilon = 0.1$ and $.05$ respectively

Till this point, we have assumed that the abstraction hierarchy will be an input to the algorithm. However, in many cases, we might have to construct our own hierarchy. Spectral clustering (Ng *et al.*, 2001) is one technique which we have used in our experiments to successfully induce hierarchies. If the underlying structure is a DBN of binary variables with timescale separation between each variable (as discussed in Section 5.5), there will be a gap in the eigenvalue spectrum after the two largest values. The first two eigenvectors will be analogous to indicator functions for branching on the slowest variable. We can then apply the method recursively within each cluster. The main drawback is the $O(N^3)$ computational cost. It can be argued that this is an offline and one-time cost—nonetheless it is quite expensive. It should be noted that spatial hierarchy induction is also a hard problem.

Let 2^8 denote a 8-variable DBN where each variable is binary. The slowest variables are placed at the left—hence $4^2 2^4$ denotes a DBN whose two slowest variables are 4-valued. As we change the underlying model from 2^8 to 4^4 to 16^2, is there a particular abstraction hierarchy which performs well for all the models?

For the experiments, we generated 3 different data sets using the following DBNs—2^8 (left), 4^4 (middle) and 16^2 (right)—with $N = 256$. On each data set, we used the following abstraction hierarchies (2^8; $2^4 4^2$; $4^2 2^4$; 4^4; $4^1 8^2$; $8^2 4^1$; 16^2). The results in Figure 5.7 show the computation time for TAV and CFDP using different abstraction hierarchies (deepest on the left to shallowest on the right) for two different values of ϵ. Both TAV and CFDP *perform better with deeper hierarchies*, although the improvement is much more pronounced for TAV. The trend across all 3 underlying data models indicates that we could always induce a deep hierarchy. The benefit of lightweight local searches in a deep hierarchy seems to outweigh the cost of the necessary additional refinements.

5.7 Conclusion

We have presented a temporally abstracted Viterbi algorithm, that can reason about sets of trajectories and uses A* search to provably reach the correct solution. Direct links provide a way to reason about trajectories within a set of states—something that previous DP algorithms did not do. For systems with widely varying timescales, TAV can outperform CFDP handsomely. Our experiments confirm the intuition—the greater the timescale separation, the more the computational benefit.

Another smart feature of our algorithm is that it can exploit multiple timescales present in a system by adaptive spatial and temporal refinements. TAV's limitations arise when the system has frequent state transitions and in such cases, it is better to fall back on the conventional Viterbi algorithm (CFDP is often slower as well in such cases). It might be possible to design an algorithm that uses temporal abstraction and can also switch to conventional Viterbi when the heuristic scores of direct links are low.

Chapter 6

Hierarchical image and video segmentation

We also present an image and video segmentation algorithm based on our proposed maximization algorithm.

Existing image and video segmentation algorithms, such as graph cuts, allow every adjoining pixel (superpixel) or voxel (supervoxel) to have a different label. This leads to optimization problems over an exponentially large search space. In practice, real images and videos are both spatially and temporally coherent, and the space of coherent labelings is significantly smaller than the space of all possible labelings. In this chapter, we propose an efficient coarse-to-fine optimization scheme that uses a hierarchical abstraction of the supervoxel graph to distinguish the small set of coherent labelings from the large set of not-so-coherent ones. This abstraction allows us to solve the minimization problem over a coarser graph and to refine the solution only when needed.

The proposed approach is **exact** (i.e., it produces the same solution as minimization over the finest graph), it can be used with many different segmentation algorithms (e.g., graph cuts and belief propagation), and it gives significant speedups in inference for several datasets with varying degrees of spatio-temporal coherence. We also discuss the strengths and weaknesses of our algorithm relative to existing hierarchical approaches, and the kinds of image and video data that provide the best speedups.

6.1 Introduction

Segmenting moving objects in a video sequence is a key step in video interpretation. Most of the prior work on motion segmentation (see, e.g., (Darrel & Pentland, 1991; Shi & Malik, 1998; Cremers & Soatto, 2005; Vidal *et al.*, 2008; Rao *et al.*, 2010)) uses local motion and appearance cues to segment the video in a bottom-up unsupervised manner. However, the use of category-

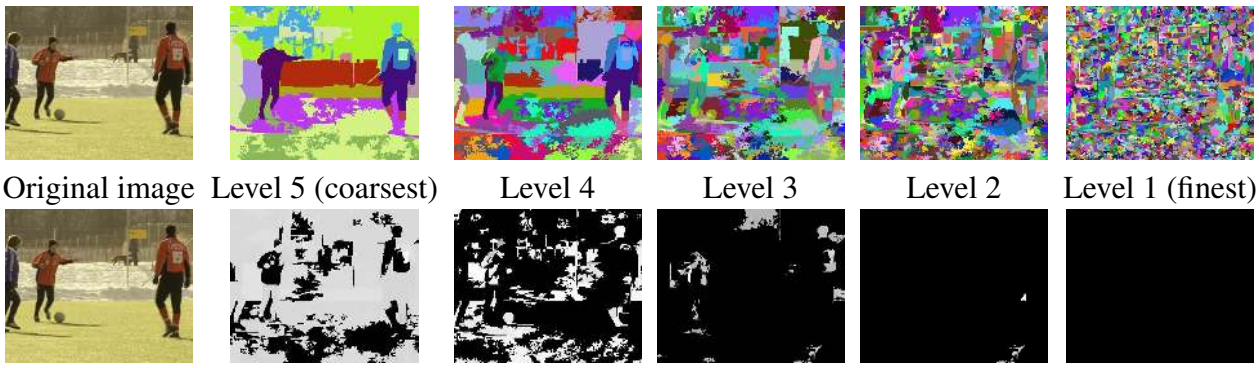


Figure 6.1: Supervoxel hierarchy for an image. The top row shows the various abstraction levels in the supervoxel tree. The second row shows the portion of the supervoxel tree explored to find the optimal labeling of segments.

specific information about the object being segmented can be really helpful in the segmentation task. This has motivated the development of semantic motion segmentation algorithms, which use supervision to label every pixel in video according to the object class it belongs to.

Many of the existing approaches for semantic video segmentation are graph based (Grundmann *et al.*, 2010; Galmar *et al.*, 2008). Generally an over-segmentation of the video is obtained using some standard methods (Floros & Leibe, 2012) and a random field (RF) is defined on a graph whose nodes are the resulting supervoxels. The segmentation of the video is then obtained by minimizing a cost defined on this RF. This minimization is done using different methods (e.g., graph cuts and belief propagation). However, the optimization procedure is typically very slow because of the exponentially large number of possible labelings in a video. For instance, for a video with 100 frames, where each frame has 100×100 superpixels and 10 possible labels, the number of possible segmentations is $10^{1000000}$.

In practice, contiguous supervoxels (both in space and in time) are very likely to have the same label, and *the space of coherent labelings is much smaller than the space of all labelings*. For instance, if we consider supervoxels of size $k \times k$ superpixels spanning k frames, then the number of possible segmentations reduces to $10^{\frac{1000000}{k^3}}$, which is significant even for $k = 2$.

In this chapter, we propose an efficient coarse-to-fine optimization scheme for image/video segmentation that can be used to speedup any graph-based segmentation algorithm, e.g., graph cuts and belief propagation. The key idea behind our approach is to define a hierarchical abstraction of the supervoxel graph such that most supervoxels at the coarser scales belong to a single category. We can solve a much smaller problem over the coarser graph and refine this solution only when needed.

We use a hierarchical graph based supervoxel segmentation method (see (Xu & Corso, 2012) for an overview) to identify the supervoxels (at various scales) that are likely to have the same label. Such methods create a supervoxel tree with the biggest (coarsest) supervoxels at the highest

level. The top row of Figure 6.1 shows the hierarchy for one of the frames from a video of the SUNY-Xiph.org dataset (Chen & Corso, 2010). The second row shows the set of superpixels that our coarse-to-fine inference scheme uses. At each abstraction level, the blacked out portions denote superpixels whose refinements were not required to find the optimal labeling. It is clear that we can prune away a large part of the search space, by assigning several labels at the coarser levels.

Given this hierarchy, we define a coarse-to-fine refinement scheme to find the optimal segmentation. The scheme we use is similar to the approach proposed in coarse-to-fine dynamic programming (Raphael, 2001b) and temporally abstract Viterbi algorithm (Chatterjee & Russell, 2011). We first augment the set of labels with an artificial "mixed" label, which accounts for the fact that coarse supervoxels may contain finer supervoxels with more than one label. We then define the edge costs at any level of the hierarchy as an admissible function of the costs at the next finer level. The coarse-to-fine scheme starts at the coarsest level of supervoxels. If the solution at the current level of refinement is such that some supervoxels have mixed labels, the mixed supervoxels are refined. This process is repeated until the optimal labeling does not assign the "mixed" label to any supervoxel. By properly defining the edge costs at each level, we can guarantee that the optimal segmentation upon termination is identical to the segmentation we would have obtained had we solved the original, non-hierarchical problem, which is exponentially larger in size. The speedup of our approach increases with the spatio-temporal coherence of the data. We attain a speedup of between 2x - 10x on videos from the SUNY-Xiph.org (Chen & Corso, 2010) and CamVid (Brostow *et al.*, 2009) datasets using the hierarchical inference scheme as opposed to the corresponding flat algorithm (graph-cuts and belief propagation).

Related work. There are several existing approaches to hierarchical video (and image) segmentation. One family of work considers a hierarchical cost function which is defined over supervoxels at all levels. This includes the Pylon model (Lempitsky *et al.*, 2011), associative hierarchical CRFs (Ladicky *et al.*, 2009) and a different hierarchical variant of graph cuts (Kumar & Koller, 2009). These approaches find a solution which is *more accurate* than the solution found by solving the problem at the finest level, but they are also *computationally more expensive* (sometimes, only marginally so). Our work finds the solution possessing the same accuracy but speeds up computation time.

Another line of work in hierarchical video segmentation is a bottom-up approach based on merging supervoxels using similarity metrics based on variation of intensity inside a supervoxel (Felzenszwalb & Huttenlocher, 2004; Grundmann *et al.*, 2010). However, there is no explicit cost minimization here, which makes it difficult to compare it with our method. The supervoxel tree obtained by these approaches can be used as the abstraction hierarchy in our algorithmic framework.

Finally, (Felzenszwalb & Huttenlocher, 2006) proposes a version of hierarchical belief propagation for images. However, the abstraction used is image-agnostic and the messages at a coarse level are only used to initialize messages at the finer level and not to prevent expanding all nodes (unlike our work).

The remainder of the chapter is structured as follows. We present the exact problem formulation and some intuition behind the coarse-to-fine (hierarchical) approach in Section 6.2. The hierarchical inference scheme is presented in 6.3 along with the relevant refinement details for a particular cost function. We discuss experimental results in Section 6.4 while also proposing a new on-demand supervoxel tree construction scheme. Finally, in Section 6.5, we conclude by identifying the pros and cons of our framework.

6.2 Problem formulation

Most of the graph-based segmentation algorithms define a random field (RF) whose nodes correspond to pixels (superpixels) or voxels (supervoxels) in the image or video. For the sake of concreteness, we will describe our formulation using a RF whose nodes are the supervoxels in a video. However, the formulation is valid in the other cases as well.

Let $\mathcal{V}$ denote the set of supervoxels in a video V . Each supervoxel $v_i \in \mathcal{V}$ is associated with a state $x_i \in \mathcal{L} = \{1, \ldots, L\}$, which corresponds to the category label at supervoxel v_i. The labeling of a video is denoted by a vector $x \in \mathcal{L}^{\mathcal{V}}$. The edges of the RF are defined using the neighborhood structure of the supervoxels $\mathcal{E} \subset \mathcal{V} \times \mathcal{V}$, where $e_{ij} \in \mathcal{E}$ if supervoxels i and j share a common boundary.

Having defined the structure of the RF, a segmentation cost $E(x, V)$ is associated with every label assignment x, and the minimizer x^* gives the segmentation of the video. Depending on the form of the energy function E, minimization is done using a variety of graph inference methods like α-expansion, belief propagation etc. As discussed before we need to create small supervoxels to learn potentials for a good labeling. In the case of a video with around 100 frames, each frame having a resolution of 960×720, the number of supervoxels we get is of the order of $100,000$.

Hierarchical abstraction

Since the labels are coherent both in space and in time, we expect many large, contiguous patches of supervoxels to have the same label (category). In other words, *the set of coherent labelings, which are very likely, is much smaller than the set of all other labelings, which are very unlikely.*

An effective abstraction scheme that enables us to differentiate between these two sets is a hierarchical supervoxel tree (Felzenszwalb & Huttenlocher, 2004; Grundmann *et al.*, 2010). In the hierarchical setup, a supervoxel at site i and level j is denoted by ${v_i}^j$ and its label is denoted by ${x_i}^j$. The finest level of supervoxels correspond to $j = 1$. The set of all supervoxels at level j is denoted by $\mathcal{V}^j$. The refinement of a supervoxel ${v_i}^j$ ($j \geq 2$) is the set of supervoxels at the next finer level (i.e., $j - 1$) that occupy the same set of voxels in the video as ${v_i}^j$. $Parent : \mathcal{V}^j \rightarrow \mathcal{V}^{j+1}$ is the reverse function mapping a supervoxel to its parent supervoxel at the next coarser level. In this paper, we will only consider hierarchical supervoxel *trees*, and hence each supervoxel has a

unique parent. The aforementioned supervoxel hierarchy can be obtained by running hierarchical segmentation as described in (Grundmann *et al.*, 2010).

Coarse-to-fine inference

Given a hierarchical supervoxel tree, we propose a coarse-to-fine algorithm for efficient inference. The algorithm is designed to distinguish between to scenarios. The likely scenario is when all the (contiguous) supervoxels at level $j-1$ that constitute a supervoxel at level j get the same label (i.e., a label from the set $\mathcal{L}$). The unlikely scenario is when a supervoxel at level j has constituents with different labels. To represent the latter scenario, we introduce a new label $L+1$, which denotes that the corresponding supervoxel ${v_i}^j$ ($j \geq 2$) has constituents with more than one label. We refer to label $L+1$ as the "mixed" label, and to the original L labels as "pure". Of course, only supervoxels which can be further refined, can have the "mixed" label, i.e., ${x_i}^1 \neq L+1$. The augmented label set is denoted by $\mathcal{L}^A = \mathcal{L} \cup \{L+1\}$. A similar label augmentation scheme was used in (Chatterjee & Russell, 2011).

Our coarse-to-fine hierarchical refinement scheme proceeds as follows. We start off with the coarsest supervoxels $\mathcal{V}^m$ and find the optimal label allocation from the augmented label set $\mathcal{L}^A$, using some inference algorithm. This inference algorithm can be graph cuts or belief propagation or some linear program — the choice depends upon the form of the energy function being optimized. Let us denote the selected algorithm by $\mathcal{A}$. All current supervoxels which receive label $L+1$ are replaced in the current problem by their constituent supervoxels from the next finer level (this refinement is always possible, since a supervoxel can only receive the "mixed" label if it can be further refined). After this refinement is done, we again find the optimal label allocation for the current set of supervoxels using $\mathcal{A}$. We repeat this process iteratively, until all supervoxels receive "pure" labels. Since every supervoxel eventually refines to its finest constituents, which in turn can only take "pure" labels, this process is guaranteed to terminate. Also, at any point in the algorithm, there exists exactly one ancestor of every finest level supervoxel ${v_i}^1$ in the current set of supervoxels.

Admissible heuristics and exactness of solution

In order to make our hierarchical graph cuts scheme converge to the same label allocation as running $\mathcal{A}$ on the finest level of supervoxels (for instance, a flat graph cuts algorithm), we have to define the potentials of the abstract supervoxels in a specific way. We use the notion of admissible heuristics as used in the A* algorithm (Russell & Norvig, 2010). Since we are minimizing an energy function, the admissible heuristic for the coarse-level component potentials (unary, pairwise or other higher degree terms) need to be lower bounds of their exact values. The construction of such lower bounds will be explained further in the context of a particular energy function in Section 6.3. Let x denote any label assignment for the finest level supervoxels and let x^* denote the optimal labeling. Let x_{hier} and x^*_{hier} denote the same for the hierarchical setting at any stage of

the algorithm. If we use admissible heuristic costs, then

$$E(x^*, V) \geq E(x^*_{hier}, V) \text{ and } E(x, V) \geq E(x_{hier}, V).$$

What this ensures is that when we terminate upon finding a "pure" label assignment to the current set of supervoxels (at various levels), all other possible assignments have a higher or equal cost (since their lower bound cost is worse than the current optimal cost of the "pure" labeling).

The next section gives the details of how we obtain the lower bounds on the energy potentials for an energy consisting of unary and pairwise terms.

6.3 Hierarhical video segmentation

Let us now formulate a cost function to demonstrate how to use the hierarchical coarse-to-fine strategy to speed up inference.

Cost definition

Consider a cost function which is a linear combination of the sum of unary costs associated with every site $v_i \in \mathcal{V}$ and pairwise costs associated with every edge $e_{ij} \in \mathcal{E}$, i.e.,

$$E(x, V) = \lambda_U \sum_{v_i \in \mathcal{V}} \psi_i^U(x_i, V) + \lambda_P \sum_{e_{ij} \in \mathcal{E}} \psi_{ij}^P(x_i, x_j, V) \tag{6.1}$$

Here, λ_U and λ_P are weights representing the relative importance of the unary and pairwise potentials. These weights are learnt using structural SVMs.

The unary potential, $\psi_i^U(x_i, V)$, represents the cost of assigning the label $x_i \in \mathcal{L}$ to the supervoxel v_i in video V. Unary potentials are usually obtained by training a classifier for every class on appropriate supervoxel descriptors from the videos in the training data.

Supervoxel Size. Almost all the algorithms for creating supervoxels from a video (Grundmann *et al.*, 2010) provide an option of either creating very large and few supervoxels or very fine and numerous supervoxels. Although it is tempting to choose large supervoxels to reduce the graph size, such a decision could be detrimental to the overall objective because of two reasons. First, large supervoxels might contain voxels from more than one category. Since all the voxels which belong to a supervoxel (at the finest level) get the same label, having large supervoxels will lead to inaccurate results. Second, we want the unary potentials to capture the local appearance and motion characteristics of small patches from the object. The pose, scale and illumination variation in the images make it harder to get consistent unary potentials for large supervoxels.

The pairwise potentials $\psi_{ij}^{P}(x_1, x_2, V)$ for an edge $e_{ij} \in \mathcal{E}$ in video V captures the cost of interaction for v_i and v_j for label assignments x_i and x_j. The pairwise potentials are usually designed to enforce spatial smoothness and temporal continuity of the labels.

We will provide more details of the specific form of the unary and pairwise potentials in Section 6.4. For notational convenience, we might drop the video index V in later notation and use $\psi(\cdot)$ and $\psi(\cdot, V)$ interchangeably.

Hierarchical Inference

Let us now introduce some more notation which will be helpful in defining the unary and pairwise potentials of the coarser supervoxels and in formally specifying the general hierarchical inference algorithm. The RF $R_{\mathcal{V}^{curr}}$ is defined by a set of nodes $\mathcal{V}^{curr}$ and the cost defined on this RF is denoted by $E_{\mathcal{V}^{curr}}$. These nodes can correspond to supervoxels at different levels of refinement. As before, a node associated with a supervoxel i at level j is denoted by ${v_i}^j$ and its label by ${x_i}^j$. For every pair of neighboring supervoxels ${v_{i_1}}^{j_1}$ and ${v_{i_2}}^{j_2}$ — neighboring supervoxels can be at different levels of refinement — $e_{(i_1,j_1),(i_2,j_2)} \in \mathcal{E}$ denotes the edge the connecting them. $C(i, j, k)$, where $j \geq k$, gives the list of sites at level k which we can get by refining supervoxel ${v_i}^j \in \mathcal{V}^j$ (this is actually a refinement of $j - k$ levels). The pseudocode of the hierarchical inference algorithm is provided in Algorithm 6.1.

Algorithm 6.1 — Hierarchical Inference Algorithm($\mathcal{V}^{1:m}, \psi$)

$\mathcal{V}^{curr} \leftarrow \mathcal{V}^m$
repeat
  Find $x_{\mathcal{V}^{curr}}$ which minimizes $E_{\mathcal{V}^{curr}}$
  for all ${v_i}^j \in \mathcal{V}^{curr}$ such that ${x_i}^j = L + 1$ **do**
    Refine ${v_i}^j$
    $\mathcal{V}^{curr} \leftarrow \mathcal{V}^{curr} \cup C(i, j, j-1) \setminus {v_i}^j$
  end for
until $L + 1 \notin x_{\mathcal{V}^{curr}}$

As we discussed in Section 6.2, in order for the hierarchical inference algorithm to converge to the exact solution, the unary and pairwise potentials associated with the nodes at the coarse levels should be lower-bounds on the cost associated with the patches of fine nodes constituting these coarse nodes. In what follows, we show how those lower bounds can be computed four our specific energy.

Coarse Unary Potentials. The unary cost for a node ${v_i}^j$ taking one of the pure labels l is the sum of the unary costs of assigning label l to all the nodes at level 1 that constitute ${v_i}^j$, i.e.,

$$\psi_{(i,j)}^{U}({x_i}^j) = \sum_{k \in C(i,j,1)} \psi_{(k,1)}^{U}({x_i}^j), {x_i}^j \in \mathcal{L} \tag{6.2}$$

Notice that this is an exact cost.

The cost of assigning a "mixed" label to a coarse supervoxel is the minimum cost associated with the RF defined by the constituent supervoxels at level 1 *subject to the constraint that all the constituent supervoxels cannot get the same label*. This minimum can be obtained by using α-expansion on $R_{C(i,j,1)}$ if we do not have the constraint that the the nodes in this subgraph can not take the same label. This results in a weaker lower bound. To find the minimum cost with this constraint, we can formulate it as an integer programming problem (Boros & Hammer, 2002) with an extra constraint which prevents all the nodes from taking the same label.

Pairwise Potentials. We define the pairwise potential for nodes $v_{i_1}^{j_1}$ and $v_{i_2}^{j_2}$, $\psi^P_{(i_1,j_1)(i_2,j_2)}(x_{i_1}{}^{j_1}, x_{i_2}{}^{j_2})$, as follows

$$\begin{cases} 0 & \text{if } x_{i_1}{}^{j_1} = x_{i_2}{}^{j_2} \\ \sum_{\hat{\mathcal{E}}} \psi^P_{(i,1)(j,1)}(x_{i_1}{}^{j_1}, x_{i_2}{}^{j_2}), & \text{otherwise} \end{cases}$$

Here, $\hat{\mathcal{E}} \subseteq \mathcal{E}$ and is defined by $\hat{\mathcal{E}} = \{e_{(i,1)(j,1)} \in \mathcal{E} : i \in C(i_1, j_1, 1), j \in C(i_2, j_2, 1)\}$. The cost of the edge between two big supervoxels (i_1, j_1),(i_2, j_2) is the sum of the costs of the edges connecting the constituent supervoxels of (i_1, j_1) and (i_2, j_2). In case one of the supervoxels gets the mixed label the potential associated to the edge is set to zero. Although this is a loose lowerbound to the actual cost of these edges (while minimizing the cost on $R_{\mathcal{V}_1}$) this saves us a lot of computation time.

Optimization algorithm

In Algorithm 6.1, the optimization in Step 3 can be performed using different methods. The most popular choice is graph cuts (Boykov *et al.*, 2001) via α-expansion and α-β swap moves. Another algorithm frequently used is max belief propagation (Felzenszwalb & Huttenlocher, 2004). Other alternatives include various linear and non-linear optimization algorithms. The choice is most frequently guided by the particular form of the energy function we are trying to minimize.

Our hierarchical algorithm is exact only with respect to the underlying optimization algorithm. For instance, α-expansion is guaranteed to find a solution within a factor of 2 for metric pairwise potentials (Boykov *et al.*, 2001). Since the hierarchical scheme will find the exact same solution that α-expansion would find when run on the original finest level problem, it will share its approximation quality guarantees. For energy functions with higher order terms, we can still use the hierarchical scheme with an appropriate optimization algorithm in every iteration.

Practical considerations

Trade-off between accuracy (tighter bounds) and computation time

Consider two scenarios: all the lower bounds on potentials in scenario 1 are tighter than the corresponding bounds in scenario 2, for the same supervoxel hierarchy. In that case, the number of

refinements required in scenario 1 will be strictly non-greater than the number of iterations required in scenario 2. Hence, *it is always beneficial to consider tighter lower bounds*. This is true for any algorithm using admissible heuristics. However, the downside of using tighter bounds is that they generally are more expensive to compute. Thus, a trade-off exists between accuracy of heuristic costs and time required to compute them.

On demand supervoxel refinement

In most cases, only a small number of nodes in the supervoxel tree are used in the entire inference procedure. Thus, we can save computation time by not computing the entire supervoxel tree upfront, and only refining the desired supervoxels (the ones with the "mixed" label) when needed. This on-demand refinement scheme, however, can be more expensive if we end up expanding most of the nodes in the supervoxel tree. We see moderate benefits using this scheme as reported in Section 6.4.

Extension to label hierarchy

In this work, we have only considered a flat label hierarchy. However, it is possible to consider a hierarchy among labels as well. For instance, since the sky and the sea labels are (often) similar, we might get additional computational benefits by considering them together (and therefore eliminating them via a single consideration for non-sky, non-sea nodes). Such a hierarchical scheme would have a "mixed" label at every label level. For more details on how to manage a label hierarchy simultaneously with a supervoxel hierarchy, refer to (Chatterjee & Russell, 2011), where such a scenario is considered.

6.4 Experiments

Although our proposed hierarchical scheme can be exponentially faster than flat optimization in the best case, it could also be much slower when we need to refine all the supervoxels down to their finest level. Hence, it is important to validate the usefulness of this coarse-to-fine approach to see whether it actually provides a speedup and how large this speedup is. It is also expected that the speedup will be much larger for videos with greater spatio-temporal coherence.

Dataset

For our experiments, we used two datasets — the SUNY Buffalo-Xiph.org 24-class Dataset (Chen & Corso, 2010) and the CamVid dataset (Brostow *et al.*, 2009).

The SUNY Buffalo-Xiph.org 24-class Dataset is a collection of general-purpose videos collected at `Xiph.org`. The frame-by-frame labels in the video have a fair bit of temporal consistency (which our algorithm is well-equipped to exploit). For each video, we use half of the frames for training and the remaining half for testing. We retained all 24 labels for this dataset, although most videos only had $5 - 10$ labels.

The CamVid Dataset has 700 hand segmented frames of street scenes with varying backgrounds captured from a video camera in a moving car. The video sequence has been annotated into 32 classes. However in this chapter we have combined some of the classes and are working with a total of 5 classes - people, vehicles, sky, road and background. Around 250 frames were used for training and rest of them were used for testing. This represents a more challenging dataset (in terms of an expected speedup).

Learning potentials

The unary potentials We obtained the unary potentials by learning an SVM on the supervoxel descriptors for every class. The score associated with a supervoxel descriptor d_i of the supervoxel v_i for a class x defines the unary potential $\psi_i^U(x, V)$.

Supervoxel Descriptors: The supervoxel descriptor needs to be chosen such that it captures the discriminative characteristics of the supervoxels (both appearance and motion attributes) across various classes. In our experiments, we found the Spatio-Temporal Interest Points (STIP) (Laptev & Lindeberg, 2003; Laptev, 2005) descriptors to be a good fit.

The pairwise potentials We define the pairwise potential between the sites i and j as:

$$\psi_{ij}^P(x_i, x_j) = \frac{l_{ij}}{1 + |I_i - I_j|}\delta(x_i \neq x_j)$$

where l_{ij} is the area of the common boundary between the supervoxels v_i and v_j and I_i denotes the average intensity of supervoxel v_i.

Experimental setup

For our experiments, we use α-expansion (graph cuts (Boykov *et al.*, 2001)) and belief propagation (Felzenszwalb & Huttenlocher, 2004) as the optimization algorithm in Step 3 of Algorithm 6.1. We implemented the algorithms in Python and used various functionalities from the graph libraries **Python-graph** (`https://code.google.com/p/python-graph/`) and **igraph** (Csardi & Nepusz, 2006). We used the LIBSVX (`http://www.cse.buffalo.edu/~jcorso/r/supervoxels/`) library's implementation of a graph-based hierarchical method (Grundmann *et al.*, 2010) to generate the supervoxel trees.

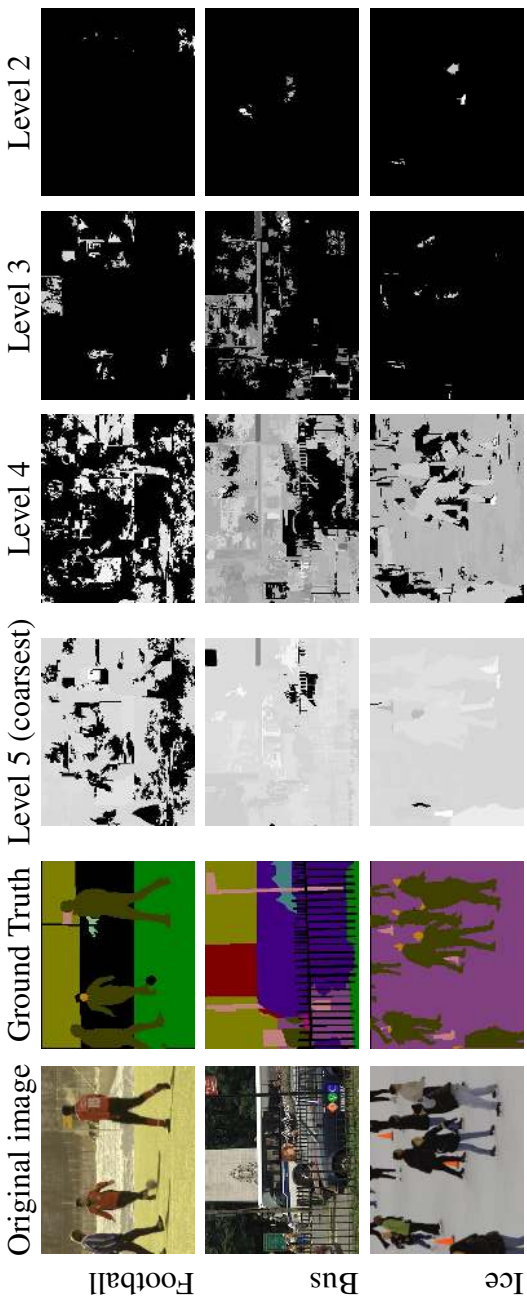


Figure 6.2: Explored portions of the supervoxel tree. The blacked out portions in each superpixel level denotes the patch of superpixels which were never refined during inference. The top row shows results from the "football" video, the middle row from the "bus" video and the bottom row from the "ice" video (all from the SUNY dataset).

The parameters were adjusted such that the CamVid videos, which had much larger number of voxels, only had about 2x the number of supervoxels at the finest level as the SUNY-Xiph.org videos.

Results

Computational speedup: The computation time taken by the flat algorithm (both α-expansion and belief propagation) and its hierarchical counterpart are reported in Table 6.1. For the CamVid videos, the speedup is between 3x - 5x. The speedup obtained for the SUNY dataset videos is between 7x -10x. The increased speedup for the SUNY videos is expected due to the increased spatio-temporal consistency in those videos. These speedups do not account for the supervoxel tree creation time, which is only required by our algorithm. If we include the time of the on-demand refinement scheme discussed in Section 6.3, the overall speedup reduces to 2x - 4x for CamVid and 5x - 6x for the SUNY videos.

Reduced problem size: Besides the computation time, it is also informative to look at the explored portions of the supervoxel tree to get a better understanding of where the computational savings come from. In Figure 6.2, we show the explored portions of the state space for one frame each of three SUNY videos. The leftmost image is the original image, followed by the ground truth label for the image. The next four images from left to right are superpixels (since we are only looking at one frame) at different levels of abstraction (coarsest on the left). The blacked out superpixels at each level are the ones which *never* received the "mixed" label and hence were never refined. Eliminating such entire superpixel (supervoxel) trees rooted at these blackened superpixels (supervoxels) is the key behind obtaining computational speedup.

We are not showing any segmentation results in the chapter since the quality of the segmentation is the same as what would be obtained by using α-expansion or belief propagation with the chosen energy function. We will include the segmentation videos (both final and intermediate iterations) in the supplementary material.

Accuracy vs time

Another interesting metric to track is the percentage of correctly classified voxels. When the hierarchical algorithm terminates, this percentage reaches the same value that would be obtained by running inference on the flat problem formulation. As shown in Figure 6.3, this final accuracy lies between 55% and 75%. However, there seems to be no clear trend in how this accuracy is achieved as a function of iterations. For the "bus" video, the accuracy quickly spikes up and then reaches a plateau, while for "ice", it spikes up after a few iterations. A surrogate for this accuracy (the percentage accuracy is often unavailable since there is no ground truth) is the cost function. We can use the cost function to design an anytime version of the algorithm, where termination could be guided by sharp spikes (or the lack thereof) in the cost function.

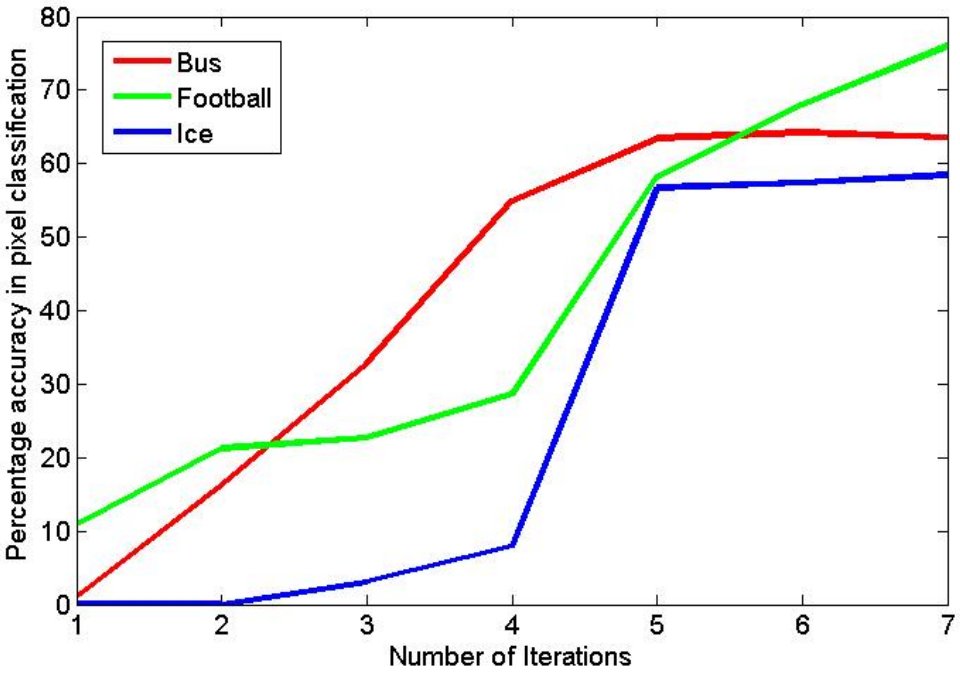


Figure 6.3: Percentage of correctly classified supervoxels after every iteration of the hierarchical belief propagation algorithm.

6.5 Conclusion

In this chapter, we have presented a general coarse-to-fine or hierarchical scheme for video segmentation. The key intuition behind the proposed solution is the fact that the set of likely label assignments is exponentially smaller than the set of all possible label assignments. A flat problem formulation works with the latter large set, while we use an abstraction scheme (namely supervoxel trees) to identify the former smaller set and work on the smaller problem. The framework is general since it can use any optimization algorithm to find the optimal label for the intermediate problems. It is also exact since it uses admissible heuristic costs for the coarser supervoxel potentials. We show results using α-expansion and belief propagation on two different video datasets and obtain speedups ranging from 2x - 10x. As expected, the speedup obtained is larger for videos with more spatio-temporal continuity.

As with any general framework, there remains a fair bit of exploration to do. Other abstraction schemes and optimization algorithms could yield better results (for other specific data sets). There is also the accuracy to computation time trade-off in the heuristics computation. Another direction would be to compromise on the exact nature of the solution and design really fast (and perhaps, slightly more inaccurate) segmentation algorithms.

Algorithm		**CamVid**					**SUNY**		
		CamVid1	CamVid2	CamVid3	CamVid4	CamVid5	Bus	Football	Ice
α-expansion	Flat	130.1	137.3	117.6	145.1	140.1	35.3	25.0	32.7
	Hierarchical	32.7	40.9	27.3	43.8	29.4	6.5	2.3	5.3
Belief Propagation	Flat	256.0	270.1	258.3	307.0	319.2	50.3	34.7	50.9
	Hierarchical	50.5	79.1	61.5	107.7	90.5	9.3	4.1	8.3

Table 6.1: Time taken by the different inference algorithms on different data sets (in minutes). The times reported for the hierarchical case *does not* include supervoxel tree computation time.

Chapter 7

MCMC and near-determinism

Markov chain Monte Carlo (MCMC) algorithms are notorious for getting stuck in the presence of near-deterministic relationships in graphical models. In this chapter, we propose a methodology to design MCMC algorithms for a subclass of near-deterministic probabilistic models.

In the previous chapters, we have focused on exploiting near-determinism in probabilistic models for exact inference algorithms. We now turn our focus to approximate inference algorithms, namely Markov chain Monte Carlo (MCMC), and show how we can make these algorithms more efficient in near-deterministic systems.

MCMC algorithms are used ubiquitously to generate samples from a probability distribution where exact sampling is not feasible. However, in the presence of near-deterministic components in a joint distribution, it is well-known that mixing is very slow and the chain often gets stuck in local modes.

In this chapter, we explore one potential fix to this problem by visiting the deterministic problem corresponding to the near-deterministic components. If there exists an efficient algorithm (say A_{det}) to sample from the solutions to this deterministic problem, then we show that it is often possible to design an MCMC algorithm A_{MCMC} which provides the computational benefits in the near-deterministic domain that A_{det} provides in the deterministic scenario.

We present a general method to design A_{MCMC} from A_{det} and enumerate specific MCMC algorithms for two different constraint satisfaction problems. These new algorithms show promising results in the near deterministic domain as compared to a Gibbs sampler.

7.1 Introduction

Markov chain Monte Carlo (MCMC) methods, introduced in Section 2.3, are a general class of algorithms which have been used widely in the fields of computational biology, econometrics, physics, statistics and many others over the past three decades. MCMC methods generate samples from probability distributions based on the idea of constructing a Markov chain whose invariant distribution corresponds to the probability distribution (often referred to as the target distribution) of interest. They have been used in numerical integration, optimization and simulation of systems in the various fields mentioned above. Specific instances of MCMC algorithms like the Metropolis algorithm (Metropolis *et al.*, 1953), the Metropolis-Hastings algorithm (Hastings, 1970) and Gibbs sampling (Geman & Geman, 1984) have had great influence on computational aspects of science and engineering in general. For a more detailed introduction to MCMC methods, and their various theoretical and practical nuances, please refer to (Robert & Casella, 2005). Applications in machine learning have been well highlighted in (Andrieu *et al.*, 2003).

In the presence of near-determinism, MCMC methods often run into trouble. The process of exploring the state space by an MCMC algorithm is inherently tied to the stochastic nature of the problem. As the amount of stochasticity in a problem decreases, the solution modes become more and more "isolated" in the probability density landscape. This makes navigation from one mode to another increasingly difficult. As a result, the mixing time of the MCMC algorithm increases.

Certain approaches have previously been proposed. One potential solution is blocked Gibbs sampling (Geman & Geman, 1984), (Robert & Casella, 2005), where any tightly coupled set of variables (i.e., a set of variables having a near-deterministic relationship) are sampled as a block. While this might solve the original problem, it introduces two new additional problems – identifying such tightly coupled blocks and sampling for such potentially large blocks. In the worst case, we might have to sample the entire system as a block (which was not feasible in the first place). The blocked Gibbs sampling can also be used as a global kernel which jumps between modes, and is then mixed with a local kernel which explores the space around each identified mode. Such mixtures of kernels also form an important class of MCMC algorithms (Tierney, 1994).

An interesting alternative is MC-SAT (Poon & Domingos, 2006), which is an MCMC algorithm to generate samples for Markov logic networks (MLNs) (Richardson & Domingos, 2006). MC-SAT can handle near-deterministic and deterministic clauses in MLNs. It proceeds by first identifying the set of clauses which are satisfied by the current state. It then samples from this set of clauses based on their level of determinism (more the determinism, more the likelihood of the clause getting sampled). Finally, it generates the new state by sampling from the solution space of the set of sampled clauses from the previous step. The final sampling is done using SampleSAT (Wei *et al.*, 2004) which can sample (approximately) uniformly from the solution set of any deterministic SAT problem.

In this chapter, we explore a different approach to solving this problem. Let P_{det} be a deterministic problem (e.g., $P_{det} = k - SAT$) and we are interested in sampling from $P_{near-det}$ which is a near-deterministic version of P_{det}. Let A_{det} be an algorithm which finds solutions to P_{det}. Under

this setting, is it possible to design an MCMC algorithm, say A_{MCMC} which converges[1] to A_{det} as $P_{near-det}$ approaches the limit of determinism (i.e., P_{det})? We show that the answer to this question is in the affirmative, although A_{det} has to satisfy certain conditions.

While a conventional MCMC algorithm's performance worsens as the problem approaches determinism, A_{MCMC} (which converges in behavior to A_{det}) is relatively unaffected. There are two key intuitions which drive the design of A_{MCMC} based on A_{det}. Firstly, A_{det} is used to design a proposal distribution for the next state, given the current state. Secondly, delayed rejection MCMC (Mira, 2001), (Green & Mira, 2001) is used to retain the suboptimal samples that the proposal distribution (analogous to A_{det}) generates, since they are often vital to get out of a local mode. This differs from the conventional use of delayed rejection MH to shape proposals to stay away from rejected samples. In the paper, we prove a theorem which states, that under certain assumptions, it is always possible to design A_{MCMC} for a problem $P_{near-det}$ given an A_{det} for the deterministic problem P_{det}. Another important point to note is that A_{MCMC} will share both the advantages and disadvantages of A_{det} as $P_{near-det}$ approaches determinism.

Using this principle, we design MCMC algorithms for the stochastic SAT problem and for a class of sum constraint problems. We present guidelines on how to design A_{MCMC}, given A_{det}. Following these guidelines, we also show that Gibbs sampling for certain stochastic constraint satisfaction problems (CSPs) converges to the Min-Conflicts heuristic algorithm in the limit of determinism (i.e., A_{MCMC} = Gibbs for A_{det} = Min-Conflicts). Our experiments show significant improvement in the quality of samples generated by these new algorithms over Gibbs sampling. Such a design of A_{MCMC} from A_{det} will allow the user to incorporate well-research insights from A_{det} into an MCMC algorithm for the near-deterministic realm with relative ease — this can be considered to be the main contribution of the paper.

The rest of the chapter is structured as follows. Section 7.2 presents some notations and algorithms which we use as building blocks in the following sections. We then describe the general framework and guidelines on designing A_{MCMC} based on A_{det} in Section 7.3. Section 7.4 and Section 7.5 outline the stochastic SAT problem and the sum constraint problem respectively and present corresponding MCMC algorithms. Empirical evaluations are reported and discussed in Section 7.6 before we conclude in Section 7.7.

7.2 Preliminaries

Notation

We will be representing random variables with capital letters (e.g., X, Y) while sets of random variables will be denoted by bold capital letters $\mathbf{X}, \mathbf{Y}$. Their instantiations will be symbolized by

[1]The convergence is in terms of the set of samples that A_{det} and A_{MCMC} generate. A detailed definition of convergence is provided in Section 7.3 and Theorem 7.3.1.

small letters – x and $\mathbf{x}$ respectively. For brevity, we will often use $p(.|x, y)$ to represent $p(.|X = x, Y = y)$. For a brief introduction to MCMC algorithms and some of the notation we are using in this chapter, please refer to Section 2.3. Let us now briefly review some specific MCMC algorithms that we will use later in this chapter.

Delayed Rejection MH

The Metropolis-Hastings (MH) algorithm (Hastings, 1970) is an MCMC algorithm that was introduced earlier is Section 2.3. We now introduce delayed rejection MH algorithm (Mira, 2001)(Green & Mira, 2001), which is a specific example of an adaptive MCMC (Andrieu & Thoms, 2008)(Tierney & Mira, 1999) algorithm. Adaptive MCMC is a family of MCMC algorithms where the proposal distribution $q(.|.)$ adapts as more samples are generated. In delayed rejection MH, when a new proposed state $\mathbf{x}'$ is rejected, the proposal distribution adapts and takes the form $q(.|\mathbf{x}, \mathbf{x}')$. For instance, this can be used to avoid proposing states similar to $\mathbf{x}'$ in case similar states have similar probability values. This process can be continued until the current proposed state is accepted. After a state (say $\mathbf{x}^*$) is accepted, the proposal distribution reverts back to $q(.|\mathbf{x}^*)$. It should be noted that $\mathbf{x}$ is not replicated as the sample while the intermediate proposed states are rejected. The next sample is $\mathbf{x}^*$.

The acceptance probability of an intermediate step (starting from $\mathbf{x}_0$ and having rejected $\mathbf{x}_1$, $\cdots$, $\mathbf{x}_{k-1}$) is given by:

$$\alpha_1(\mathbf{x}_0, \mathbf{x}_1) = \min\left(1, \frac{\pi(\mathbf{x}_1)q_1(\mathbf{x}_0|\mathbf{x}_1)}{\pi(\mathbf{x}_0)q_1(\mathbf{x}_1|\mathbf{x}_0)}\right) \tag{7.1}$$

$$\alpha_2(\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2) = \min\left(1, \frac{\pi(\mathbf{x}_2)q_1(\mathbf{x}_1|\mathbf{x}_2)q_2(\mathbf{x}_0|\mathbf{x}_1, \mathbf{x}_2)}{\pi(\mathbf{x}_0)q_1(\mathbf{x}_1|\mathbf{x}_0)q_2(\mathbf{x}_2|\mathbf{x}_1, \mathbf{x}_0)} \frac{[1-\alpha_1(\mathbf{x}_2, \mathbf{x}_1)]}{[1-\alpha_1(\mathbf{x}_0, \mathbf{x}_1)]}\right) \tag{7.2}$$

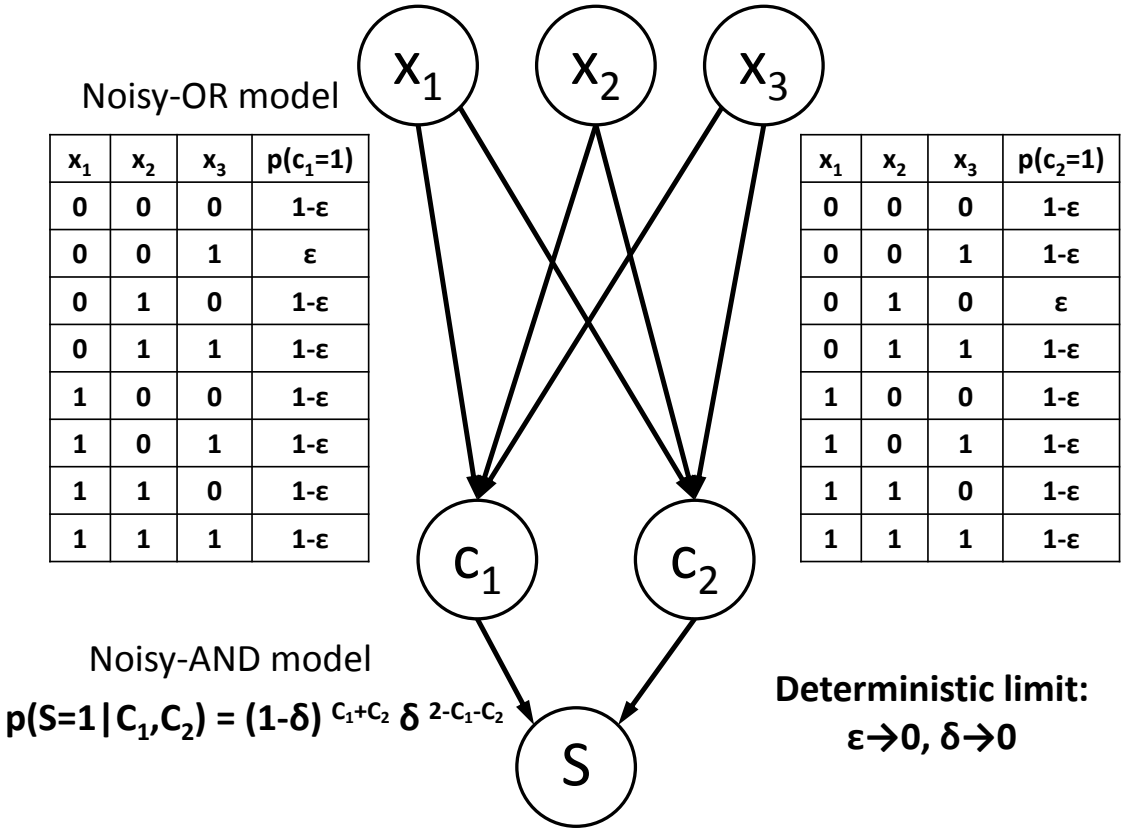
$$\vdots$$

$$\alpha_k(\mathbf{x}_0, \cdots, \mathbf{x}_k) = \min\left(1, \frac{\pi(\mathbf{x}_k)}{\pi(\mathbf{x}_0)} \frac{q_1(\mathbf{x}_{k-1}|\mathbf{x}_k)\cdots q_k(\mathbf{x}_0|\mathbf{x}_1, \cdots, \mathbf{x}_k)}{q_1(\mathbf{x}_1|\mathbf{x}_0)\cdots q_k(\mathbf{x}_k|\mathbf{x}_{k-1}, \cdots, \mathbf{x}_0)} \frac{[1-\alpha_1(\mathbf{x}_k, \mathbf{x}_{k-1})]\cdots[1-\alpha_{k-1}(\mathbf{x}_k, \cdots, \mathbf{x}_1)]}{[1-\alpha_1(\mathbf{x}_0, \mathbf{x}_1)]\cdots[1-\alpha_{k-1}(\mathbf{x}_0, \cdots, \mathbf{x}_{k-1})]}\right)$$

where $\alpha_i(\mathbf{x}_0, \mathbf{x}_1, \cdots, \mathbf{x}_i)$ $(i < k)$ denotes the acceptance probability of $\mathbf{x}_i$ having started from $\mathbf{x}_0$ and previously rejecting $\mathbf{x}_1, \cdots, \mathbf{x}_{i-1}$ before proposing $\mathbf{x}_i$. This is computed by enforcing detailed balance at every intermediate step. For more details, please refer to (Mira, 2001).

Example of a near-deterministic problem

An example of a near-deterministic problem is a stochastic version of the well-studied $3-SAT$ problem. We will now explain what "near-determinism" means and how we can characterize the amount of stochasticity. Let there be N literals $\mathbf{X} = \{X_1, X_2, \cdots, X_N\}$ and M disjunctive clauses $\mathbf{C} = \{C_1, C_2, \cdots, C_M\}$ where $C_i = X_{i,1} \vee X_{i,2} \vee X_{i,3}$. Here, $X_{i,j} \in \mathbf{X}$ and could also be the negation of a literal. Finally, there is a conjunctive clause denoted by a variable S, where $S = C_1 \wedge C_2 \wedge \cdots \wedge C_M$. P_{det} is the problem of generating $\mathbf{x}$ such that $s = 1$.



x_1	x_2	x_3	$p(c_1=1)$
0	0	0	1-ε
0	0	1	ε
0	1	0	1-ε
0	1	1	1-ε
1	0	0	1-ε
1	0	1	1-ε
1	1	0	1-ε
1	1	1	1-ε

x_1	x_2	x_3	$p(c_2=1)$
0	0	0	1-ε
0	0	1	1-ε
0	1	0	ε
0	1	1	1-ε
1	0	0	1-ε
1	0	1	1-ε
1	1	0	1-ε
1	1	1	1-ε

Figure 7.1: A stochastic CSP in conjunctive normal form, where the clauses are disjunctions. The CPTs (corresponding to the example in the text) show the near-deterministic nature of the disjunctions and conjunction.

Now, we can introduce stochasticity in two ways – by having a prior distribution on $\mathbf{X}$ or by adding noise to the disjunctive and conjunctive clauses. Let us focus on the latter for the moment. In the example shown in Figure 7.1, $N = 3, M = 2, C_1 = X_1 \vee X_2 \vee \neg X_3$ and $C_2 = X_1 \vee \neg X_2 \vee X_3$. Let us now introduce some noise in the disjunction and conjunction conditional probability tables (CPTs) – the order of the noise is denoted by ϵ and δ respectively. The actual amount of noise for each CPT entry can be obtained by multiplying ϵ (or δ) with a random number $r \in (0, 1)$.

If we wish to have different values of ϵ for the different disjunctions, we can represent it with $\vec{\epsilon}$. As $\epsilon, \delta \rightarrow 0$, the model approaches determinism. An example problem $P_{near-det}$ in this near-deterministic graphical model could be to generate samples from $p(\mathbf{x}|S = 1)$.

7.3 General Framework and algorithm

Designing A_{MCMC}

In this section, we outline the general procedure of designing the MCMC algorithm A_{MCMC} given the algorithm A_{det} for the deterministic problem P_{det}. Firstly, A_{det} should satisfy the following two properties:

1. A_{det} should generate solutions to P_{det} using some local search heuristics. A_{det} can also generate intermediate non-solution states.
2. The next state proposed by A_{det} can depend on all previous states upto and including the last solution state. Of course, it is also okay if the proposal does not depend on the current state at all.

An example of A_{det} can be the WalkSAT algorithm (Selman *et al.*, 1994)(McAllester *et al.*, 1997) for generating solutions to SAT problems. The algorithm is outlined in Algorithm 7.3. In every iteration, WalkSAT picks a clause which is currently unsatisfied, and then flips a variable within that clause. The unsatisfied clause is generally selected at random. Then with probability α (a parameter of the algorithm) the variable to flip is chosen randomly from the selected clause, and with probability $1 - \alpha$, the variable to flip is chosen greedily so as to minimize the number of unsatisfied clauses. It should be noted that WalkSAT (like many other search algorithms in the deterministic space) makes suboptimal moves. In fact, these suboptimal moves often help the algorithm escape a local non-solution maxima and find solutions. In order for A_{MCMC} to perform well as $P_{near-det}$ approaches determinism, we propose two general design choices:

1. The proposal distribution $q(.|x)$ should be structured so that it converges, in the limit of determinism, to the distribution of the next state that A_{det} would generate after x.
2. The suboptimal moves that A_{det} proposes should not be immediately discarded. These suboptimal states often help in moving between modes or escaping a local maxima. However, the probability of accepting a suboptimal state tends to 0 as $P_{near-det}$ approaches determinism. We work around this by employing delayed rejection Metropolis Hastings (Mira, 2001). As described in Section 7.2, the subsequent proposals can be based on all previous states upto the last accepted sample, i.e., $q_k(.|x_0, x_1, \cdots, x_{k-1})$. Depending on A_{det}, we

choose an appropriate $q(.|.)$. For instance, when A_{det} is WalkSAT, then $q_k = q(x_k|x_{k-1})$. Note that x_0 is the last accepted sample in A_{MCMC}.

Based on these two principles, we can design A_{MCMC} for $P_{near-det}$ when A_{det} satisfies the aforementioned properties.

Properties of proposal distributions

There is some flexibility in the choice of the proposal distribution. For example, if we wish to choose an unsatisfied clause in the SAT problem in the deterministic limit, then we could use either $\cap p(C_i = 0)$ or $\cap p(C_i = 0)^2$ to guide our selection ($\cap p(.)$ denotes an unnormalized distribution). One way to potentially quantify the quality of the proposal distribution is to use a divergence measure between the next state distribution of A_{det} and $q(.|\mathbf{x})$ when the current state is $\mathbf{x}$. This divergence should monotonically decrease with ϵ and $\epsilon \to 0$.

Distribution of A_{MCMC} samples

Let us now prove an important property about the set of samples generated by an A_{MCMC} designed according to the guidelines mentioned above.

Theorem 7.3.1 *Let S_{det} represent the set of unique solution states that are visited infinitely often by A_{det} and let S_{MCMC} be the set of unique states that are sampled by A_{MCMC}. Then, in the deterministic limit, $S_{MCMC} = S_{det}$. Additionally, the frequency with which A_{MCMC} generates states in S_{det} will follow the target distribution,* ***even if*** *A_{det} does not satisfy that property.*

Proof To prove the above statement, let us refer back to the acceptance probability for delayed rejection MH, as given in Equation 7.2 in Section 7.2. Also, we can ignore the initial phase of A_{MCMC} before it has found a solution state. We now focus on the case when $\mathbf{x}_0$ is a solution state. If $\mathbf{x}_1$ is not a solution state, then $\pi(\mathbf{x}_1) \to 0$ and therefore $\alpha_1(\mathbf{x}_0, \mathbf{x}_1) \to 0$.

Now let $\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_{i-1}$ all be non-solution states. In Equation 7.2, $\pi(\mathbf{x}_i)/\pi(\mathbf{x}_0) \to 0$. Additionally, the alpha terms in the numerator like $\alpha_1(\mathbf{x}_i, \mathbf{x}_{i-1})$ denote acceptance probability starting from a non-solution state and ending in a non-solution state. Hence, these terms could be non-zero. The denominator α terms however all approach 0. Hence, the ratio of the $[1 - \alpha_.(.)]$ terms is also smaller than 1. Also, all the $q_.(.)$ terms in the denominator are bounded away from 0 since those are actual states which were proposed. Therefore, $\alpha_i(\mathbf{x}_0, \mathbf{x}_1, \cdots, \mathbf{x}_i) \to 0$.

Now, let $\mathbf{x}_n$ be a solution state, and all the states $\mathbf{x}_1, \cdots, \mathbf{x}_{n-1}$ are non-solution states. In the expression for $\alpha_n(\mathbf{x}_0, \cdots, \mathbf{x}_n)$, all the $\alpha_\cdot(.)$ terms $\rightarrow 0$, since they all start at a solution state (either $\mathbf{x}_0$ or $\mathbf{x}_n$) and end in a non-solution state. Hence, the acceptance probability in the limit of determinism can be calculated as follows:

$$\begin{aligned}\alpha_n(\mathbf{x}_0, \cdots, \mathbf{x}_n) \rightarrow \min \Bigg(& \frac{\pi(\mathbf{x}_n)}{\pi(\mathbf{x}_0)} \\ & \frac{q(\mathbf{x}_{n-1}|\mathbf{x}_n) \cdots q(x_1|x_0)}{q(\mathbf{x}_1|\mathbf{x}_0) \cdots q(\mathbf{x}_n|\mathbf{x}_{n-1})} \Bigg) \\ = \min \Bigg(& \frac{\pi(\mathbf{x}_n) q^*(\mathbf{x}_0|\mathbf{x}_n)}{\pi(\mathbf{x}_0) q^*(\mathbf{x}_n|\mathbf{x}_0)} \Bigg)\end{aligned} \tag{7.3}$$

Thus, A_{MCMC}, in the limit of determinism accepts only solution states. Additionally, the proposal distribution is derived directly from A_{det}. Now, A_{det} is a first-order Markovian algorithm, which satisfies the property $\forall \mathbf{x} \in S_{det}, \pi^*(\mathbf{x}) > 0$, where $\pi^*(.)$ is the invariant distribution for A_{det}. This entails that $\forall \mathbf{x}' \pi(\mathbf{x}') > 0 \implies \mathbf{x}' \in S_{MCMC}$, where $\pi(.)$ is the target distribution, which in the deterministic limit is proportional to the prior probability of the solution state.

An alternate way of looking at it is that the support of the composite proposal distribution $q^*(.)$ includes the support of A_{det}. Thus $S_{MCMC} = S_{det}$.

Additionally, since A_{MCMC} is a valid MCMC algorithm, it will sample all states in S_{MCMC} according to their posterior probability. So, even though WalkSAT is known to generate certain samples exponentially more frequently than others, WalkSAT-MCMC (presented in Section 7.4) would not exhibit a similar bias. $\square$

However, if A_{det} has a bias towards certain solutions, then the mixing time of A_{MCMC} will be affected. We will show this through empirical results in Section 7.6.

7.4 Specific problem I: Near-deterministic SAT

We have already introduced the problem in Section 7.2. Now, let us look at some specific algorithms for P_{det} corresponding to generating solutions to a $k - SAT$ problem (for $k \geq 3$). $P_{near-det}$ corresponds to generating samples from $p(\mathbf{x}|S = 1)$. For now, we assume $\delta = 0$ for ease of disposition. We will revisit other noise models later in this section.

Min-Conflicts and Gibbs sampling

The first algorithm we consider is the Min-conflicts algorithm, which is a popular local search strategy for finding solutions to CSPs. At every iteration, a variable is selected which occurs in a

currently unsatisfied clause. The value of this variable is then set so as to minimize the number of unsatisfied clauses. The details are provided in Algorithm 7.1.

Algorithm 7.1 — Min-conflicts(CSP(**X**, **C**, S), iter)

1: $\mathbf{x} \leftarrow$ Initialize from prior of **X**
2: **for** i = 1 to iter **do**
3: Pick a variable $X_k \in \mathbf{X}$ which occurs in some unsatisfied clause C_j
4: $X_k \leftarrow x_k$ which minimizes #Conflicts$(x_k, \mathbf{x}_{\setminus k}, \mathbf{C})$
5: (Break ties randomly)
6: **end for**

Now, let us consider Gibbs sampling for the stochastic CSP ($\epsilon > 0, \delta = 0$). The full details are shown in Algorithm 7.2.

Algorithm 7.2 — GibbsSampling(CSP(**X**, **C**, S), iter)

1: $\mathbf{x} \leftarrow$ Initialize from prior of **X**
2: **for** i = 1 to iter **do**
3: Pick a variable $X_k \in \mathbf{X}$
4: $x_k \sim p(x_k|\mathbf{x}_{\setminus k}, S = 1)$
5: **end for**

The conditional distribution in Line 4 of Algorithm 7.2 can be rewritten as:

$$p(x_k|\mathbf{x}_{\setminus k}, S = 1) \quad = \quad \epsilon^{m(x_k, \mathbf{x}_{\setminus k})}(1 - \epsilon)^{M - m(x_k, \mathbf{x}_{\setminus k})} \tag{7.4}$$

where $m(x_k, \mathbf{x}_{\setminus k})$ is the number of unsatisfied clauses (corresponding to the deterministic problem as $\epsilon \to 0$). In the limit of determinism (i.e., $\epsilon \to 0$), if $m(x_k, \mathbf{x}_{\setminus k}) \neq M - m(x_k, \mathbf{x}_{\setminus k})$, then X_k is assigned the value which minimizes #Conflicts as that term dominates in the conditional probability. Otherwise, if $m(x_k, \mathbf{x}_{\setminus k}) = M - m(x_k, \mathbf{x}_{\setminus k})$, the limiting conditional distribution (as $\epsilon \to 0$) is the uniform distribution and can be obtained using L'Hospital's rule.

Since A_{MCMC} is Gibbs sampling and it converges[2] in behavior to the Min-Conflicts algorithm in the deterministic limit. Thus, it shares the same set of problems that Min-Conflicts has (i.e., getting stuck in a local maxima).

WalkSAT and WalkSAT-MCMC

We have already described the WalkSAT algorithm in Section 7.3. It is outlined in Algorithm 7.3.

Now to design A_{MCMC} (let us call this new algorithm WalkSAT-MCMC), we will first design a proposal distribution. Instead of choosing randomly among the currently unsatisfied clauses

[2] as defined in Section 7.3

Algorithm 7.3 — WalkSAT(CSP(**X**, **C**, S), α, iter)

x $\leftarrow$ Initialize from prior of **X**
for i = 1 to iter **do**
  Pick a clause $C \in$ **C** unsatisfied by **x**
  $r \leftarrow$ Uniform$(0, 1)$
  if $r < \alpha$ **then**
    $idx \leftarrow$ Uniform$(k : X_k \in C)$
  else
    $idx \leftarrow \text{argmin}_{k:X_k \in C}$ #Conflicts$(\neg x_k, \mathbf{x}_{\setminus k}, \mathbf{C})$
      (Break ties randomly)
  end if
  $x_{idx} \leftarrow \neg x_{idx}$
end for

(as in WalkSAT), we now choose a clause C_i with probability proportional to it being current unsatisfied, i.e., $p(C_i = 0|\mathbf{x})$. Once a clause C^* is selected, we select the literal in C^* randomly with probability α. With probability $1 - \alpha$, we flip a literal with probability proportional to the posterior of the proposed state. It is straightforward to see that this proposal scheme is identical to WalkSAT when $\epsilon = 0$. After proposing the new state $\mathbf{x}'$, we accept it using delayed rejection MH. This acceptance probability is represented by α_{DRMH}. If $\mathbf{x}'$ is not accepted, the next state is proposed using $\mathbf{x}'$ (just like in WalkSAT). However, the set of samples generated by the algorithm correspond to every accepted sample (whenever $last\_sample$ is renewed).

SampleSAT and Sample-SAT MCMC

SampleSAT (Wei *et al.*, 2004) is a more recently proposed algorithm, which improves over WalkSAT. The authors in (Wei *et al.*, 2004) showed that WalkSAT was exponentially more likely to generate certain solutions as compared to other solutions for a given SAT problem. SampleSAT adds an occasional Simulated Annealing (Van Laarhoven & Aarts, 1987) step to make things much more uniform. The details are outlined in Algorithm 7.5.

By now, we have already shown two examples. So we will keep the description for the design process of SampleSAT-MCMC (Algorithm 7.6) brief. Our proposal for the next state involves a mixture of simulated annealing and the proposal for WalkSAT-MCMC. The proposed sample is then accepted via delayed rejection MH.

Other noise models

Case I : $\epsilon = 0$**,** $\delta > 0$

In this case, all the proposed algorithms remain almost identical. The unsatisfied clause selection in WalkSAT-MCMC (and therefore in SampleSAT-MCMC) will be deterministic. This will be a good

Algorithm 7.4 — WalkSAT-MCMC(CSP(**X**, **C**, S), α, iter)

x ← Initialize from prior of **X**
for i = 1 to iter **do**
  accept ← False
  last_sample ← **x**
  rejected_samples ← $\emptyset$
  while accept = False **do**
    Pick a clause $C \in \mathbf{C} \propto p(C = 0|\text{last\_sample})$
    $r \leftarrow \text{Uniform}(0, 1)$
    if $r < \alpha$ **then**
      $idx \leftarrow \text{Uniform}(k : x_k \in C)$
    else
      $idx \propto \pi(\neg x_{idx}, \text{last\_sample}_{\setminus idx})$
    end if
    $\mathbf{x}' \leftarrow \{\neg x_{idx}, \text{last\_sample}_{\setminus idx}\}$
    $r \leftarrow \text{Uniform}(0, 1)$
    if $\alpha_{DRMH}(\text{last\_sample}, \text{rejected\_samples}) < r$ **then**
      last_sample ← $\mathbf{x}'$
      rejected_samples ← $\emptyset$
    else
      rejected_samples ← rejected_samples + $\mathbf{x}'$
    end if
  end while
end for

Algorithm 7.5 — SampleSAT(CSP(**X**, **C**, S), T, β, α iter)

x ← Initialize from prior of **X**
for i = 1 to iter **do**
  $r \leftarrow \text{Uniform}(0, 1)$
  if $r < \beta$ **then**
    x ← Simulated-Annealing($\mathbf{x}, T, \text{CSP}(\mathbf{X}, \mathbf{C}, S)$)
  else
    x ← WalkSAT-Step($\mathbf{x}, \text{CSP}(\mathbf{X}, \mathbf{C}, S), \alpha$)
  end if
end for

Algorithm 7.6 — SampleSAT-MCMC(CSP(**X**, **C**, S), T, β, α, iter)

```
1: x ← Initialize from prior of X
2: for i = 1 to iter do
3:    accept ← False
4:    last_sample ← x
5:    rejected_samples ← ∅
6:    while accept = False do
7:       r ← Uniform(0, 1)
8:       if r < β then
9:          x′ ← Simulated-Annealing(x, T, CSP(X, C, S))
10:      else
11:         Pick a clause C ∈ C ∝ p(C = 0|last_sample)
12:         r ← Uniform(0, 1)
13:         if r < α then
14:            idx ← Uniform(k : x_k ∈ C)
15:         else
16:            idx ∝ π(¬x_idx, last_sample_\idx)
17:         end if
18:         x′ ← {¬x_idx, last_sample_\idx}
19:      end if
20:      r ← Uniform(0, 1)
21:      if α_DRMH(last_sample, rejected_samples) < r then
22:         last_sample ← x′
23:         rejected_samples ← ∅
24:      else
25:         rejected_samples ← rejected_samples + x′
26:      end if
27:   end while
28: end for
```

proposal distribution for small values of δ, since we are essentially ignoring δ during our proposal. As δ becomes large, we might select the clause using either $p(C_i = 0|\mathbf{x})$ or $p(C_i = 0|S = 1)$ – generally using a mixture of these would be prudent.

Case II : $\epsilon > 0$**,** $\delta > 0$

In this case, computing $\pi(\mathbf{x}) = p(\mathbf{x}|S = 1)$ might be exponentially expensive since the set of clause variables **C** have to be marginalized. This marginalization was trivial when either $\epsilon = 0$ or $\delta = 0$. One potential solution is to update the clause variables using Gibbs sampling. The clause selection for the various A_{MCMC} algorithms can then proceed as before, as we use $\pi(\mathbf{x}, \mathbf{c})$ as a proxy for $\pi(\mathbf{x})$.

7.5 Problem Instance II - Sum constraint sampling

Problem Definition

Given s and k independent random variables $X_1, X_2, \cdots, X_k$ such that $X_i \ P_i(x)$, the sum constraint sampling involves sampling these variables from the distribution $P(x_1, \cdots, x_k | \sum_{i=1}^{k} x_i = s)$.

ECSS and ECSS-MCMC

Efficient solutions have been proposed to the sum constraint problem which use dynamic programming. The algorithm is called exact constrained sequential sampling algorithm (ECSS). The idea behind ECSS is to sample the variables sequentially from the marginal distribution i.e., X_i is sampled from $P(X_i|s, X_1, \cdots, x_{i-1})$. The marginal probability is computed using the following formula:

$$P(X_i = j|s, X_1, \cdots, x_{i-1}) \propto P(\sum_{l=1}^{k} X_l = s|x_1, \cdots, x_i)$$

where $P(\sum_{l=1}^{k} X_l = s|x_1, \cdots, x_i)$ is estimated using a dynamic programming approach.

For more details on A_{det}^{SC} refer to (Huseby *et al.*, 2004). In this chapter we will be discussing a solution to a stochastic version of the above problem where s is influenced by a multiplicative noise. In our setup $X_i \sim P_i(x)$ can take only discrete values.

Now we introduce stochasticity by adding multiplicative noise to S. Let O denote the variable corresponding to observed noisy value of the sum. The graphical model for this problem is shown in figure 7.2. We need to sample from the posterior distribution $P(x_1, \cdots, x_k|O = o)$ such that $X_i \ P_i(x)$, $S = \sum_{i=1}^{k} x_i$, $O = (1 + \epsilon S)$ where $\epsilon \ P_\epsilon(e)$ represents the noise model.

In ECSS-MCMC also variables are sampled sequentially. Let us first discuss the proposal mechanism. At every time step the proposed sample is generated by sampling the noise first. Noise is estimated by sampling ϵ from $P_\epsilon(e)$. $(x_1, \cdots, x_k)$ can then be sampled using ECSS from $P(X_1, \cdots, X_k | \sum_{i=1}^{k} X_i = s)$ where $s = o/(1 + \epsilon)$.

For a given current state $(y_1, \cdots, y_k, \epsilon_y)$ the probability of proposing a state $(x_1, \cdots, x_k, \epsilon_x)$,

$$q(x, y) \propto P_\epsilon(\epsilon_x) P(x_1, \cdots, x_k|o, \epsilon_x)$$

where $P(x_1, \cdots, x_k|o, \epsilon)$ is the product of the individual marginals computed in ECSS.

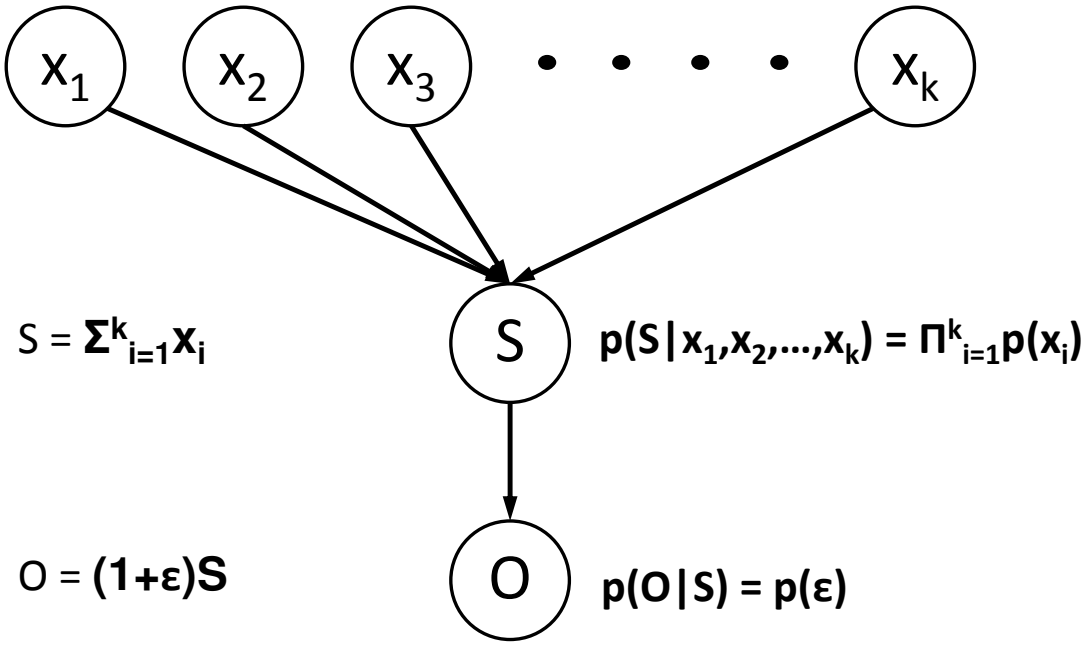


Figure 7.2: The graphical model for the sum constraint problem for discrete variables.

7.6 Experiments

Let us now evaluate the various algorithms presented and see if they can be used to design better-performing algorithms in the near-deterministic domain.

Stochastic SAT

For these experiments, we generated random $3 - SAT$ instances with 50 literals and 220 clauses. The number of clauses were chosen to increase the chances of the problem being satisfiable, but still close to unsatisfiable (Kamath *et al.*, 1995). For each clause in each problem instance, 3 literals were chosen and each of them was negated with probability 0.5. This defined P_{det}. The average performance of the three deterministic algorithms – Min-Conflicts, WalkSAT and SampleSAT over 5000 samples is shown in Figure 7.3. As expected, Min-Conflicts often gets stuck in local maxima, and has the worst performance. WalkSAT does much better while SampleSAT is the best

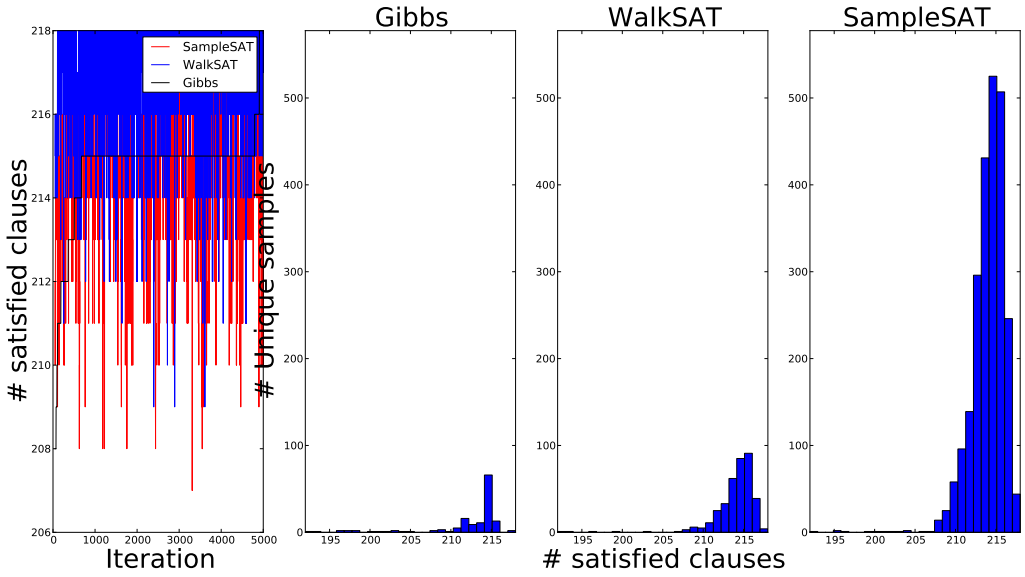


Figure 7.3: Average Performance of Min-Conflicts, WalkSAT and SampleSAT on a 50 literal, 220 clause $3 - SAT$ system. The leftmost figure tracks the number of satisfied clauses over iterations. The other three figures plot histograms of the number of unique samples in bins divided by number of satisfied clauses. It is evident that SampleSAT is the best performer, since it gets the most number of unique solutions, followed by WalkSAT and then Min-Conflicts.

performing algorithm, as existing literature suggests.

The 2 key metrics that we will measure in our experiments are the following:

1. The sample log likelihood, which is analogous to the data log-likelihood.
2. The number of unique samples generated by each algorithm in each log likelihood bin. A good algorithm should generate more unique samples in the high log likelihood bins.

Corresponding to each P_{det}, we defined 4 instances of $P_{near-det}$, for $\epsilon = \{0.1, 0.01, 0.001, 0.0001\}$. $\delta = 0$ for all the experiments. We also set uniform priors on all the literals (each of them is equally likely to be either 0 or 1).

The average performance of the three A_{MCMC} algorithms – Gibbs, WalkSAT-MCMC and SampleSAT-MCMC – are largely similar to the trends observed in the deterministic problem. Figures 7.6, 7.7, 7.8 and 7.9 show the details of the performance for $\epsilon = 0.1, 0.01, 0.001$ and 0.0001 respectively. As we had mentioned in Section 7.3, the properties of the deterministic algorithm that A_{MCMC} derives from, affects its performance. This is why SampleSAT-MCMC outperforms WalkSAT-MCMC and is likely to have a much smaller mixing time since SampleSAT visits solution states much more uniformly.

Sum constraint sampling

For experiments on the sum constraint problem, we once again generated random instances of the problem with 50 discrete random variables, each having a bimodal prior. ϵ had a Gaussian prior distribution with $\mu = 0$. We varied the variance $\sigma^2 = 0.1, 0.01, 0.001, 0.0001$. We noticed a marked performance improvement in the last two cases, which has been in Figures 7.4 and 7.5.

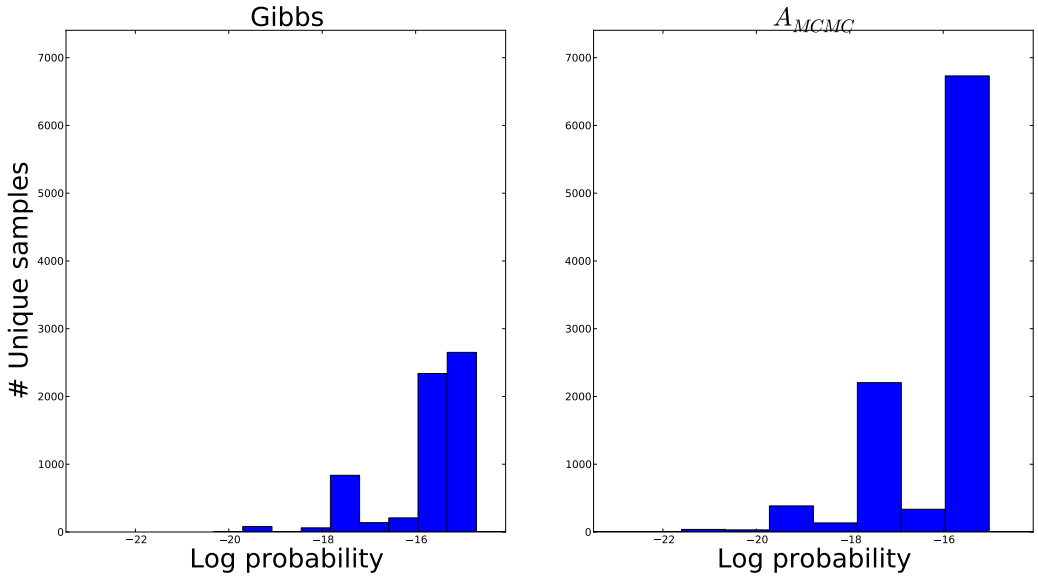


Figure 7.4: Average Performance of Gibbs sampling vs A_{MCMC} for the sum constraint sampling problem. This graph plots the number of unique samples in bins divided by log likelihood. $Var_\epsilon = 0.001$.

7.7 Discussion and Conclusion

In this chapter, we have identified a general procedure to design effective MCMC algorithms for the near-deterministic realm, by using algorithms for the corresponding deterministic problem. We compare the performance of these MCMC algorithms against Gibbs sampling, as the amount of time and effort to design and implement A_{MCMC} from A_{det} is arguably similar to that required to implement a Gibbs sampler for a particular problem. The A_{MCMC} algorithms outperform the Gibbs sampler as the amount of non-determinism decreases in the problem. We also identify certain properties of algorithms, which might make them better suited for our purpose. For instance, SampleSAT is a better choice than WalkSAT since the former samples solutions more uniformly, resulting in better mixing in SampleSAT-MCMC over WalkSAT-MCMC.

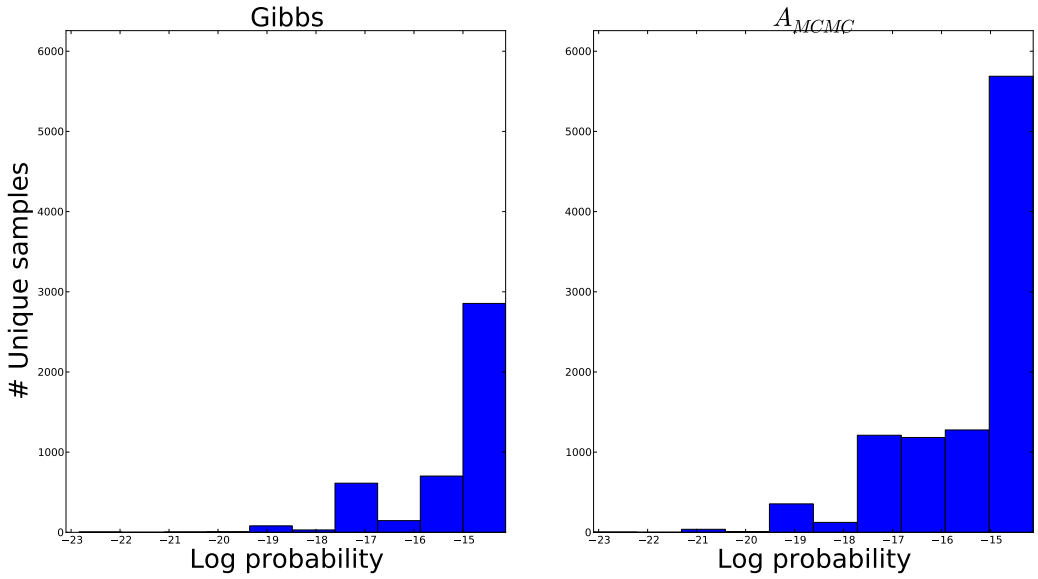


Figure 7.5: $Var_\epsilon = 0.0001$.

Also, similar ideas can be used to design MCMC optimization algorithms. Search heuristics like $k-opt$ can be used to design proposal moves which can be used in conjunction with simulated annealing to tackle optimization problems.

A potential downside of these algorithms would be their speed. Depending on A_{det} and $P_{near-det}$, computing $q(\mathbf{x}_i|\mathbf{x}_j)$ could be very expensive. Also, delayed rejection Metropolis Hastings slows down quickly as the length of rejected samples grows. However, an MCMC algorithm requires a lot more computations than a heuristic search algorithm – so perhaps this slowdown is inevitable, but the extent of it can be alleviated.

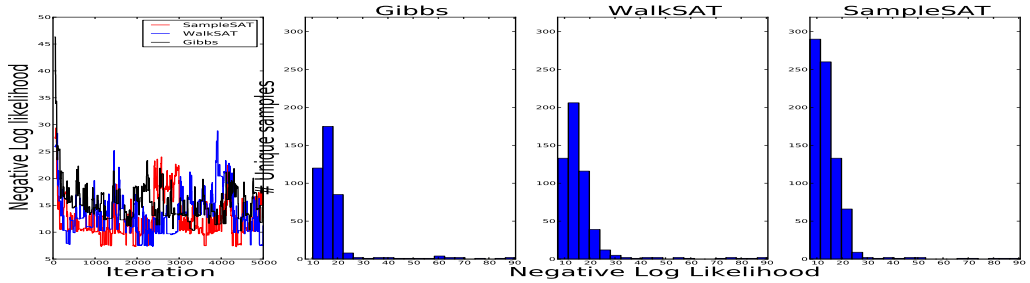


Figure 7.6: $\epsilon = 0.1$

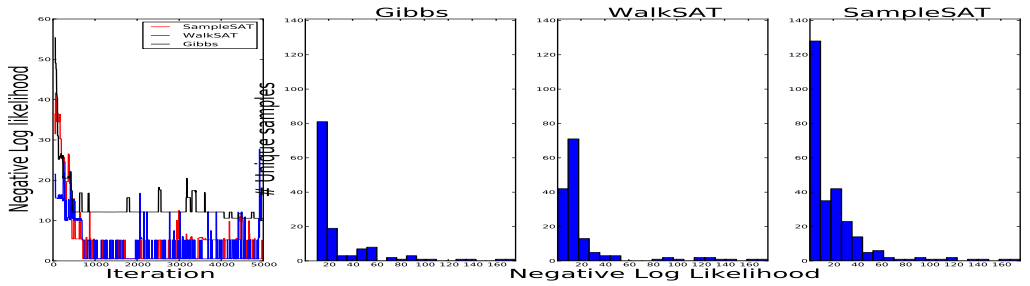


Figure 7.7: $\epsilon = 0.01$

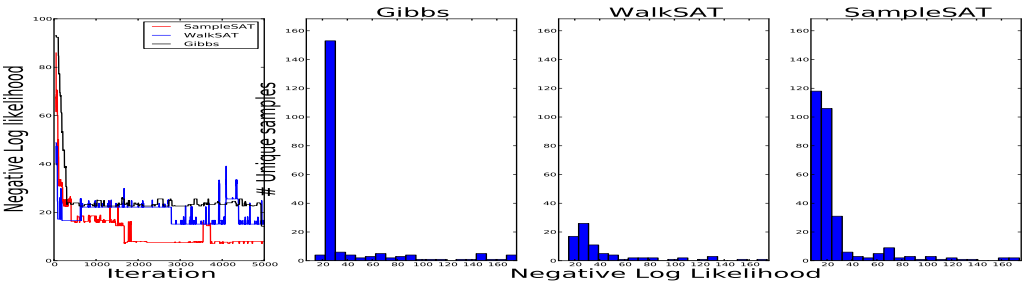


Figure 7.8: $\epsilon = 0.001$

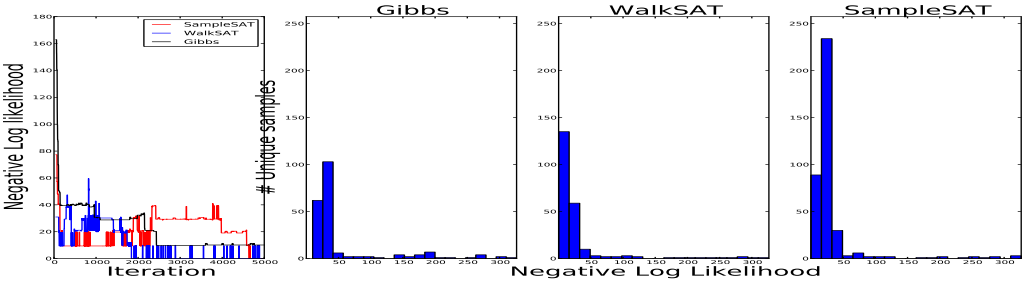


Figure 7.9: Comparison of the three algorithms – Gibbs, WalkSAT-MCMC, SampleSAT-MCMC. The first figure shows the sample likelihood (analogous to the data likelihood) of the three algorithms vs iteration. The next three graphs show histograms of unique samples generated by each algorithm. The y-axis denotes the number of unique samples generated, the x-axis denotes the negative log likelihood of the sample. This panel is for $\epsilon = 0.0001$.

Chapter 8

Conclusions

8.1 Summary

The primary contributions of this thesis are as follows: First, Chapter 3 investigates the effect of near-determinism on the structure of temporal graphical models, namely dynamic Bayesian networks (DBNs). We show that how the presence of widely varying evolution rates of variables leads to sparse graphical models over large time-steps. The sparsity results from approximating the distribution of the fast-evolving variables with the pseudo-equilibrium distribution conditioned on the slow-evolving variable state. This decoupled approximation is shown to be quite accurate, with the accuracy increasing with greater timescale separation between the slow and fast variables. Using this insight, we provide a general algorithm to choose an appropriate time-step and find the corresponding approximate graphical model (for the chosen time-step) given the original small time-step DBN. We also show that the approximation scheme works well on a human physiology model – namely, the blood pH control mechanism.

Second, Chapter 4 proposes an algorithm to approximately compute the few largest eigenvalues and eigenvectors of a factored linear system, whose full specification is exponential in size in the number of variables and hence too large to work with in its expanded form. We use a combination of numerical methods (Arnoldi iteration and QR algorithm) and graphical model inference (the junction tree algorithm) to compute the first few vectors in the relevant Krylov subspace while working with Kronecker product vectors to avoid exponential blowup in dimensionality. Although the results are mixed, this approach is a promising first step towards solving a critical problem.

Third, Chapter 5 studies the maximum a posteriori (MAP) estimation problem in a non-factored temporal model and proposes a hierarchical inference scheme to prune out large parts of the search space and find the provably optimal solution in (potentially) exponentially smaller time. The tem-

porally abstracted Viterbi (TAV) algorithm can provide significant speedups as shown in various experiments on synthetic datasets by using appropriate spatio-temporal abstractions.

The idea of coarse-to-fine iterative refinement, as used in the TAV algorithm, can also be applied to the problem of image/video segmentation as shown in Chapter 6. We propose a hierarchical video segmentation algorithm, which leverages existing work in computer vision for both abstraction creation (supervoxel tree) and inference (graph cuts) within each iteration.

Finally, Chapter 7 looks at the problem of approximate inference using sampling algorithms for near-deterministic probabilistic systems. Markov chain Monte Carlo (MCMC) algorithms are well-known to get stuck in local extrema and hence have large mixing times. This problem is exacerbated as the problem becomes more deterministic. Yet, several deterministic problems have efficient inference algorithms. We propose a general mechanism of designing MCMC algorithms which are designed to mimic the behavior of the deterministic domain algorithm for near-deterministic systems, and converge in behavior to the the deterministic domain algorithm in the limit of determinism.

The high level contribution is to demonstrate the possibility of exploiting near-determinism in probabilistic systems to speed up inference. Structure in causality can be expressed using graphical models and can be exploited by several algorithms designed for specific types of graphical models. Like causal sparsity, near-determinism provides another potential source for more efficient inference. This idea is applicable to all possible inference problems — exact and approximate inference, marginal and MAP estimation.

8.2 Future Work

Inference

Based on the current work, structural sparsity from large time-step DBNs are useful only when there is a timescale separation between the fast and slow sets of variables. If we can find a better algorithm to compute the eigenvalues of a DBN (i.e. , a factored system), then we can use the dominant eigen-pairs to make inference faster and more accurate.

In order to make the eigenvalue computation more accurate, we need to design an algorithm to obtain projections of negative element vectors through a factored representation of a linear model. We also need to improve the stochastic gradient based summation algorithm of factored Kronecker products in order to increase the accuracy of the Arnoldi iteration process.

Learning

In this dissertation, we have focused on the inference problem in near-deterministic systems. Similar ideas can be used to design better learning algorithms. Firstly, inference is often a sub-step of a learning algorithm like expectation-maximization (EM) (namely in the E-step, we can compute the expected value of the log likelihood function more efficiently for a near-deterministic system).

Another possibility is to explore better optimization algorithms where the posterior probability distribution on the parameter space is highly skewed. This is analogous to the image/video segmentation problem where the number of likely segmentations are far outnumbered by the possible number of segmentations.

8.3 Outlook

Probabilistic models are increasing in complexity and size to achieve higher accuracy and also to handle larger amounts of data. In order to scale inference, we need to exploit every possible avenue of computational saving. Near-determinism is a commonly found property in several stochastic systems. Using the various algorithms described in this dissertation, and other strategies yet to be devised, we can solve many interesting problems within feasible computational limits.

Bibliography

Aleks, N., Russell, S., Madden, M. G., Morabito, D., Staudenmayer, K., Cohen, M., & Manley, G. T. (2009). Probabilistic detection of short events, with application to critical care monitoring. In Koller, D., Schuurmans, D., Bengio, Y., & Bottou, L. (Eds.), *Advances in Neural Information Processing Systems 21*, pp. 49–56. MIT Press.

Andrieu, C., de Freitas, N., Doucet, A., & Jordan, M. I. (2003). An introduction to mcmc for machine learning. *Machine Learning*, *50*(1-2), 5–43.

Andrieu, C., & Thoms, J. (2008). A tutorial on adaptive mcmc. *Statistics and Computing*, *18*(4), 343–373.

Attias, H. (1999). Inferring parameters and structure of latent variable models by variational bayes. In *Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence*, pp. 21–30. Morgan Kaufmann Publishers Inc.

Baum, L. E., & Petrie, T. (1966). Statistical inference for probabilistic functions of finite state Markov chains. *Annals of Mathematical Statistics*, *41*.

Boros, E., & Hammer, P. L. (2002). Pseudo-boolean optimization. *Discrete Appl. Math.*, *123*(1-3), 155–225.

Boyen, X., & Koller, D. (1998). Tractable inference for complex stochastic processes. In *Uncertainty in Artificial Intelligence: Proceedings of the Fourteenth Conference*, Madison, Wisconsin. Morgan Kaufmann.

Boykov, Y., Veksler, O., & Zabih, R. (2001). Fast approximate energy minimization via graph cuts. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *23*(11), 1222–1239.

Brostow, G. J., Fauqueur, J., & Cipolla, R. (2009). Semantic object classes in video: A high-definition ground truth database. *Pattern Recognition Letters*, *30*(2), 88–97.

Chatterjee, S., & Russell, S. (2010). Why are DBNs sparse?. *Journal of Machine Learning Research - Proceedings Track*, *9*, 81–88.

Chatterjee, S., & Russell, S. (2011). A temporally abstracted viterbi algorithm. In Cozman, F. G., & Pfeffer, A. (Eds.), *UAI*, pp. 96–104. AUAI Press.

Chen, A., & Corso, J. (2010). Propagating multi-class pixel labels throughout video frames. In *Image Processing Workshop (WNYIPW), 2010 Western New York*, pp. 14 –17.

Cremers, D., & Soatto, S. (2005). Motion competition: A variational framework for piecewise parametric motion segmentation. *Int. Journal of Computer Vision*, *62*(3), 249–265.

Csardi, G., & Nepusz, T. (2006). The igraph software package for complex network research. *InterJournal, Complex Systems*, *1695*(5).

Darrel, T., & Pentland, A. (1991). Robust estimation of a multi-layered motion representation. In *IEEE Workshop on Visual Motion*, pp. 173–178.

Dean, T., & Kanazawa, K. (1989). A model for reasoning about persistence and causation. *Computational Intelligence*, *5*(3), 142–150.

Felzenszwalb, P., & Huttenlocher, D. (2004). Efficient Graph-Based Image Segmentation. *International Journal of Computer Vision*, *59*(2), 167–181.

Felzenszwalb, P., & Huttenlocher, D. (2006). Efficient belief propagation for early vision. *International Journal of Computer Vision*, *70*, 41–54. 10.1007/s11263-006-7899-4.

Felzenszwalb, P. F., & McAllester, D. (2007). The generalized A* architecture. *J. Artif. Int. Res.*, *29*, 153–190.

Fine, S., Singer, Y., & Tishby, N. (1998). The hierarchical hidden markov model: Analysis and applications. In *Machine Learning*, pp. 41–62.

Floros, G., & Leibe, B. (2012). Joint 2d-3d temporally consistent semantic segmentation of street scenes. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*.

Forney, G.D., J. (1973). The Viterbi algorithm. *Proceedings of the IEEE*, *61*(3), 268 – 278.

Friedman, N., Murphy, K., & Russell, S. J. (1998). Learning the structure of dynamic probabilistic networks. In *Uncertainty in Artificial Intelligence: Proceedings of the Fourteenth Conference*, Madison, Wisconsin. Morgan Kaufmann.

Galmar, E., Athanasiadis, T., Huet, B., & Avrithis, Y. S. (2008). Spatiotemporal semantic video segmentation. In *MMSP*, pp. 574–579. IEEE Signal Processing Society.

Geman, S., & Geman, D. (1984). Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *IEEE Trans. Pattern Anal. Mach. Intell.*, *6*(6), 721–741.

Ghahramani, Z., & Jordan, M. I. (1997). Factorial hidden Markov models. *Machine Learning*, *29*, 245–274.

Golub, G. H., & Van Loan, C. F. (2012). *Matrix computations*, Vol. 3. JHU Press.

Gómez-Uribe, C. A., Verghese, G. C., & Tzafriri, A. (2008). Enhanced identification and exploitation of time scales for model reduction in stochastic chemical kinetics. *The Journal of Chemical Physics*, *129*(24).

Green, P. J., & Mira, A. (2001). Delayed rejection in reversible jump metropolis–hastings. *Biometrika*, *88*(4), 1035–1053.

Grundmann, M., Kwatra, V., Han, M., & Essa, I. (2010). Efficient hierarchical graph-based video segmentation. In *Computer Vision and Pattern Recognition (CVPR 2010)*.

Guyton, A., & Hall, J. (1997). *Human Physiology and Mechanisms of Disease*. W.B. Saunders Company.

Hastings, W. K. (1970). Monte carlo sampling methods using markov chains and their applications. *Biometrika*, *57*(1), 97–109.

Holte, R. C., Perez, M. B., Zimmer, R. M., & MacDonald, A. J. (1996). Hierarchical A*: Searching Abstraction Hierarchies Efficiently. In *AAAI/IAAI, Vol. 1*, pp. 530–535.

Huseby, A. B., Naustdal, M., & Varli, I. D. (2004). System reliability evaluation using conditional monte carlo methods. *Statistical Research Report*, *2*.

Iwasaki, Y., & Simon, H. A. (1994). Causality and model abstraction. *Artif. Intell.*, *67*(1), 143–194.

Jensen, C. S., Kjærulff, U., & Kong, A. (1995). Blocking gibbs sampling in very large probabilistic expert systems. *International Journal of Human-Computer Studies*, *42*(6), 647–666.

Kalman, R. (1960). A new approach to linear filtering and prediction problems. *Journal of Basic Engineering*, *82*, 35–46.

Kamath, A., Motwani, R., Palem, K., & Spirakis, P. (1995). Tail bounds for occupancy and the satisfiability threshold conjecture. *Random Structures & Algorithms*, *7*(1), 59–80.

Klein, D., & Manning, C. D. (2003). A* Parsing: Fast Exact Viterbi Parse Selection. In *HLT-NAACL*.

Kumar, M., & Koller, D. (2009). MAP estimation of semi-metric MRFs via hierarchical graph cuts. In *Proceedings of the Twenty-fifth Conference on Uncertainty in AI (UAI)*.

Ladicky, L., Russell, C., Kohli, P., & Torr, P. (2009). Associative hierarchical CRFs for object class image segmentation.. In *IEEE Int. Conf. on Computer Vision*.

Laptev, I. (2005). On space-time interest points. *Int. Journal of Computer Vision*, *64(2-3)*, 107–123.

Laptev, I., & Lindeberg, T. (2003). Space-time interest points. In *IEEE International Conference on Computer Vision*, pp. 432–439.

Lempitsky, V. S., Vedaldi, A., & Zisserman, A. (2011). A pylon model for semantic segmentation. In *Neural Information Processing Systems*.

Liu, J. S. (1994). The collapsed gibbs sampler in bayesian computations with applications to a gene regulation problem. *Journal of the American Statistical Association*, *89*(427), 958–966.

Loan, C. V., & Pitsianis, N. (1992). Approximation with kronecker products. Tech. rep., Cornell University.

Lytynoja, A., & Milinkovitch, M. C. (2003). A hidden markov model for progressive multiple alignment. *Bioinformatics*, *19*(12), 1505–1513.

McAllester, D. A., Selman, B., & Kautz, H. A. (1997). Evidence for invariants in local search. In Kuipers, B., & Webber, B. L. (Eds.), *AAAI/IAAI*, pp. 321–326. AAAI Press / The MIT Press.

Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A. H., & Teller, E. (1953). Equation of State Calculations by Fast Computing Machines. *The Journal of Chemical Physics*, *21*(6), 1087–1092.

Michaelis, L., & Menten, M. (1913). Die kinetik der invertinwirkung. *Biochem.*, *49*, 333–369.

Mira, A. (2001). On metropolis-hastings algorithms with delayed rejection. *Metron*, *59*(3-4), 231–241.

Mooij, J. M. (2010). libDAI: A free and open source C++ library for discrete approximate inference in graphical models. *Journal of Machine Learning Research*, *11*, 2169–2173.

Murphy, K., & Weiss, Y. (2001). The factored frontier algorithm for approximate inference in dbns. In *Proceedings of the Seventeenth conference on Uncertainty in artificial intelligence*, pp. 378–385. Morgan Kaufmann Publishers Inc.

Murphy, K. P., Weiss, Y., & Jordan, M. I. (1999). Loopy belief propagation for approximate inference: An empirical study. In *Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence*, pp. 467–475. Morgan Kaufmann Publishers Inc.

Ng, A. Y., Jordan, M. I., & Weiss, Y. (2001). On spectral clustering: Analysis and an algorithm. In *NIPS*, pp. 849–856.

Nodelman, U., Shelton, C., & Koller, D. (2002). Continuous time Bayesian networks. In *Uncertainty in Artificial Intelligence: Proceedings of the Eighteenth Conference*, Edmonton, Alberta. Morgan Kaufmann.

Pavliotis, G. A., & Stuart, A. M. (2007). *Multiscale methods: Averaging and homogenization*. Springer.

Pearl, J. (1988). *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann.

Poon, H., & Domingos, P. (2006). Sound and efficient inference with probabilistic and deterministic dependencies. In *AAAI*, pp. 458–463. AAAI Press.

Rabiner, L., & Juang, B. (1986). An introduction to hidden Markov models. *ASSP Magazine, IEEE*, *3*(1), 4 – 16.

Rabiner, L. (1989). A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, *77*(2), 257 –286.

Rao, S., Tron, R., Vidal, R., & Ma, Y. (2010). Motion segmentation in the presence of outlying, incomplete, or corrupted trajectories. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, *32*(10), 1832–1845.

Raphael, C. (2001a). Coarse-to-Fine Dynamic Programming. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *23*, 1379–1390.

Raphael, C. (2001b). Coarse-to-Fine Dynamic Programming. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *23*, 1379–1390.

Richardson, M., & Domingos, P. (2006). Markov logic networks. *Machine learning*, *62*(1), 107–136.

Robert, C. P., & Casella, G. (2005). *Monte Carlo Statistical Methods (Springer Texts in Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Russell, S. J., & Norvig, P. (2010). *Artificial Intelligence - A Modern Approach (3. internat. ed.)*. Pearson Education.

Selman, B., Kautz, H. A., & Cohen, B. (1994). Noise strategies for improving local search. In Hayes-Roth, B., & Korf, R. E. (Eds.), *AAAI*, pp. 337–343. AAAI Press / The MIT Press.

Shi, J., & Malik, J. (1998). Motion segmentation and tracking using normalized cuts. In *IEEE Int. Conf. on Computer Vision*, pp. 1154–1160.

Srkk, S. (2013). *Bayesian Filtering and Smoothing*. Cambridge University Press.

Sutton, R. S., Precup, D., & Singh, S. P. (1999). Between MDPs and Semi-MDPs: A Framework for Temporal Abstraction in Reinforcement Learning. *Artif. Intell.*, *112*(1-2), 181–211.

Tierney, L. (1994). Markov Chains for Exploring Posterior Distributions. *The Annals of Statistics*, *22*(4), 1701–1728.

Tierney, L., & Mira, A. (1999). Some adaptive monte carlo methods for bayesian inference. *Statistics in Medicine*, *18*(1718), 2507–2515.

Van Laarhoven, P. J., & Aarts, E. H. (1987). *Simulated annealing*. Springer.

Vidal, R., Tron, R., & Hartley, R. (2008). Multiframe motion segmentation with missing data using PowerFactorization and GPCA. *International Journal of Computer Vision*, *79*(1), 85–105.

Viterbi, A. (1967). Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *Information Theory, IEEE Transactions on*, *13*(2), 260 – 269.

Wainwright, M. J., & Jordan, M. I. (2008). Graphical models, exponential families, and variational inference. *Foundations and Trends® in Machine Learning*, *1*(1-2), 1–305.

Wei, W., Erenrich, J., & Selman, B. (2004). Towards efficient sampling: Exploiting random walk strategies. In McGuinness, D. L., & Ferguson, G. (Eds.), *AAAI*, pp. 670–676. AAAI Press / The MIT Press.

Xing, E. P., Jordan, M. I., & Russell, S. (2002). A generalized mean field algorithm for variational inference in exponential families. In *Proceedings of the Nineteenth conference on Uncertainty in Artificial Intelligence*, pp. 583–591. Morgan Kaufmann Publishers Inc.

Xu, C., & Corso, J. J. (2012). Evaluation of super-voxel methods for early video processing. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pp. 1202–1209. IEEE.

Yin, G. G., & Zhang, Q. (2004). *Discrete-Time Markov Chains: Two-Time-Scale Methods and Applications*. Applications of Mathematics Stochastic Modelling and Applied Probability 55. Springer-Verlag.