GMTK: The Graphical Models Toolkit

October 1, 2002

Contents

Part I

Introduction

This document describes the use of the graphical models toolkit GMTK, and its supporting programs, which are a software package for graphical-model based speech recognition written by Jeff Bilmes and Geoff Zweig. At the moment, this document is in draft form, and there are certain sections that have not yet been written. This includes a proper introduction, and a general overview of graphical models and their use in automatic speech recognition (ASR), and a chapter on graphical models theory and algorithms. These chapters are forthcoming. Even though they are missing, however, the material that is presented here documents all the current features of GMTK, and is plenty of information to get going running experiments. The current distribution also includes a tutorial (on the Aurora 2.0 corpus). The tutorial along with this document should answer all questions that you have. Please feel free, however, to let me (JB) know about any bugs, corrections, or suggestions you have regarding this document, the tutorial, and/or the toolkit.

In this first chapter, we provide a brief overview of the various features in (GMTK). These features are fully elaborated upon in later chapters. Graphical models are a flexible, concise, and expressive probabilistic modeling framework with which one may rapidly specify a vast collection of statistical models. We begin with a brief description of the representational and computational aspects of the framework. Following that is a detailed description of GMTK's features, including a language for specifying structures and probability distributions, logarithmic space exact training and decoding procedures, the concept of switching parents, and a generalized EM training method which allows arbitrary sub-Gaussian parameter tying. Taken together, these features endow GMTK with a degree of expressiveness and functionality that significantly complements other publicly available packages. GMTK was recently used in the 2001 and 2002 Johns Hopkins Summer Workshops.

0.1 Introduction

Although the statistical approach to pattern classification has been an integral part of automatic speech recognition (ASR) for over 30 years, the general paradigm is in no way exhausted. Today, new and promising statistical models are proposed for ASR every year. Sometimes one can simulate these new models using hidden Markov model (HMM) toolkits, but in such cases the new models cannot stray far from the basic HMM methodology. More often, a new model requires significant modifications on top of existing and already complex software. This is inefficient because a large amount of human effort must be placed into building new systems without having any guarantees about their performance. Therefore it is important to develop an over-arching and unifying statistical framework within which novel ASR methods can be accurately, succinctly, and rapidly employed.

Graphical models (GMs) are such a flexible statistical framework. With GMs, one uses a graph to describe a statistical process, and thereby defines one of its most important attributes, namely conditional independence. Because GMs describe these properties visually, it is possible to rapidly specify a variety of models without much effort. Interestingly, GMs subsume much of the statistical underpinnings of existing ASR techniques — no other known statistical abstraction appears to have this property. For example, it has been shown that the standard HMM Baum-Welch algorithm is only a special case of GM inference [20]. More importantly, the space of statistical algorithms representable with a GM is enormous; much larger than what has so far been explored for ASR. The time therefore seems ripe to start seriously examining such models.

Of course, this task is not possible without a (preferably freely-available and open-source) toolkit with which one may maneuver through the model space easily and efficiently. This paper describes the first version of GMTK, an open source, publicly available toolkit for developing

graphical-model based speech recognition systems. GMTK is meant to complement rather than replace other publicly available packages — it has unique features, ones that are different from both standard ASR-HMM [21, 2, 3] and standard Bayesian network [1, 18] packages.

This chapter provides a brief overview of GMTK, its notation, algorithms, main features, and reports baseline GMTK results. Full documentation of these features begins in Chatper 1. Section 0.2 describes the main representational ability of graphical models including the meanings of graphs, factorization of joint probability distributions, conditional independence properties, and parameterizations. The section also outlines the many computational benefits offered by GMs. And Section 0.3 describes the main features of GMTK.

0.2 Representation and Computation

Two primary benefits offered by graphical models include representational ability and efficient algorithms for fast inference. This section briefly outlines them both.

0.2.1 Representation

A graphical model is a graph (i.e., a set of nodes and edges) that represents certain properties about sets of random variables. The nodes in the graph correspond to random variables, and the edges encode a set of conditional independence properties. These properties may be used to obtain a number of valid factorizations of the joint probability distribution. There are many different types of graphical models, such as Bayesian networks (a type of directed graphical model), Markov random fields (undirected models), causal models, chain graphs, and so on. Each type has its own formal semantics [17] for specifying conditional independence relations. Only along with its agreed upon semantics does a GM precisely specify conditional independence properties. Variables in a GM may either be observed (their values are known), or hidden. The name hidden Markov model, for example, results from there being a Markov chain consisting only of hidden variables.

The first version of GMTK uses the semantics of Bayesian networks (BNs) [19, 16]. This means that the graphs are directed, and conditional independence properties are determined by the notion of "d-separation" [19]: a set of variables A is conditionally independent of a set B given a set C if and only if A is d-separated from B by C. D-separation holds if and only if *all* paths that connect any node in A and any other node in B are blocked. A path is blocked if it has a node v with either: 1) the arrows along the path **do not** converge at v (i.e., serial ($\rightarrow v \rightarrow$) or diverging ($\leftarrow v \rightarrow$) arrows at v) and $v \in C$, or 2) the arrows along the path **do** converge at v (i.e., $\rightarrow v \leftarrow$), and neither v nor any descendant of v is in C. From d-separation, one may "read off" a list of conditional independence statements from a graph. The set of probability distributions for which this list of statements is true is precisely the set of distributions represented by the graph. Further, the joint probability distribution may be factored as the product of the probability of each variable's value, given the values of its parents in the graph.

Speech is a time signal, and any GM intending to model speech must somehow take this into account. Accordingly, dynamic Bayesian networks (DBNs) [13] are Bayesian networks which include directed edges pointing in the direction of time. Other than the existence of time-edges, DBNs have the same semantics as other BNs. Note also that a directed edge in a BN can encode either a random or a deterministic relationship between the two variables. In the latter case, if $A \rightarrow B$ exists in the network, B is a deterministic function of A, meaning that B given A is a constant random variable.

The structure of a graphical model represents the way in which a set of random variables probabilistically represents natural and artificial processes (such as the existence of articulatory or noise-type variables), and how these variables interact (such as an n^{th} order Markov chain, tree-based dependencies, and so on). The structure also represents constraints on variable values and sequences of values (such as valid phone sequences in a speech recognizer). These representational aspects of graphical modeling are detailed in [22, 5].

Lastly, ASR is inherently a problem of pattern classification, and requires statistical models to discriminate between different speech utterances. Apart from discriminatively learned model parameters (such as means, variances, or transition matrices), graphical models are ideally suited for experimenting with discriminative structures [10, 23].

0.2.2 Computation

Probabilistic inference, such as evaluating (or computing the most likely value of) a conditional distribution, is the foundation behind all statistical computing. Graphical models have an associated set of algorithms which perform inference as efficiently as possible. Apart from describing the structure of a domain, conditional independence can lead to enormous computational savings when doing inference. For example, computing the quantity $p(y|x)$ could be done the hard way $p(y|x) = \sum_{a,b} p(y, a, b|x)$ or the easy way $p(y|x) = \sum_a p(y|a) \sum_b (a|b)p(b|x)$, the latter case assuming independence relations making the computation probabilistically valid. Graphical models use inference algorithms (e.g., the junction-tree algorithm [19, 16] or the generalized distributed law [4]) that provably correspond to valid calculations on probabilistic equations. These algorithms essentially distribute summations to the right into products as efficiently as possible, as above. There are many ways of doing this, one of which is used in GMTK (and described in Section 0.3.3). Other approaches forfeit exact inference for the sake of speed, and resort instead to approximate methods. In any case, when efficient and accurate probabilistic inference is required, GMs provide numerous possibilities.

0.3 Toolkit Features

GMTK has a number of features that support a wide array of statistical models suitable for speech recognition and other time-series data. GMTK may be used to produce a complete ASR system for both small- and large-vocabulary domains. The graphs themselves may represent everything from N-gram language models down to Gaussian components, and the probabilistic inference mechanism supports first-pass decoding in these cases.

0.3.1 Explicit vs. Implicit Modeling

In general, there are two representational extremes one may employ when using GMTK for an ASR system. On the one hand, a graph may explicitly represent all the underlying variables and control mechanisms (such as sequencing) that are required in an ASR system [22]. We call this approach an "explicit representation" where variables can exist for such purposes as word identification, numerical word position, phone or phoneme identity, the occurrence of a phoneme transition, and so on. In this case, the structure of the graph explicitly represents the interesting hidden structure underlying an ASR system. On the other hand, one can instead place most or all of this control information into a single hidden Markov chain, and use a single integer state to encode all contextual information and control the allowable sequencing. We call this approach an "implicit" representation.

As an example of these two extremes, consider the word "yamaha" with pronunciation /y aa m aa hh aa/. The phoneme /aa/ occurs three times, each in different contexts, first preceding an /m/, then preceding an /hh/, and finally preceding a word boundary. In an ASR system, it must somewhere be specified that the same phoneme /aa/ may be followed only by one of /m/, /h/, or a word boundary depending on the context — /aa/, for example, may not be followed by a word boundary if it is the first /aa/ of the word. In the explicit GM approach, the graph and associated conditional probabilities unambiguously represent these constraints. In an implicit approach, all of the contextual information is encoded into an expanded single-variable hidden state space, where multiple HMM states correspond to the same phoneme /aa/ but in different contexts.

The explicit approach is useful when modeling the detailed and intricate structures of ASR. It is our belief, moreover, that such an approach will yield improved results when combined with a discriminative structure [5, 10, 23], because it directly exposes events such as word endings and phone transitions for use as switching parents (see Section 0.3.4). The implicit approach is useful for tempering computational and/or memory requirements. In any case, GMTK supports both extremes and everything in between — a user of GMTK is therefore free to experiment with quite a diverse and intricate set of graphs. It is the task of the toolkit to derive an efficient inference procedure for each such system.

0.3.2 The GMTKL Specification Language

A standard DBN [13] is typically specified by listing a collection of variables along with a set of intra-frame and inter-frame dependencies which are used to unroll the network over time (see Section 2 for more detail). GMTK generalizes this ability via dynamic GM templates. The template defines a collection of (speech) frames and a chunk specifier. Each frame declares an arbitrary set of random variables and includes attributes such as parents, type (discrete, continuous), parameters to use (e.g. discrete probability tables or Gaussian mixtures) and parameter sharing. At the end of a template is a chunk specifier (two integers, $N : M$) which divides the template into a prologue (the first $N - 1$ frames), a repeating chunk, and an epilogue (the last $T - M$ frames, where T is the frame-length of the template). The middle chunk of frames is "unrolled" until the dynamic network is long enough for a specific utterance.

GMTK uses a simple textual language (GMTKL) to define GM templates. Figure 0.3.2 shows the template of a basic HMM in GMTKL. It consists of two frames each with a hidden and an observed variable, and dependences between successive hidden and between observed and hidden variables. For a given template, unrolling is valid only if all parent variables in the unrolled network are compatible with those in the template. A compatible variable has the same name, type, and cardinality. It is therefore possible to specify a template that can not be unrolled and which would lead to GMTK reporting an error.

A template chunk may consist of several frames, where each frame contains a different set of variables. Using this feature, one can easily specify multi-rate GM networks where variables occur over time at rates which are fractionally but otherwise arbitrarily related to each other.

0.3.3 Inference

GMTK supports a number of operations for computing with arbitrary graph structures, the four main ones being:

1. Integrating over hidden variables to compute the observation probability: $P(\mathbf{o}) = \sum_{\mathbf{h}} P(\mathbf{o}, \mathbf{h})$

```
frame: 0 {
   variable : state {
      type : discrete hidden cardinality 4000;
      switchingparents : nil;
      conditionalparents : nil using MDCPT("pi");
   }
   variable : observation {
      type : continuous observed 0:38;
      switchingparents : nil;
      conditionalparents : state(0)
         using mixGaussian mapping("state2obs");
   }
}

frame: 1 {
   variable : state {
      type : discrete hidden cardinality 4000;
      switchingparents : nil;
      conditionalparents : state(-1)
         using MDCPT("transitions");
   }
   variable : observation {
      type : continuous observed 0:38;
      switchingparents : nil;
      conditionalparents : state(0)
         using mixGaussian mapping("state2obs");
   }
}

chunk 1:1;
```

Figure 1: GMTKL specification of an HMM structure. The feature vector in this case is 39 dimensional, and there are 4000 hidden states. Frame 1 can be duplicated or ”unrolled” to create an arbitrarily long network.

2. Finding the likeliest hidden variable values: $argmax_{\mathbf{h}} P(\mathbf{o}, \mathbf{h})$
3. Sampling from the joint distribution $P(\mathbf{o}, \mathbf{h})$
4. Parameter estimation given training data $\{\mathbf{o}_k\}$ via EM/GEM: $argmax_\theta \prod_k P(\mathbf{o}_k|\theta)$ The GEM algorithm will be defined in Section 0.3.3.

A critical advantage of the graphical modeling framework derives from the fact that these algorithms work with *any* graph structure, and a wide variety of conditional probability representations. GMTK currently uses the *Frontier Algorithm*, detailed in [22, 26], which converts arbitrary graphs into equivalent chain-structured ones, and then executes a forwards-backwards recursion. The chain structure easily supports beam-pruning, allows deterministic relationships between variables to be immediately identified and exploited, and, as we see in the next section, allows for easy implementation of exact inference in logarithmic space. In a future version of GMTK, it will support more typical junction trees, which can have smaller clique sizes and therefore better complexity.

Logarithmic Space Computation

In many speech applications, observation sequences can be thousands of frames long. When there are a dozen or so variables per frame (as in an articulatory network), the resulting unrolled network might have tens of thousands of nodes, and cliques may have millions of possible values. A naive implementation of exact inference, which stores all clique values for all time, would result in (an obviously prohibitive) gigabytes of required storage

To avoid this problem, GMTK implements a recently developed procedure [12, 24] that reduces memory requirements exponentially from $O(T)$ to $O(\log T)$. This reduction has a truly dramatic effect on memory usage, and can additionally be combined with GMTK's beam-pruning procedure for further memory savings. The key to this method is recursive divide-and-conquer. With k-way splits, the total memory usage is $O(k \log_k T)$, and the runtime is $O(T \log_k T)$. The constant of proportionality is related to the number of entries in each clique, and becomes smaller with pruning. For algorithmic details, the reader is referred to [24].

Generalized EM

GMTK supports both EM and generalized EM (GEM) training, and automatically determines which to use based on the parameter sharing currently in use. GEM training is distinctive because it provides a provably convergent method for parameter estimation, even when there is an arbitrary degree of tying, even down to the level of Gaussian means, covariances, or factored covariance matrices (see Section 0.3.6).

Sampling

Drawing variable assignments according to the joint probability distribution is useful in a variety of areas ranging from approximate inference to speech synthesis, and GMTK supports sampling from arbitrary structures. The sampling procedure is computationally inexpensive, and can thus be run many times to get a good distribution over hidden (discrete or continuous) variable values.

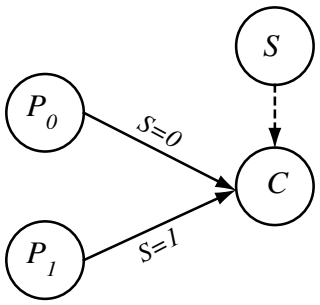

Figure 2: When $S = 1$, A is B's parent, when $S = 2$, B is C's parent. S is called a switching parent, and A and B conditional parents.

0.3.4 Switching Parents

GMTK supports another novel feature rarely found in GM toolkits, namely switching parent functionality (also called Bayesian multi-nets [10]). Normally, a variable has only one set of parents. GMTK, however, allows a variable's parents to change (or switch) conditioned on the current values of other parents. The parents that may change are called conditional parents, and the parents which control the switching are called switching parents. Figure 2 shows the case where variable S switches the parents of C between A and B, corresponding to the probability distribution: $P(C|A, B) = P(C|A, S = 1)P(S = 1) + P(C|B, S = 2)P(S = 2)$. This can significantly reduce the number of parameters required to represent a probability distribution, for example, $P(C|A, S = 1)$ needs only a 2-dimensional table whereas $P(C|A, B)$ requires a three dimensional table. Switching functionality has found particular utility in representing certain language models, as experiments during the JHU2001 workshop demonstrated.

0.3.5 Discrete Conditional Probability Distributions

GMTK allows the dependency between discrete variables to be specified in one of three ways. First, they may be deterministically related using flexible n-ary decision trees. This provides a sparse and memory-efficient representation of such dependencies. Alternatively, fully random relationships may be specified using dense conditional probability tables (CPTs). In this case, if a variable of cardinality N has M parents of the same cardinality, the table has size N^{M+1}. Since this can get large, GMTK supports a third sparse method to specify random dependencies. This method combines sparse decision trees with sparse CPTs so that zeros in a CPT simply do not exist. The method also allows flexible tying of discrete distributions from different portions of a CPT.

0.3.6 Continuous Conditional Probability Distributions

GMTK supports a variety of continuous observation densities for use as acoustic models. Continuous observation variables for each frame are declared as vectors in GMTKL, and each observation vector variable can have an arbitrary number of conditional and switching parents. The current values of the parents jointly determine the distribution used for the observation vector. The mapping from parent values to child distribution is specified using a decision tree, allowing a sparse representation of this mapping. A vector observation variable spans over a region of the feature vector at the current time. GMTK thereby supports multi-stream speech recognition, where each stream may have its own set of observation distributions and sets of discrete parents.

The observation distributions themselves are mixture models. GMTK uses a splitting and vanishing algorithm during training to learn the number of mixture components. Two thresholds are defined, a mixture-coefficient vanishing ratio (mcvr), and a mixture-coefficient splitting ratio (mcsr). Under a K-component mixture, with component probabilities p_k, if $p_k < 1/(K \times \text{mcvr})$, then the k^{th} component will vanish. If $p_k > \text{mcsr}/K$, that component will split. GMTK also supports forced splitting (or vanishing) of the N most (or least) probable components at each training iteration. Sharing portions of a Gaussian such as means and covariances can be specified either by hand via parameter files, or via a split (e.g., the split components may share an original covariance).

Each component of a mixture is a general conditional Gaussian. In particular, the probability density of the c^{th} component is $p(x|z_c, c) = \mathcal{N}(x|B_c z_c + f_c(z_c) + \mu_c, D_c)$ where x is the current observation vector, z_c is a c-conditioned vector of continuous observation variables that might come from any observation stream and/or from the past, present, or future, B_c is an arbitrary sparse matrix, $f_c(z_c)$ is a multi-logistic non-linear regressor, μ_c is a constant mean residual, and D_c is a diagonal covariance matrix. Any of the above terms may be tied across multiple distributions, and trained using the GEM algorithm.

GMTK treats Gaussians as directed graphical models, and can thereby represent all possible Gaussian factorization orderings, and all subsets of parents in any of these factorizations. Under this framework, GMTK supports diagonal, full, banded, and semi-tied factored sparse inverse covariance matrices [11]. GMTK can also represent arbitrary switching dependencies between individual elements of successive observation vectors. GMTK thus supports both linear and non-linear buried Markov models [9]. All in all, GMTK supports an extremely rich set of observation distributions.

Part II

The Toolkit

Chapter 1

Toolkit Overview

As mentioned above, the Graphical Models Toolkit (GMTK) is a software system for developing graphical-model based speech recognition and general time series systems [6, 22, 8, 7, 25, 9]. This set of chapters provides a detailed description of GMTK's features, including its textual language for specifying graphical structures and probability distributions, and how to specify the parameters that must accompany these structures. The graphs themselves may represent everything from types of N-gram language models down to Gaussian components, and the probabilistic inference mechanism supports first-pass decoding for any of these cases.

Chapter 2

Representing Structure in GMTK

The next most crucial property of any graphical model after its semantics has been specified is its structure. The structure of a graphical model specifies the set of nodes and edges that constitute a graph. It is therefore important to have a natural and relatively easy-to-use method with which to specify these structures.

There are a number of possible ways of doing this. For example, given the number of nodes (say N), it is possible to specify an $N \times N$ adjacency matrix where a non-zero in the matrix position i, j says that there is a directed edge from parent i to child j. Alternatively, a textual language can be used to specify structure. A third possible way is to use a visual graphical user interface, where pointers, menus, and on-screen icons are used to specify structure.

As mentioned above, a dynamic Bayesian network (DBN) [13] is a normal Bayesian network but with certain edges pointing in the direction of "time" (or along some common spatial or axial dimension). While the standard ways of specifying structure (above) could be used, it is often not known in advance the length T of the time dimension. Moreover, in a DBN, it is typical for there to be a common core structure that is repeated T times. Therefore, a method is required that specifies a structure "template."

A template determines a "rolled up" structure, so that when the actual time length T becomes known, the template is unrolled sufficiently so that the network spans the entire length. When a network is unrolled, the template structure is essentially repeated T times, one repetition for each time "frame" or time slice. In GMTK, the notion of a frame will be the same as is typical in automatic speech recognition (ASR) systems.[1] Note that the length T might change for each use of the template. In ASR, for example, T would be different for each training and testing utterance. In the current version of GMTK, it is assumed that T is known for each utterance. Therefore, GMTK is suitable for fixed frame decoding and/or rescoring of large-vocabulary systems.[2]

With most dynamic Bayesian networks (DBNs), a structure template is specified by listing a collection of N variables that are to exist at each time frame. Along with these variables, a set of intra-frame and inter-frame dependencies used to unroll the template are also given. When the template is unrolled, the set of variables are repeated T times, leading to a network with a total of TN variables for. The intra-frame dependencies in the template are repeated along with the set of variables, essentially "rubber-stamping" the variables and their edges. The inter-frame dependencies are then repeated $T-1$ times, and are used to connect pairs of nodes in neighboring frames. This is depicted in Figure 2.1. GMTK has a textual template specification language that

[1] In ASR, this is often a 25ms sliding time window, with a 10ms time-step between windows. This is only a convention, however, any frame width and step can be used.

[2] Future versions of GMTK will allow for dynamic unrolling of the network, where T is not known, and decoding proceeds before reaching the end of the utterance.

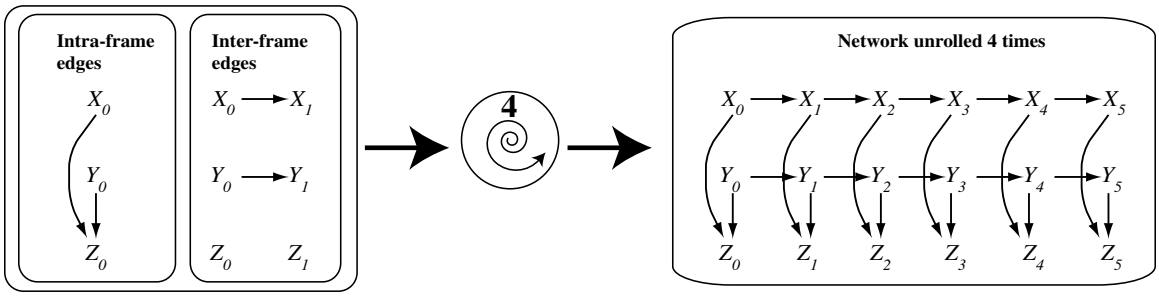

Figure 2.1: A "rolled" and unrolled network in a typical dynamic Bayesian network. On the left, we see two networks. The first one on the left shows the intra-frame edges, and the other shows the inter-frame edges. On the right is the network that has been unrolled 4 (four) times. Note that this uses the convention that a network can be unrolled 0 (zero) times; in this case, the resulting network is identical to what would be obtained by applying the intra-frame edges to the inter-frame network. In the figure, therefore, a network unrolled zero times would consist only of two frames.

generalizes the typical intra-frame and inter-frame dependencies used for DBN specification. This is described in the following section.

2.1 GMTKL: The GMTK Structure Language

GMTKL is a structure specification language for specifying templates in GMTK. GMTKL is a generalization of the standard DBN methodology to specify and then unroll a template.

A GMTKL template declares a list of (time) frames and a chunk specifier. Each frame defines an arbitrary collection of random variables and includes attributes such as the variable's parents, switching parents, type (discrete or continuous), parameters and implementation (e.g. discrete probability tables, deterministic relationship, decision trees, Gaussian mixtures, etc.). A frame in a template might not have the same nodes and edges as other frames, thereby making GMTKL quite flexible in the graphs that it can represent.

At the end of a template is a chunk specifier, two integers $N : M$ where $M \geq N$, which divides the template into a prologue (the first N frames), a repeating chunk (consisting of template frames N through M inclusive), and an epilogue (the last $T_t - M - 1$ frames, where T_t is the number of frames of the template). The middle $M - N + 1$ chunk of frames is "unrolled" or repeated a number of times until the dynamic network is long enough to fill up the frames of a specific utterance. GMTK's generalization into a prolog, chunk, and epilogue of the a standard DBN notion of intra- and inter-frame edges can be quite useful in specifying a wide variety of both desirable and computationally tractable structures, as will be seen in this document.

Figure 2.2 shows an example of a GMTKL template for a simple HMM. The template consists only of two frames (frame 0 and frame 1) each with a hidden variable `state` and an observed variable `observation`. In both frames, `state` is a hidden discrete variable with cardinality 4000, which means that it can have at most the 4000 values 0 through 3999. `state` has no switching parents (to be described below). At time frame 0, `state` has no parents at all, and is distributed according to a single one-dimensional dense conditional probability table (CPT) named `pi`, that must contain 4000 probability values. GMTK probabilities and other parameters are not represented in a structure file, and instead are kept elsewhere (see Section 3).

At frame 1, `state` now has a parent which is the variable `state` at the preceding frame. This

```
%%% This is a comment %%%
% Simple GMTKL template for an HMM.

% The first frame, starting at index 0
frame: 0 {

   % variable 'state' in first frame
   variable : state {
      type : discrete hidden cardinality 4000;
      % state at frame 0 has no switching parents.
      switchingparents : nil;
      % state at frame 0 has no normal parents.
      % 'pi' is the name of the conditional probability table (CPT)
      % containing the parameters for this variable. Since
      % the variable has no parents, it is only a 1-D table.
      conditionalparents :
            nil using DenseCPT("pi");
   }

   % variable observation in first frame
   variable : observation {
      % 'observation' is continuous 39-dimensional feature
      % vector. '0:38' refers to elements in the observation file.
      type : continuous observed 0:38;
      % 'observation' at frame 0 has one parent, 'state(0)'.
      % Note that 'state(0)' refers to the state variable
      % in the current frame.
      switchingparents : nil;
      conditionalparents : state(0)
         % collections and mappings will be described later,
         % when we talk about parameters.
         using collection("global")
         mixGaussian mapping("state2obs");
   }
}
% The second frame
frame: 1 {

   % variable 'state' in second frame
   variable : state {
      type : discrete hidden cardinality 4000;
      switchingparents : nil;
      % In this case, state in the second frame has
      % as a parent 'state(-1)' which means the state
      % variable one time frame preceeding this one. In other
      % words, when a variable named 'var' is specified
      % as 'var(n)' to designate a parent, the 'n' is
      % the relative frame offset. If n=0, it refers

      % to the variable named 'var' in the current frame,
      % if n=-1, it refers to the previous frame. Other
      % values of n (both positive and negative) are also
      % possible.
      conditionalparents : state(-1)
         using DenseCPT("transitions");
   }

   % variable 'observation' in second frame
   variable : observation {
      type : continuous observed 0:38;
      % 'observation' at frame 1 has one parent, 'state(0)'
      % Note that 'state(0)' refers to the state variable
      % in the current frame, which in this case is frame 1.
      switchingparents : nil;
      conditionalparents : state(0)
         using collection("global")
         mixGaussian mapping("state2obs");
   }
}
% The chunk specifier, saying only that
% frame 1 is the chunk to be repeated.
chunk 1:1;
```

Figure 2.2: GMTKL specification of a simple HMM structure. The feature vector in this case is 39 dimensional, and the total possible number of hidden states is 4000. The chunk specifier is $1:1$ which means that frame 1 is duplicated or "unrolled" enough times to create an arbitrarily long network structure.

is specified using the relative integer frame offset -1. In general, variables are defined in a frame, using the `variable : varname` notation, without using any relative frame offset. Different frames can specify the same variable name, and it is the frame location that disambiguates multiple variables of the same name (but in different frames). Two variables may *not* have the same name within the same frame.

When a name of a parent variable is to be specified, the parent variable name uses a relative frame offset to specify, relative to the frame of the current child, which variable is the parent. The syntax is `varname(n)` where n is either zero (the same frame as the current child), negative (one or more frames **before** (earlier than) the current frame), or positive (one or more frames **after** (later than) the current frame). This and the above mean, for example, that a variable can not have a parent variable both with the same name and with an offset of 0.

In frame 1 in Figure 2.2, the `state` variable uses a dense CPT called `transitions`. Unlike in frame 0, in frame 1 this must be a two-dimensional 4000×4000 CPT, specified in the parameter files. This is because the `state` variable in frame 1 has as a parent in frame 0 named 'state' (i.e., `state(-1)`) which has cardinality 4000. The `state` variable in frame 0, on the other hand, has no parents.

The `observation` variable is also specified in both frames, and in each case `observation` uses the current `state` variable as a parent. `observation` is a continuous and observed variable. Moreover, it is a 39-dimensional vector observation, as specified by the string `0:38`. This is a range specification which says that this variable corresponds to features 0 through 38 in the observation feature file (to be described in Section 4.7). For this `observation` variable, the parent `state(0)` affects the child variable by determining which of a number of Gaussian mixtures to use, one possible mixture for each value of `state(0)`. This mapping, from the value of a discrete variable `state(0)` to a particular Gaussian mixture is specified using the decision tree mapping `state2obs` and via the collection of names `global`. These are described in later sections.

The following is an annotated grammar that describes the complete GMTKL syntax in great detail. Those of you familiar with this notation can quickly determine what syntactic constructs are allowed in GMTKL. Later sections will describe each case with a number of examples. In the example, lines that start with the % character are comments.

```
% This is a comment, as is any line started with a '%' character

% All GM structure files must start with the
% "GRAPHICAL_MODEL" magic string. This is followed
% by a list of frames and ends with a chunk specifier.
GM = "GRAPHICAL_MODEL" identifier FrameList ChunkSpecifier ;

% A list of frames.
FrameList = Frame FrameList
          | NULL ;


% A frame consists of a list of random variables.
Frame = "frame" ":" integer "{" RandomVariableList "}" ;

RandomVariableList = RandomVariable RandomVariableList
                   | NULL ;

% A random variable starts with its name such as "variable : foo"
% followed by a list of random variable attributes.
RV = "variable" ":" name "{" RandomVariableAttribute "}" ;

RandomVariableAttributeList =
```

```
        RandomVariableAttribute RandomVariableAttributeList
       | NULL ;

% Random variable attributes can either specify the random
% variable type, or might specify parents and parameters.
RandomVariableAttribute = TypeAttribute
                        | ParentsAttribute ;


TypeAttribute = "type" ":" RandomVariableType ";"

% A random variable is either of discrete or continuous type.
RandomVariableType = RandomVariableDiscreteType
                   |  RandomVariableContinuousType ;

% A discrete random variable is either
%  1) hidden
%  2) observed, in which case you must specify either
%       a) a observation range of size 1 corresponding
%          to the observed values of this variable. The
%          range takes the form n:n, where n is the index of the
%          element in the observation file to obtain the value
%          of this variable for each frame.
%   or
%       b) a fixed immediate and inline value of this observed
%          random variable.
% In either case, you also provide the RV cardinality.
RandomVariableDiscreteType =
    "discrete"
    ( "hidden" | "observed" (integer ":" integer | "value" integer) )
     "cardinality" integer ;

% A continuous random variable is either
%  1) hidden (not currently supported in GMTK)
%  2) observed, in which case you must specify
%  an observation range. The observation range takes
%  the form of n:m and determines the
%  size of the continuous observation vector.
RandomVariableContinuousType =
    "continuous"  ("hidden" | "observed" integer ":" integer)  ;


% A parent can be either switching or conditional. A conditional
% parent is identical to the normal notion of a parent
% in a Bayesian network.
ParentsAttribute =
      ( "switchingparents" ":" SwitchingParentAttribute ";" )
   |  ( "conditionalparents" ":" ConditionalParentSpecList  ";" )  ;


% A variable may have a list of switching parents, or
% if no switching parents are desired, use just 'nil'
SwitchingParentAttribute =
     "nil"
   | ParentList "using" MappingSpec ;

% For each vector value in the joint state space of the set of
% switching variable values, a decision tree maps that vector
% value down to an integer. For each possible integer value,
% there is a collection of conditional parents.
ConditionalParentSpecList =
     ConditionalParentSpec "|" ConditionalParentSpecList
  |  ConditionalParentSpec ;


% A set of conditional parents consists of a list
% of parent specifiers, and an implementation.
ConditionalParentSpec = ConditionalParentList using Implementation ;
```

```
ConditionalParentList = "nil"
                      | ParentList ;

ParentList =
       Parent "," ParentList
     | Parent ;

% A parent consists of a variable identifier, and an integer
% frame offset.
Parent = identifier "(" integer ")" ;

Implementation = DiscreteImplementation
               | ContinuousImplementation ;

% A discrete implementation can be either
%  1) a Dense conditional probability table (CPT)
%  2) a Sparse CPT
%  3) a deterministic CPT (only one non-zero probability
%     value per parent value set).
DiscreteImplementation =
    ( "DenseCPT" | "SparseCPT" | "DeterministicCPT" )  "(" ListIndex ")" ;


% There are two forms of a ContinousImplementation:
% 1) The first case uses the syntax
%
%          "(" ListIndex ")"
%
% This is used if there are no conditional parents (i.e., specified as
% nil). Even when there are no conditional parents, a distribution must
% be specified. 'ListIndex' above refers directly to a particular
% gaussian mixture, using the normal 'ListIndex' format (a string
% identifier referring to the name of the mixture).  This situation is
% analogous to the discrete case where, even when no parents exist, the
% child still requires a one dimensional CPT of the appropriate
% size. Here, a single mixture of Gaussians of the
% appropriate dimension must be specified.
%
% Note that this situation applies only for the current switching
% parent configuration. In other words, even if nil is given as
% conditional parents for a switching configuration, the child might
% have other parents, namely the switching parents, and possibly other
% conditional parents.
%
% 2) The second form of ContinuousImplementation uses the syntax
%
%           "collection" "(" ListIndex ")" MappingSpec
%
% This case applies when multiple conditional parents are
% specified. Here, a decision tree is used to map from conditional
% parent values to the appropriate distribution, and this is done
% indirectly via the variable's collection.  The collection refers to one
% of the collection of objects (via the name given in the 'ListIndex')
% in GMTK's global arrays.  MappingSpec must refer to one of the
% decision trees which maps random variable parent indices to the
% relative offset in the collection array.

ContinuousImplementation = ContObsDistType
         (
             "(" ListIndex ")"
           |
              "collection" "(" ListIndex ")" MappingSpec
        )

% At the moment, the current version of GMTK only supports
% a mixGaussian distribution (the others are not yet supported),
% but the underlying mixture components can be a variety of different
```

```
% types of distribution.
ContObsDistType =
   "mixGaussian"
 | "gausSwitchMixGaussian"
 | "logitSwitchMixGaussian"
 | "mlpSwitchMixGaussian" ;


% A MappingSpec always refers to a decision tree.
% The ListIndex string is used to index into a table
% of decision trees to choose the decision tree
% that will map from the switching parents to one of the
% conditional parent lists.
MappingSpec = "mapping" "(" ListIndex ")"

% A list index is just the name (a string) of an internal GMTK object.
ListIndex = string

ChunkSpecifier = "chunk"  integer ":" integer
```

2.1.1 Variables

A random variable (RV) is the basic unit in GMTK. A variable may be discrete or continuous.

A discrete RV is always scalar valued in GMTK. A discrete RV has a cardinality c which specifies the number of valid values the RV may possess, ranging from 0 to $c-1$ inclusive.

A discrete RV may be hidden or observed. If the variable is observed, that means that its value is always known, and must be specified in some way, either immediately in the structure definition (see below in this section), or the variable must be associated with a slot in the global observation matrix (see Section 4.7.2).

A hidden discrete RV is most commonly integrated or marginalized (or sometimes even "maxed", as in a Viterbi-like computation) away as part of inference in GMTK. Therefore, the cardinality of a hidden random variable can have a significant effect on the computational complexity of the resulting inference. There are of course aspects other than just the cardinality which can affect complexity. For example, the average number of a variable's non-zero probability values (averaged over the different sets of possible parent variable values) can have a huge affect on complexity. The following is an example of a discrete hidden RV[3]

```
variable : state {
   type : discrete hidden cardinality 4000;
   ...
}
```

An observed RV, on the other hand, must always have its value specified somewhere, and there are two ways this can be done. In the first way, an range specification is used to specify an single element in the observation file (Section 4.7), for example:

[3]In GMTK, there are both structure files and parameter files. In this document, text material that belongs in a structure file will be placed in shaded boxes, and text material that belongs in parameter files will be placed in framed boxes.

```
variable : state {
   type : discrete observed 26:26 cardinality 4000;
   ...
}
```

In this example, feature number 26, which *must* be an integer in the observation file (i.e., not a floating point value), is used to supply the observed value for the variable `state` at each time frame. Furthermore, the value supplied in the observation file must be in the valid range of the RV (it must be non-negative, and must not be greater than $c - 1$ if c is the number of possible values the RV might have, something that we call the RV's *cardinality*), or a run-time error will occur.

The second way of specifying a value for a discrete observed random variable is to do so inline, or what is called an *immediate RV value*, as in the following example:

```
variable : state {
   type : discrete observed value 2000 cardinality 4000;
   ...
}
```

In this example, the variable `state` will always have observed value 2000 for any time frame in which it occurs. This is useful when using the observed variable as a child, and which can greatly enrich the sets of distributions and conditional independence statements that a Bayesian network (i.e., a type of directed graphical model) can represent (see Section xxx).

A continuous random variable in GMTK is a vector. At this time, GMTK supports only *observed* continuous random variables.

```
variable : obs {
   type : continuous observed 0:38;
   ...
}
```

The example defines a continuous observed random vector named `obs` of dimension 39. The given range determines which feature elements in the observation file correspond to the RV at each time frame. These observations must be continuous (not discrete).

For another example, to create three separate continuous observation vectors over regular features (such as MFCCs), their deltas (derivatives), and their double deltas (second derivatives), one could use the following:

```
variable : obs_feats  { type : continuous observed 0:12;  ... }
variable : obs_delts  { type : continuous observed 13:25; ... }
variable : obs_ddelts { type : continuous observed 26:38; ... }
```

This of course makes the assumption that the first 13 features in the observation file are the original features (i.e., standard MFCC cepstral coefficients C_0 through C_12), the next 13 are the deltas, and so on.

			Child			
			Discrete		Continuous	
			Hidden	Observed	Hidden	Observed
Parent	Discrete	Hidden	X	X		X
		Observed	X	X		X
	Continuous	Hidden				
		Observed				X

Table 2.1: GMTK can implement dependencies between two parent and child random variables depending on the type of the variables. This table shows which forms are currently implemented in GMTK. The squares marked with an X are currently supported.

2.1.2 RV Dependencies: Parents and Children

Dependencies[4] between parent and child random variables may be specified in several ways, depending on the types of the corresponding parent and child. There are a total of 16 possible combinations of parent and child type, as shown in Table 2.1. GMTK supports a subset of the possible dependencies in these 16 cases. In particular, if both parent and child variables are discrete, then any configuration of observed or hidden is supported. Furthermore, both hidden and observed discrete variables may be parents of observed continuous variables, and observed continuous variables may be parents of other observed continuous variables.

GMTK currently does not support hidden continuous parents or children, or observed continuous parents of discrete children. Note that some of these options can cause probabilistic inference to become intractable.

Discrete variables may be parents of other discrete variables regardless of their continuous/observed disposition. In the following example, the variable `parent` is a binary parent of the variable `child`, using a dense 2×2 CPT named `foo` to obtain the probabilities. In this case, both the parent and the child live in the same frame, since the frame offset `(0)` is used (see above).

```
variable : parent  {
     type : discrete hidden cardinality 2;
     ...
}
variable : child  {
     type : discrete hidden cardinality 2;
     ...
     conditionalparents: parent(0) using DenseCPT("foo");
}
```

In the example, either of the `hidden` keywords could change to `observed` in order to make one or more of the variables observed. In that case, of course, the observed value would need to be specified in some way.

Discrete parents of continuous observations may also be specified. In such a case, the value of the discrete parent determines which distribution (e.g., amongst say a set of forms of Gaussian

[4]More precisely, we should say "edges" since the word "dependency" would suggest that there indeed is a dependency between a parent and a child. In a Bayesian network, the existence of an edge implies that there might be but there is not necessarily a dependency. A lack of an edge implies that there is guaranteed to be some conditional independence. We use the term "dependency" rather loosely, however, since this has become common. Feel free in the following to substitute the word "edge" where "dependency" is used.

mixtures) should be used. In the following example, the variable `state` is a discrete hidden parent of the continuous observed `child` variable.

```
variable : state  {
     type : discrete hidden cardinality 2;
     ...
}
variable : child  {
     type : continuous observed 0:1;
     ...
     conditionalparents: parent(0) using
              collection("global")
              mixGaussian mapping("state2gaussianIndexfoo");
}
```

Continuous observed variables may also be parents of other continuous observed parents. The edge structure that should exist between continuous observed variables is not represented in the structure file, however, and instead is represented in the data file as `dlinks` and `dlink` matrices (see Section 4.5). While a design goal of GMTK was to keep everything relating to structure in the structure file, an exception was made in this case for several reasons. First, observation variables are vectors, and therefore the dependency structure in this case is between the *individual elements* of the vector observation variables. Second, the vectors themselves can be fairly long (e.g., typically 39 or more). Third, the dependency structure is often learned in a data-driven fashion. Fourth, such structure often *switches* (see below) which is to say that the structure might change depending on the current values of the discrete parents. For all of these reasons, including observation structure in the structure file would complicate the file beyond what seemed reasonable. Structure over observations is fully described in Section 4.5.

2.1.3 Switching Parents and Dependencies

In GMTK, there are two types of parent designations – a parent may be either a switching parent (a parent that determines the set of other parents a variable might have), a conditional parent (a parent whose parental status is conditional on the value of a switching parent), or both. Moreover, the way in which a conditional parent affects a child variable (e.g., the CPT) might switch depending on the current value of the switching parent.

Networks that use switching parents are also called Bayesian multi-nets [14, 10]. Normally, a variable has only one set of parents. GMTK, however, allows a variable's parents to change (or switch) conditioned on the current values of other (conditional) parents.

We will motivate this feature with an example. Suppose we are given four random variables, the binary variable S and variables C, P_0, and P_1, and we are interested in representing the distribution $p(c|p_0, p_1)$, but depending on the value of S, either P_0 or P_1 has no effect on C. In other words, we have that

$$\begin{aligned} p(c|p_0,p_1) &= \sum_s p(c,s|p_0,p_1) \\ &= p(c|p_0,p_1,S=0)p(S=0) + p(c|p_0,p_1,S=1)p(S=1) \\ &= p(c|p_0,S=0)p(S=0) + p(c|p_1,S=1)p(S=1) \end{aligned}$$

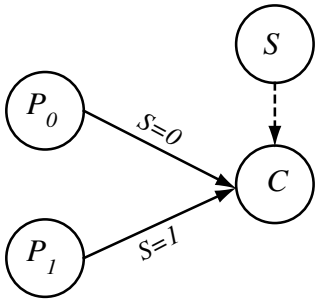

Figure 2.3: When $S = 0$, P_0 is C's only parent, and when $S = 1$, P_1 is C's only parent. S is called a switching parent, and P_0 and P_1 conditional parents.

This representation can significantly reduce the number of parameters required to represent a probability distribution. For example, $p(C|P_1, S = 1)$ needs only a two-dimensional table whereas $p(C|P_0, P_1)$ requires a three-dimensional table. The equation above corresponds to the conditional independence statements $C \perp\!\!\!\perp P_1 | \{P_0, S = 0\}$ and $C \perp\!\!\!\perp P_0 | \{P_1, S = 1\}$. In other words, conditional independence can occur for certain values of random variables, rather rather than for all values (as is more typical). This scenario is depicted graphically in Figure 2.3.

GMTK supports switching parents by having two types of parent designators. The following example shows how to implement the switching parent functionality described in Figure 2.3.

```
 variable : S   { type : discrete hidden cardinality 2; ... }
 variable : P0  { type : discrete hidden cardinality 2; ... }
 variable : P1  { type : discrete hidden cardinality 2; ... }
 variable : C   {
      type : discrete hidden cardinality 2;
      switchingparents: S(0) using mapping("s1toindex");
      conditionalparents:
            P0(0) using DenseCPT("p0CPT")
          | P1(0) using DenseCPT("p1CPT");
}
```

In this example, all variables are binary (cardinality 2), and the decision tree[5] named `s1toindex` is used to map from values of `S` (which in this case is either the value 0 or the value 1) to either the integer 0 (which causes parent P_0 to be used) or to the integer 1 (which causes parent P_1 to be used). When P_0 is used, the two-dimensional CPT `p0CPT` will provide the conditional probabilities of C, and correspondingly for P_1.

A slightly more complicated example follows.

```
 variable : S1  { type : discrete hidden cardinality 100; ... }
 variable : S2  { type : discrete hidden cardinality 100; ... }
 variable : P1  { type : discrete hidden cardinality 100; ... }
 variable : P2  { type : discrete hidden cardinality 100; ... }
 variable : P3  { type : discrete hidden cardinality 100; ... }
 variable : C  {
```

[5] Decision trees are described in Section 4.2.4.

```
      type : discrete hidden cardinality 2;
      switchingparents: S1(0), S2(0) using mapping("s1s2toindex");
      conditionalparents:
            P1(0), P2(0) using DenseCPT("p1p2CPT")
          | P1(0), P3(0) using DenseCPT("p1p2CPT")
          | P2(0), P3(0) using DenseCPT("p1p2CPT");
}
```

In this example, variable `C` is the child variable with two switching parents `S1` and `S2` and conditional parents `P1`, `P2`, and `P3`. This corresponds to the equation:

$$\begin{aligned} P(C|P_1,P_2,P_3) = & P(C|P_1,P_2,\{S_1,S_2\}\in\mathcal{R}_0)P(\{S_1,S_2\}\in\mathcal{R}_0) \\ & + P(C|P_1,P_3,\{S_1,S_2\}\in\mathcal{R}_1)P(\{S_1,S_2\}\in\mathcal{R}_1) \\ & + P(C|P_2,P_3,\{S_1,S_2\}\in\mathcal{R}_2)P(\{S_1,S_2\}\in\mathcal{R}_2) \end{aligned}$$

where $\mathcal{R}_1$, $\mathcal{R}_2$, and $\mathcal{R}_3$ are disjoint regions in the joint state space of S_1, S_2. The regions are determined by the decision tree `s1s2toindex`. The decision tree functions to map any pair of values of S_1, S_2 that lives within $\mathcal{R}_0$ to the integer 0, thereby selecting the first set of conditional parents (P_1 and P_2 in this case). Similarly, the decision tree will map any pair of values of S_1, S_2 that live within $\mathcal{R}_1$ to the integer 1 (and likewise for $\mathcal{R}_2$ and integer 2). This way, even though the joint state space of the set of switching parent variables might be large (10,000 in this case), only a small list of conditional parents need be specified.

Note also that GMTK's switching parent functionality can also be used to switch implementations, or CPTs (conditional probability tables, described in Section 4.2). For example, in the following:

```
 variable : S   { type : discrete hidden cardinality 2; ... }
 variable : P  { type : discrete hidden cardinality 2; ... }
 variable : C   {
      type : discrete hidden cardinality 2;
      switchingparents: S(0) using mapping("s_to_index");
      conditionalparents:
            P(0) using DenseCPT("pCPT_1")
          | P(0) using DeterministicCPT("pCPT_2");
}
```

The switching parent `S` is used not to switch between different parents, but instead to switch between two different CPTs.

Switching parents have an analogous effect when used as parents of continuous random variables. Consider the following example:

```
 variable : S   { type : discrete hidden cardinality 2; ... }
 variable : P0  { type : discrete hidden cardinality 2; ... }
 variable : P1  { type : discrete hidden cardinality 2; ... }
 variable : C   {
      type : continuous observed 0:1;
      switchingparents: S(0) using mapping("s1toindex");
```

```
        conditionalparents:
              P0(0) using collection("p0_collection")
                 mixGaussian mapping("p0_to_gm_index")
            | P1(0) using collection("p1_collection")
                 mixGaussian mapping("p1_to_gm_index");
  }
```

In this case, the switching parent S determines which parent will determine the Gaussian mixture at the current time. Each parent uses a different collection and different decision tree mapping function (collections are described in Section 4.1).

In GMTK, switching parent functionality may also exist for dependencies that exist between continuous observations. In fact, explicit switching parents as described above are not used to specify switching structure over observations. Consider the following example:

```
   variable : P  { type : discrete hidden cardinality 2; ... }
   variable : C   {
        type : continuous observed 0:1;
        switchingparents: nil;
        conditionalparents:
              P(0) using collection("global")
                 mixGaussian mapping("p_to_gm_index");
  }
```

Even though parent P is not officially a switching parent, it could in effect be a switching parent, depending on the type of Gaussian mixture that is used. As will be seen in Section 4.4, each Gaussian mixture can have a mixture of different types of mixture components, and each component is a general conditional Gaussian. This means that the component could be a diagonal- or full-covariance Gaussian, a sparse-covariance Gaussian, or a Gaussian with conditional dependencies on parents that come from either the past or the future of the current time frame. Furthermore, each Gaussian component can have a separate set of such dependencies. Therefore, by having P select amongst sets of mixtures having different components, and by having each mixture use components with different dependencies, P becomes a switching parent. More on this topic will be described in Section 4.5.

2.1.4 GMTKL Templates and Unrolling

As mentioned above, a GMTKL structure file defines a template which then is "unrolled" or rubber stamped for each utterance enough times in order to be of length T frames. Consider the following structure file fragment which shows only parent-child relationships for clarity:

```
frame: 0 {
   variable : S { ...;  conditionalparents : nil ...;  }
   variable : O { ...;  conditionalparents : S(0) ...; }
}
frame: 1 {
   variable : S { ...; conditionalparents : S(-1) ...; }
   variable : O { ...;  conditionalparents : S(0) ...; }
```

```
}
frame: 2 {
   variable : S { ...; conditionalparents : S(-1) ...; }
   variable : O { ...;  conditionalparents : S(0) ...; }
   variable : E { ...;  conditionalparents : S(0) ...; }
}
chunk 1:1;
```

In this template, the three frames are not identical to each other. In first frame, S has no parents. In the third frame, there is an extra variable E which has S in the current frame as a parent. The chunk specifies only frame 1, so it is only frame 1 which is unrolled $T-2$ times to make a network of length T frames. This is depicted in Figure 2.4.

GMTK allows unrolling of more general structures than the one described above. A GMTK template consists of a prologue, a chunk to be repeated, and an epilogue. Each of the prologue, chunk, and epilogue can consist of any number of frames, and need not have the same set of random variables. While the prologue and the epilogue can have zero frames, the chunk must have at least one frame.

Let's say that T_p, and respectively T_c and T_e, are the number of frames in the prologue, chunk, and epilogue, meaning that the template consists of a total of $T_p + T_c + T_e$ frames. After GMTK unrolls the chunk to obtain a T frame network, the network will consist of $T = T_p + kT_c + T_e$ for some positive integer k (see Figure 2.5). In the current version of GMTK, if no such k can be found which satisfies this equation, a run-time error will occur.[6]

Note that when the chunk is unrolled, the parameters for each of the variables in the unrolled network are shared with the corresponding variables in the rolled template. In this way, unrolling allows for an easy way to tie parameters of different random variables together, based on only specifying the parameters in the original template (See Figure 2.4). Other forms of GMTK parameter sharing are described in Section 4.6.[7]

One must be careful not to specify a template that, when unrolled, results in an invalid network. Consider the following template fragment for example.

```
frame: 0 {
 variable : R { type : discrete hidden cardinality 2; ... }
 ...
}
frame: 1 {
 variable : S { type : discrete hidden cardinality 4; ... }
 conditionalparents : R(-1) using ... ;
 ...
}
frame: 2 {
 variable : S { type : discrete hidden cardinality 4; ... }
 conditionalparents : S(-1),R(-2) using ... ;
```

[6] At the current time, T must be of the proper multiple, but I've had requests to allow the automatic skipping of say the last or first number of frames of each utterance so that we don't always have to have this restriction. This will be included in a future version of GMTK.

[7] Note, that if no parameter sharing is desired across an unrolled network, then an extra frame counter variable can be specified which switches the implementation of all the variables in the current frame.

```
 ...
}
frame: 3 {
 variable : S { type : discrete hidden cardinality 4; ... }
 conditionalparents : S(-1) using ... ;
 ...
}
chunk 2:2
```

There are several problems with this template. Consider the graph in Figure 2.6 which shows the template after unrolling one time, and the variables are renamed with primes. From the semantics of unrolling, the variables S_3' and S_2' share exactly the same set of parameters since they both come from S_2 in the template. S_2, however, requires a parent named R_0 which has cardinality 2, and a parent named S_1 which has cardinality 4. The same set of parent names and cardinalities need to exist in the same frames relative to the frames containing S_2' and S_3', or for any variable which is derived from S_2 after unrolling. The problem with the above network that S_3' is not able to find a parent named R in frame 1.

Even if there was a variable called R in template frame 1, if that R in frame 1 had cardinality 4 there would be a problem since S_3' would ask for a parent variable named R that has cardinality 2 rather than 4. Therefore, any such R variable in template frame 1 should also have cardinality 2 for this template to be valid.

In general, unrolling is valid only if all parent variables in the unrolled network are *compatible* with those parents that exist in the template. A variable is said to be compatible if has 1) the same name, 2) the same type (i.e., discrete or continuous), and 3) the same cardinality (number of possible values). For example, suppose that A is a random variable in the template having B as a parent that is k frames to the left of A in the template. After unrolling, each variable A' in the unrolled network that is derived from A in the template must have a parent having the same name as B, it must be k frames to the left of each A', and must have the same type and (if discrete) cardinality as B. If these conditions are not met, the template is invalid and GMTK will report an error.

As can be seen, one must take care to specify templates that correspond to valid unrolled networks. GMTK will detect when an unrolled template is invalid, and will report an error.

Templates and Multi-Rate/Multi-Scale Processing

Since a template chunk may consist of several frames, and each frame may contain any number of different variables, it is possible to specify multi-rate or multi-time-scale networks using GMTK. This may be done by specifying a multi-frame chunk with a different set of variables (either hidden or observed, or both) in each frame. Consider the template fragment given in Figure 2.8 and shown graphically in Figure 2.7.

In this template, the chain R_t exists every frame, and is used to determine the distribution over observation features 0 through 4. The chain Q_t proceeds at 2/3 the rate of R_t, and it is used to determine the distribution over observation features 5 through 9, but these features only have a distribution for 2 out of every three frames. Observations must exist for each frame in the global observation matrix (see Section 4.7), so in this case, features 5 through 9 are ignored every third frame. Also, for every two steps of Q_t (equivalently every three steps of R_t), Q_t possesses a dependency on R_t. Using this feature, therefore, one can easily specify general multi-rate GM

networks where variables occur at rates which are fractionally but otherwise arbitrarily related to each other.

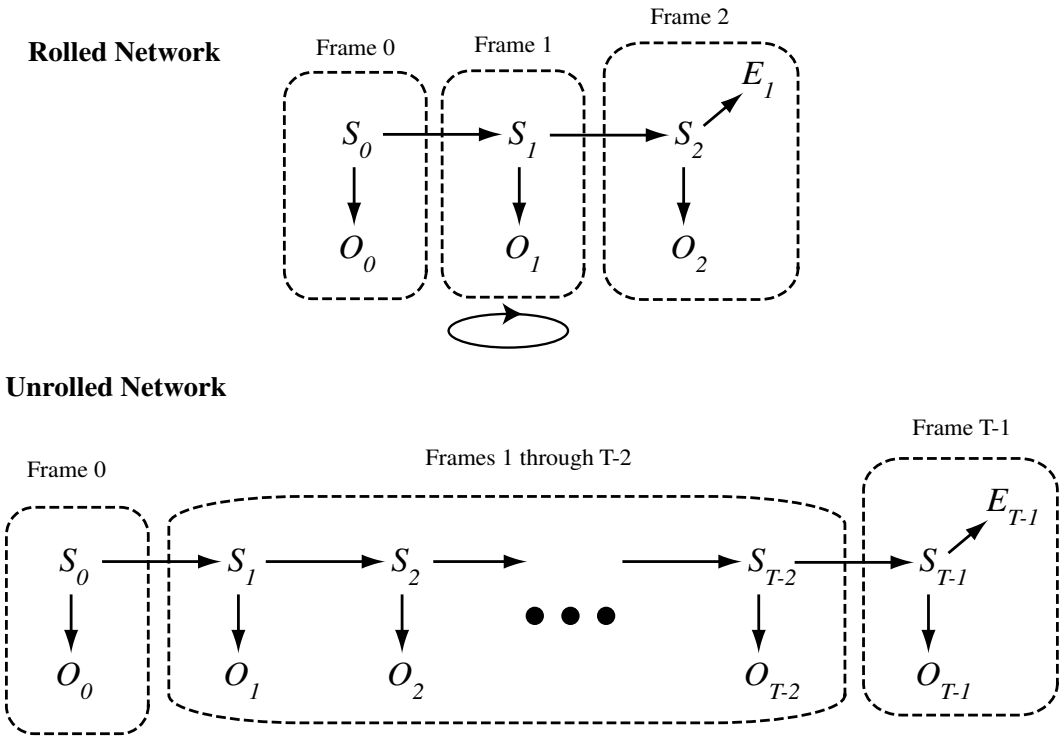

Figure 2.4: Rolled and Unrolled network. On the top, the template corresponds to a rolled up network with three frames and 7 random variables. On the bottom, frame 1 has been unrolled $T-2$ times producing a network with $2T+1$ random variables. Note that both the first and the last frame of the template are different from the middle frame. This is reflected in the unrolled network, as the middle $T-2$ frames are identical in structure, but they are distinct random variables. In $T-2$ middle frames of unrolled network, however, all variables share the same sets of parameters that their respective counterparts specified in the template. Note that the parameters of S_1 through S_{T-2} are essentially *shared*, and are the same as that which is specified in the template for variable S_1. Sharing in this case means that the probabilities of the same value for different random variables are forced to be identical (so that $p(S_1 = s) = p(S_2 = s) = \ldots = p(S_{T-2} = s)$). Similarly, the parameters for O_1 through O_{T-2} are the same, and are those specified for template variable O_1. Parameter sharing amongst variables at different times is utilized both during parameter training (such as with the EM algorithm) and just during probability evaluation. It is also possible with GMTK to share parameters between variables that do not correspond to the same template variable (see Section 4.6)

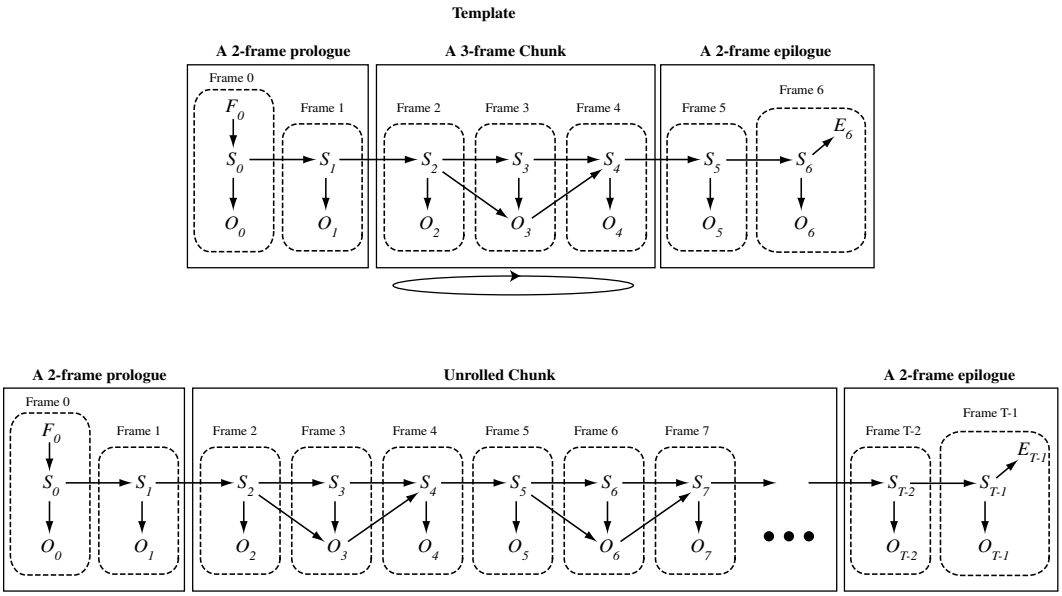

Figure 2.5: A multi-frame template (top) with a two-frame prologue, a 3-frame chunk, and a 2-frame epilogue when unrolled produces the network at the bottom.

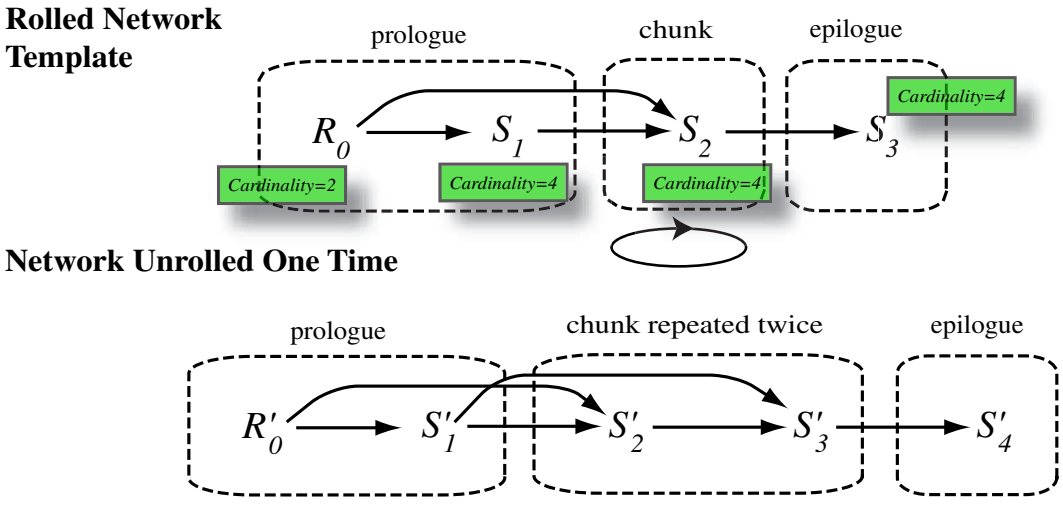

Figure 2.6: An example of an invalid template because of name and cardinality incompatibilities. In particular, in the unrolled network, the variable S'_3 (originally S_2 in the template) has parents S'_1 and S'_2. S'_1 has the wrong cardinality (4 rather than 2) and the wrong name (S rather than R).

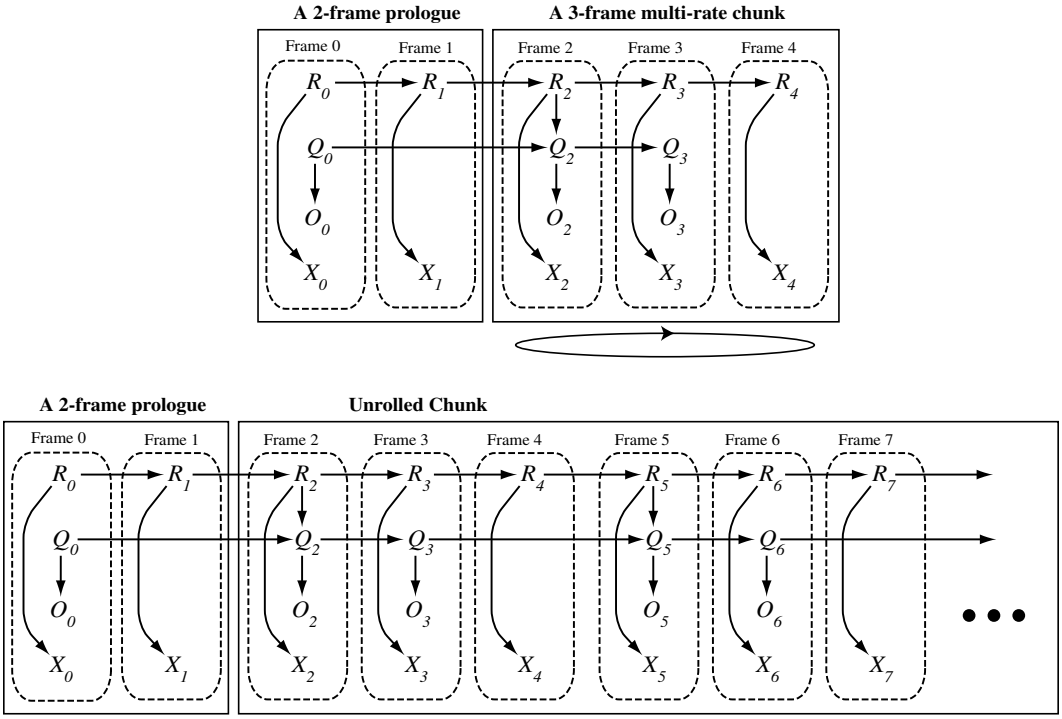

Figure 2.7: An example of a network template that specifies a multi-rate model, both over hidden and over observed variables. Top: the multi-rate template. Bottom: the unrolled multi-rate network. The structure file for this graph is shown in Figure 2.8.

```
frame: 0 { variable : R {... }  variable : Q {... } ...
           variable : O { type: continuous observed 5:9;
                          conditionalparents : Q(0) using ... ;  }
           variable : X { type: continuous observed 0:4;
                          conditionalparents : R(0) using ... ;  }}
frame: 1 { variable : R {... }  variable : Q {... } ...
           variable : X { type: continuous observed 0:4;
                          conditionalparents : R(0) using ... ;  }}
frame: 2 {
 variable : R { type : discrete hidden ... ;

                conditionalparents : R(-1) using ... ; }
 variable : Q { type : discrete hidden ... ;
                conditionalparents : Q(-2),R(0) using ... ; }
 variable : O { type: continuous observed 5:9;
                conditionalparents : Q(0) using ... ;  }
 variable : X { type: continuous observed 0:4;
                conditionalparents : R(0) using ... ;  }
}
frame: 3 {
 variable : R { type : discrete hidden ... ;
                conditionalparents : R(-1) using ... ; }

 variable : Q { type : discrete hidden ... ;
                conditionalparents : Q(-1) using ... ; }
 variable : O { type: continuous observed 5:9;
                conditionalparents : Q(0) using ... ;  }
 variable : X { type: continuous observed 0:4;
                conditionalparents : R(0) using ... ;  }
}
frame: 4 {
 variable : R { type : discrete hidden ... ;
                conditionalparents : R(-1) using ... ; }
 variable : X { type: continuous observed 0:4;

                conditionalparents : R(0) using ... ;  }
}
...
chunk 2:4
```

Figure 2.8: A structure file for a template specifying a multi-rate model. The graph for this structure file is shown in Figure 2.7.

Chapter 3

Representing Parameters in GMTK

The structure, the implementation, and the parameters of a graphical model are separate aspects of the model, a feature that GMTK exploits. For example, the structure consists of a graph's set of nodes and edges, the implementation consists of what the various dependencies or edges mean in terms of how parent and child variables are related to each other (linear, nonlinear, sparse CPT, deterministic CPT, etc.), and the parameters complete the representation by specifying values used by the implementations of the dependencies.

In GMTK, all model parameters for a graph are represented in a set of files separate from the structure file. Furthermore, the graph structure over continuous vector observations are also represented in a separate file than the hidden structure, as mentioned in Section 2.1.2.

In this section, we describe GMTK's general parameter format and observation structure. GMTK parameters include probability values for CPTs, decision trees (used to represent deterministic mappings), means and variances of Gaussians, mixture coefficients (i.e., component responsibilities), all in a flexible way that allows for a rich set of parameter tying/sharing.

3.1 Numerical vs. Non-Numerical Parameters

GMTK makes a distinction between those parameters that are numeric in nature and are typically trained by, say, the EM algorithm, and those parameters which are entirely structural or relational and which are not trained using standard training procedures.

Numerical parameters include things such as Gaussian means, variances, component responsibilities, dense and sparse CPTs, and so on. Non-numerical parameters include things like decision trees, and, as will be seen, dependency link (or *dlink*) structures.

3.2 The GMTK basic parameter "object"

All parameters in GMTK are contained in basic GMTK parameter *objects*. Objects have names (any text string) which are used to specify and identify the object for various purposes. For example, some objects (e.g., Gaussian components) might require other objects (e.g., mean vectors) for their definition. These textual names are used by objects to refer to other needed objects. The names are also used by error messages to identify which object is having trouble. Within a given type of object (say covariances), names must be unique. This means that two covariance matrices may not have the same name. Different types of objects, however, may have the same name (so you may have both a mean and a covariance named `foo`).

Objects are defined in GMTK parameter files. These files use a syntax that reduces the chance of errors that might occur when reading in a parameter file. For this reason, you will notice a certain amount of redundancy in the parameter files. The files are also designed so that a parser can quickly and efficiently read and interpret the files. The redundancy helps to both reduce errors and to speed reading.

There are a number of different types of objects (defined below). Objects are normally specified as a list of N objects of the same type. A list consists of 1) the number of objects that are about to follow, and 2) a list of that many objects, each preceded by the index number (zero-offset) of that object. The definition of each object includes a specification of its name, and any other required parameters. The definitions of each object are given in the following sections. An example of a generic list of objects follows:

Parameter File

```
% this is a comment
3 % three objects are to follow
0 % object 1 (zero offset) will follow
<definition of object 1>
1 % object 2 will follow
<definition of object 2>
2 % object 3 will follow
<definition of object 3>
```

The list consists of three objects. The contents of the definition of the objects depends on the type of object being defined (e.g., a decision tree, conditional probability table, and so on). As mentioned above, redundancy exists in the list (a specification of the number of objects that will follow in addition to the object number itself, which could be inferred while reading). If any inconsistencies are found when the parameter files are read in, GMTK will report an error. The basic data objects supported in GMTK are given in Table 3.1.

3.3 ASCII/Binary files, Preprocessing, and Include Files

Many parameter files can be either in plain ASCII or in binary. Clearly, binary files are preferred (and are often necessary) when speed and size become an issue. ASCII files can be useful when one is wishing to quickly view the parameter files, or when building scripts that generate initial sets of parameter files.

All ASCII files (both structure *and* parameter files) are processed using the C preprocessor, `cpp`. This means that all features of the C pre-processor that are available in C are also available in GMTK parameter files. This includes things like defining variables, creating macros, conditional selection of text, including files, and so on. For example, you can include files as

```
#include "foo"
```

or define a macro as in

```
#define PATH /foo/bar/baz
```

Object	Master File Keyword	Description
Dense CPT	DENSE_CPT	A Dense CPT (conditional probability table)
Sparse CPT	SPARSE_CPT	A Sparse CPT, i.e., a CPT where many of the CPT values are zero, and are therefore not represented in the table.
Deterministic CPT	DETERMINISTIC_CPT	A deterministic CPT, i.e., a CPT where for each value of the parents, only one value of the child variable has non-zero probability.
Decision Tree	DT	A decision tree, used for a number of things including sparse CPTs, deterministic CPTs, collection of discrete parent variable values to selection of a Gaussian mixture, and set of switching parent values to a choice of conditional parents.
Dense Probability Mass Function	DPMF	A DPMF is a dense one-dimensional mass function which is used for mixture weights (responsibilities) in Gaussian mixtures. A DPMF is also used as the probabilities for Sparse CPTs
Sparse Probability Mass Function	SPMF	A SPMF is a sparse one-dimensional mass function, used for a Sparse CPT. A SPMF is just like a DPMF except zero probability elements are not represented.
Gaussian Mean	MEAN	A mean vector of a multi-variate Gaussian.
Gaussian Covariance	COVAR	A positive vector for the diagonal covariances of a multi-variate Gaussian.
dlink (dependency link) matrix	DLINK_MAT	A matrix of values giving the regression coefficients of a Gaussian. This corresponds to off-diagonal elements of a covariance matrix when non-diagonal covariance matrices are used.
dlink structure	DLINK	A table giving just the (potentially sparse) structure of dlink matrix values.
Gaussian component	GC	An object which groups together a mean, covariance, and (possibly) a dlink matrix to form a Gaussian component.
Mixture of Gaussians	MG	An object which groups together multiple Gaussian components and a DPMF to form a Gaussian mixture object. Note that these objects have the ability during training to split/vanish components.
Name collection	NAME_COLLECTION	An object which consists of a set of object names used for indirect reference in sparse CPTs, and in the mapping going from random variable values via decision trees to selections of Gaussian mixtures.

Table 3.1: Summary of GMTK basic objects.

Documentation on `cpp` is certainly outside the scope of this article, but we will in the following provide a few tips/hints on some of the available features when using `cpp` with GMTK.

First, note that for GMTK to run, `cpp` must be in a user's path. Normally, this is fine since the C compiler is typically in the user's path, and `cpp` is in the same location as the C compiler. It seems, however, that on some machines and for some users, `cpp` is not in the standard location or the standard or default path.

Note also that there is a number of ways to interface with `cpp` on the command line of GMTK programs. For example, you can define macros or variables on a GMTK program command line which can often be more convenient in scripts than having to write out multiple files, one per each training or testing chunk. See Section 5 on GMTK program command line arguments for a further description of this feature.

Note that user's environment variables are not used in GMTK parameter files. But supposing you would like to specify the path of an included DT with a script, you can use `cpp` variables, possibly by defining the value of the variable on the command line of one of the GMTK programs.

There are certain issues to beware of, however, when using `cpp` when *concatenating strings*. For example, supposing you would like to do the following:

Parameter File

```
#define PATH /foo/bar/baz/
#define FILE1 spoons1
#define FILE2 spoons2
```

and then wish to concatenate the `PATH` variable with both the `FILE1` and `FILE2` variables to create two file names with a complete path specification. Depending on the version of `cpp` you use, you might be able to get away with using the `cpp` concatenation operator ##, and do something like:[1]

```
...
DT_IN_FILE PATH ## FILE1 ascii
...
DT_IN_FILE PATH ## FILE2 ascii
```

For some versions of `cpp`, this won't work. Either there will be a space inserted between the path and the filename, or something else might be wrong. In that case, you can define a concatenation macro as follows:

```
#define CAT_B(a,b) a ## b
#define CONCAT(a,b) CAT_B(a,b)
```

For esoteric reasons that we will not describe here, you need to define the concatenation macro in this way. Once this is done, you can then use:

```
...
DT_IN_FILE CONCAT(PATH,FILE1) ascii
...
DT_IN_FILE CONCAT(PATH,FILE2) ascii
```

which will expand out to

[1]The example, by the way, states that there are two ASCII files named `spoons1` and `spoons2` both in directory `/foo/bar/baz` and both of which contain decision trees.

```
...
DT_IN_FILE /foo/bar/baz/spoons1 ascii
...
DT_IN_FILE /foo/bar/baz/spoons2 ascii
```

as you would expect.

There are a number of ways of using `cpp` to make your scripts easier to write. One thing to keep in mind, however, is that when ASCII files get large, sometimes it can take `cpp` quite a long time (30 minutes or longer!!) to parse and read in these files. In such cases, it is best to convert the files to binary. This will be faster not only because there is no ASCII to binary conversion required when reading in the file but also because `cpp` is not used to parse the file in this case. Converting from ASCII floating point numbers to an internal binary IEEE floating point representation itself can itself take quite a long time. There is an easy ascii/binary conversion program available for this purpose (see Section 5).

3.4 Input/Output Parameter Files

Apart from the graphical structure specification file, there are several ways to get parameter input into and output out of GMTK. All of the GMTK programs (the training, decoding, etc.) support this file procedure, so we will describe it in general terms. The general format described below applies to all the GMTK objects described in Section 3.2.

Each GMTK program supports an `inputMasterFile`, an `outputMasterfile`, an `inputTrainableParameters` file and an `outputTrainableParameters` file. These are specified in GMTK by command line options going by the same name. For each of the objects given in Section 3.2, GMTK keeps an internal array. These data files give the user several ways to manipulate these arrays of objects. The fundamental way this occurs is either to read in and append new objects to these arrays, or write the entire arrays out to disk (in either ASCII or binary format).

The master files have the capability of reading and/or writing all of the possible types of input parameters and/or objects (other than the graph structure). The master files are always in ASCII, and are always pre-processed by `cpp`. The master files do specify other files which might or might not be ASCII.

3.4.1 Input Master File

The input master file can be used to specify all the input parameters to a GMTK program. An input master file consists of a list of input specifiers, each consisting of 1) an *object keyword*, 2) an object *locater* specification, and then 3) a *list* of objects in the appropriate location.

An object keyword specifies one of the object types listed in Section 3.2. These keywords are listed in Table 3.1, but where each keyword is appended with the string `_IN_FILE`.

An object locater specification is either 1) the word `inline` or 2) a file name followed by either the string `ASCII` or `BINARY`. If `inline` is given, then the list of objects is given immediately following the `inline` keyword, in ASCII, within the master file itself. If, instead, a file name is provided, then the list of objects is taken from the file with that name which is presumed to be in either ASCII format or binary format depending on what is specified.

For example, here is a portion of a master file that consists of an inline list of dense conditional probability tables (the objects themselves are described in Section 4.2).

```
1  DENSE_CPT_IN_FILE inline
2  2 % there are two Dense CPTs
3  0 % first one
4  wordUnigram  % name of first one
5  0 % num parents
6  5 % num values
7  0.2 0.2 0.2 0.2 0.2
8
9  1 % 2nd CPT
10 wordBigram % name
11 1 % one parent
12 3 3  % parent cardinality = 3, self card = 3
13 0.3 0.3 0.4
14 0.4 0.3 0.4
15 0.4 0.3 0.3
```

This portion of the master file declares two Dense CPTs, with names `wordUnigram` on line 3 and `wordBigram` on line 9 respectively.

The next example obtains the list of objects from an ASCII file:

```
DT_IN_FILE decision_tree.dts ascii
```

This section of the master file obtains a collection of decision trees in ASCII form from the file `decision_tree.dts`. If the file was in binary, the `ascii` keyword would instead be `binary`.

Note that if multiple keywords occur in a master file, the internal GMTK arrays are appended with the new objects encountered. For example, given the following:

```
DT_IN_FILE decision_tree1.dts ascii
DT_IN_FILE decision_tree2.dts ascii
```

First, the list of decision trees in the first file are read, followed by the second list.

The same data file may be specified multiple times within a master file, and such data files may contain a heterogeneous mix of object types. Suppose, for example, that you have a file named `file.data` containing a list of Dense CPTs followed by a list of decision trees. This can be read in as follows:

```
DENSE_CPT_IN_FILE file.data ascii
DT_IN_FILE file.data ascii
```

After encountering the dense CPTs in the file, GMTK will continue reading where it left off to read in the decision trees. While a file may contain a heterogeneous mix of different object types, any given file is either entirely in ASCII or entirely in binary (i.e., you can not mix ASCII and binary format data within a single file). Note also that multiple files can be open at the same time, so you can, for example, first read Gaussian means from one file, then read Gaussian covariance matrices from another, and then go back to the first file and continue reading decision trees.

3.4.2 Output Master File

The output master file is the analog of the input master file, but where parameters are specified to be written. Parameters are written, for example, whenever a change has occurred such as after an iteration of EM training, or during a conversion from ascii to binary format.

An output master file consists of a list of output specifiers, each consisting of 1) an *object keyword*, 2) a file name specifying where to write, and 3) an ascii/binary keyword.

An object keyword specifies one of the object types listed in Section 3.2. These keywords are listed in Table 3.1, but where each keyword is appended with the string `_OUT_FILE`. The file name specified is any file to which you have write permissions. Furthermore, you may write a file in either ASCII or binary format. Note that all of the GMTK internal objects of a given type are written — there is no way at this point to specify that certain Gaussian means should be written to one file, and certain other Gaussian means should be written to another.

Like with input master files, output file names may occur multiple times, where the new objects are appended at the end of the file each time the file name is encountered (except for the first time the file name is given when the file is truncated to zero length, see below). For example,

```
DENSE_CPT_OUT_FILE foo.out ascii
DPMF_OUT_FILE foo.out ascii
MEAN_OUT_FILE bar.out ascii
```

This output master file writes all Dense CPT objects to the file `foo.out`, and then appends all DPMF objects to the same file. It then writes all the Gaussian means to the file `bar.out`.

IMPORTANT: The first time an output file name is encountered in an output master file, the file with that name is truncated if it exists. In the example above, both `foo.out` and `bar.out` are truncated prior to writing. Therefore, it is important to ensure that these files, if they exist, do not contain needed data.

There exists a special character sequence in file names for the GMTK EM training program. If any of the file names have the string "`@D`" in them, then that string is replaced with the current EM iteration number. For example, consider the following:

```
DENSE_CPT_OUT_FILE foo@D.out ascii
DPMF_OUT_FILE foo@D.out ascii
MEAN_OUT_FILE bar@D.out ascii
```

If the EM iteration is 3, then the files that will be written are called `foo3.out`, and `bar3.out`. This makes it simple to keep track of the parameters of multiple iterations of EM.

3.4.3 Special Input/Output Trainable File

GMTK programs allow a simple way of reading and writing the trainable parameters in a fixed format using a single step. Essentially, trainable files provide a short-cut to read/write trainable files w/o needing to specify a special master file for this purpose. The trainable parameters are any of the parameters that could potentially be modified during training. This, therefore, does not include decision trees which are fixed once they are read in.

Specifying the `inputTrainableParameters` option on the command line of one of the GMTK programs (see Section 5), and, say, with file name `input.in` to be read in ASCII format, is equivalent to a master input file of the following format:

```
DPMF_IN_FILE input.in ascii
SPMF_IN_FILE input.in ascii
MEAN_IN_FILE input.in ascii
COVAR_IN_FILE input.in ascii
DLINK_MAT_IN_FILE input.in ascii
WEIGHT_MAT_IN_FILE input.in ascii
DENSE_CPT_OUT_FILE input.in ascii
GC_IN_FILE input.in ascii
MG_IN_FILE input.in ascii
GSMG_IN_FILE input.in ascii
LSMG_IN_FILE input.in ascii
MSMG_IN_FILE input.in ascii
```

Note that the last three entries (GSMG, LSMG, MSMG) are reserved, and will be used in future versions of GMTK. For now, those three groups are written with zero objects in each case. Note also that if the binary option is specified on the command line, then the behavior would be the same except all ASCII keywords would change to binary. Input master and input trainable files may be used together, where the input trainable parameters are appended to the internal GMTK arrays for the corresponding object type. In fact, master and trainable files must be used together if decision trees are to be used, as these objects are non-trainable.

Specifying the `outputTrainableParameters` option on the command line is analogous to the input case. In this case, however, all internal objects of the corresponding type are written to disk. This is equivalent to an output master file with the following format:

```
DPMF_OUT_FILE output.out ascii
SPMF_OUT_FILE output.out ascii
MEAN_OUT_FILE output.out ascii
COVAR_OUT_FILE output.out ascii
DLINK_MAT_OUT_FILE output.out ascii
WEIGHT_MAT_OUT_FILE output.out ascii
DENSE_CPT_OUT_FILE output.out ascii
GC_OUT_FILE output.out ascii
MG_OUT_FILE output.out ascii
GSMG_OUT_FILE output.out ascii
LSMG_OUT_FILE output.out ascii
MSMG_OUT_FILE output.out ascii
```

Again, changing the ASCII keyword to binary will change the format written.

Chapter 4

GMTK Representation Objects

All data structures and parameters in GMTK are specified using GMTK objects. This includes everything from objects that represent a collection of names of other objects to objects that contain a collection of real numbers which are the mean of a Gaussian. In this chapter, we will describe in detail the various GMTK objects, how they are used, their syntax, what they do, and how they are used together and depend on each other. In this discussion, we will also overview the syntax of much of GMTK's parameter file format.

4.1 Collection Objects

Our first basic GMTK object is the named collection object. Simply put, a named collection object is a mapping from integers to strings. Specifically, such an object consists of an ordered list of names of other objects, and is used as a mapping from an integer corresponding to the position in the collection, to the object that has the name given in that position. Collection objects make it much easier to specify indexing by name, rather than having to specify indexing by an integer index.

Name collection objects are used for several purposes.

1. For continuous random variables, they are used as a mapping from the decision tree (DT) integers at the leafs of the DT to a particular Gaussian mixture object. As mentioned in Section 4.2.4, decision trees are used (among other things) to map from a set of discrete parent random variables to a leaf-node integer value. This integer value specifies the index of the Gaussian mixture used for the continuous random variable, under the current set of parent values.

   A name collection object is further used to map from that leaf-node integer value to the specific textual name of the Gaussian mixture to use. This makes it much easier to specify decision trees since the absolute index number of the decision tree leaf-node is not needed to determine the specific mixture. Note, therefore, that two decision trees can be shared (see Sections 4.6 and 4.2.3) by two continuous random variable objects that use two different name collection objects. It is the name collection objects that can specify a different set of mixtures, even for the same decision tree.

   Note, that there is a special internal name collection object named "`global`". When this name is mentioned in a structure file, the decision tree leaf node index maps directly into the global internal array of Gaussian mixtures, rather than indirectly via a collection. This ability makes it easier to rapidly set up a simple system.

2. For sparse conditional probability tables, name collection objects are used to map from the decision tree leaf values to the name of the spare probability mass function (SPMF) that is to be used for the corresponding leaf value.

In both cases, the use of a name collection makes it much easier to specify other GMTK objects, as otherwise the decision trees would need to keep track of the absolute position in GMTK's object tables of the desired object. If there was a need for different groups of objects (say the first 100 Gaussian mixture was 13 dimensional, and the next 100 was 26 dimensional), not having the name collection ability would mean that the decision tree for the second group would always need to use an offset of 100. The use of name collection means that second group may be specified indirectly.

Note further that the use of a name collection does not require an additional indirection internal to the code. Each of the indirections specified by a name collection are essentially "compiled out" when GMTK loads the files, rather than calculated each time the indirection is used.

Name collection objects are specified quite simply. The following is an example of how this is done.

```
NAME_COLLECTION_IN_FILE inline
1 % one name collection to follow.
0 % index 0, first collection.

myCol % The name of the collection, 'myCol'
5     % The collection length, only 5 entries.
      % Finally, the list of five names.
      % Note that new line characters are ignored.
object1 object2
object3 object4
object5
```

Note that the white space between the set of object names is ignored.

Here is a more complete example using named collections. The example also uses decision trees (described later in Section 4.2.4).

(CHECK THIS NEXT EXAMPLE FOR BUG WITH 3 rather than 2 object!!)

```
NAME_COLLECTION_IN_FILE inline 2
0    % first
col1 % The name of the collection, 'col1'
2    % The collection length, 2 entries will follow.
gm1
gm2
1    % second
col2 % The name of the collection, 'col2'
3    % The collection length, 3 entries will follow.
gm4
gm3
gm5

DT_IN_FILE inline 1
```

```
0       % first
map     % name of decision tree, 'map1'
1       % only one parent.
-1 (p0) % just copy value of parent
```

```
variable : obs1  {
   type: continuous observed 0:12 ;
   conditionalparents: foo(0) using mixGaussian
     collection("col1")
     mapping("map");
}
variable : obs2  {
   type: continuous observed 0:12 ;
   conditionalparents: foo(0) using mixGaussian
     collection("col2")
     mapping("map");
}
```

In the example above, two observation variables are defined `obs1` with parent `foo` and `obs2` also with parent `foo`. The decision tree `map` as defined is an identity operator, i.e., it maps whatever is the current parent value to the leave node. The process is as follows. First the value of the parent `foo` is obtained. This value is mapped through a decision tree to an integer index value. This value is used to look up the name in the corresponding position of the name collection object. The name of the Gaussian mixture in that position is then used for the observed variable.

For example, the setup above states that when `foo` is 0, `obs1` uses the Gaussian mixture named `gm1` but `obs2` uses the Gaussian mixture named `gm4`. When `foo` is 1, `obs1` uses the Gaussian mixture named `gm2` and `obs2` uses the Gaussian mixture named `gm3`. When (or if) `foo` is 2, `obs1` would lead to a run-time error, while `obs2` uses the Gaussian mixture named `gm5`.

Name collection objects will be described further in Section 4.2.3 when sparse CPTs are introduced.

4.2 Conditional Probability Tables CPTs

There are a number of ways of representing probabilities in GMTK, depending on if you are using discrete or continuous variables.

The conditional probability table (CPT) is the basic object representing discrete probability values in GMTK. GMTK actually has three different types of CPT objects, dense CPTs (DenseCPTs), sparse CPTs (SparseCPTs), and deterministic CPTs (DeterministicCPTs). The different type of CPTs to use will depend on what function the random variables are mean to perform, and in certain cases (such as language modeling) on the available amount of memory and computation.

4.2.1 Dense CPTs

Lets start with an example of a DenseCPT. Suppose, for example, you were interested in representing the quantity $P(X)$ where X is a 10-valued discrete random variable. A structure file might look something like

```
variable : X {
  type : discrete hidden cardinality 10;
  switchingparents : nil;
  conditionalparents : nil using DenseCPT("X_prob_table");
}
```

This declaration of X specifies that there must be a dense CPT that is named `X_prob_table`. This is declared in the object file as follows:

```
1             % The number of DenseCPTs
0             % The index number of the first DenseCPT (zero start)
X_prob_table  % the name of the DenseCPT, which is 'X_prob_table'
0             % number of parents, zero in this case.
10            % number of values, or cardinality of the RV using this CPT
% finally, 10 probability values which in this case indicate
% a uniform distribution.
0.1 0.1 0.1 0.1 0.1
0.1 0.1 0.1 0.1 0.1
```

This is an example of a one-dimensional CPT. As X has no parents, the CPT need only specify 10 probability values. The first probability corresponds to $P(X = 0)$, the second to $P(X = 1)$, and so on until $P(X = 9)$ (recall from Section 2.1.1 that all discrete random variables have a possible value starting from 0 to the cardinality minus one). The number of entries in the the CPT is specified in two ways, first by explicitly stating how many values there will be which is then followed by that many values (this is another example of reducing the chance of errors by slightly increasing redundancy in the files). Any random variable that uses this CPT therefore must have two qualities. First, it must not have any parents (as it is a 1-D CPT). Second, the variable must have a cardinality of exactly 10. If either of these conditions do not hold, then a run-time error will occur.

Lets do a 2-D example now. Suppose we declare a variable in the structure file as follows:

```
variable : M { type : discrete hidden cardinality 4; ... }
variable : C {
  type : discrete hidden cardinality 3;
  switchingparents : nil;
  conditionalparents : M(0) using DenseCPT("C_given_M");
}
```

The variable C then assumes that a DenseCPT named `C_given_M` will exist in the parameter files. In this case, C has cardinality 3 but M has cardinality 4 meaning that `C_given_M` must have 12 entries. This is declared as follows:

```
1 % The number of DenseCPTs
0 % The index number of the first DenseCPT (zero start)
C_given_M % name of this CPT
1 % one parent
4 3 % parent cardinality (4 in this case) and self cardinality (3)
```

```
% the 4x3 table. For each value of the parent M, we have
% three probabilities for each of the possible child values. This
% means that P(C=c|M=m) = A_{mc} where m is the row, and c is the column,
% and that this is the order the probabilities are given in the table.
0.2 0.4 0.4 % P(C=0|M=0), P(C=1|M=0), P(C=2|M=0)
0.4 0.4 0.2 % P(C=0|M=1), P(C=1|M=1), P(C=2|M=1)
0.3 0.4 0.3 % P(C=0|M=2), P(C=1|M=2), P(C=2|M=2)
0.1 0.1 0.8 % P(C=0|M=3), P(C=1|M=3), P(C=2|M=3)
```

Finally, a 3-D example. We declare a variable in the structure file as follows:

```
variable : F { type : discrete hidden cardinality 4; ... }
variable : M { type : discrete hidden cardinality 2; ... }
variable : C {
  type : discrete hidden cardinality 3;
  switchingparents : nil;
  conditionalparents : F(0),M(0) using DenseCPT("C_given_F_M");
}
```

The variable C assumes that a DenseCPT named `C_given_F_M` exists in the parameter files. In this case, C has cardinality 3, F has cardinality 4, and M has cardinality 2 meaning that the table must have 24 entries.

The order that the parents are declared in the 'conditionalparents' line above determines the order of the parents used to index probability values in the table `C_given_M_F`. The first parent declared (`F` in this case) is the outer-most index, the second parent (`M`) is the next index, and so on. The table is declared follows:

```
1 % The number of DenseCPTs
0 % The index number of the first DenseCPT (zero start)
C_given_F_M % name of this CPT
2 % two parents
4 2 3 % parent cardinalities (here, 4 and 2) and self cardinality (3).
% The 4x2x3 table -- for each value of the pair F and M, we have
% three probabilities for each of the possible child values. This
% means that P(C=c|F=f,M=m) = A_{fmc} where f is the outer (major) row
% index, m is the inner (minor) row index, and c is the column index,
% and that this is the order the probabilities are given in the table.
0.2 0.4 0.4 % P(C=0|F=0,M=0), P(C=1|F=0,M=0), P(C=2|F=0,M=0)
0.4 0.4 0.2 % P(C=0|F=0,M=1), P(C=1|F=0,M=1), P(C=2|F=0,M=1)
0.3 0.4 0.3 % P(C=0|F=1,M=0), P(C=1|F=1,M=0), P(C=2|F=1,M=0)
0.1 0.1 0.8 % P(C=0|F=1,M=1), P(C=1|F=1,M=1), P(C=2|F=1,M=1)
0.1 0.5 0.4 % P(C=0|F=2,M=0), P(C=1|F=2,M=0), P(C=2|F=2,M=0)
0.9 0.0 0.1 % P(C=0|F=2,M=1), P(C=1|F=2,M=1), P(C=2|F=2,M=1)
```

```
0.3 0.3 0.4 % P(C=0|F=3,M=0), P(C=1|F=3,M=0), P(C=2|F=3,M=0)
0.7 0.2 0.1 % P(C=0|F=3,M=1), P(C=1|F=3,M=1), P(C=2|F=3,M=1)
```

The CPT must declare the cardinalities of the variables which are going to use that CPT. These cardinality values serve a dual purpose of also defining the dimensionality of the table, and ensuring that a table is compatible with a set of parent and child random variables values which wish to use that table.

4.2.2 Special Internal Unity Score CPT

Like in the Gaussian case (see Section 4.4.3), GMTK supports a special pre-defined and internal CPT that in effect always produces a unity score no matter what the argument of the distribution might be. This CPT is named `internal:UnityScore` and can be used in place of a Dense CPT name in the structure file. There are restrictions when this can be used, however, and these include:

- The random variable must (of course) be discrete.
- It must be observed.
- It must use a Dense CPT implementation that has no (i.e., `nil`) conditional parents (for a given switching parent value).
- The CPT, however, may be used regardless of the cardinality of the random variable in question.

This special CPT can be useful for a number of models, most importantly for discrete conditional-only observations (similar to the continuous conditional-only features that are available with GMTK's Gaussians). For example, suppose the discrete random variable O is always observed for a given utterance. One therefor can compute conditional probabilities without seeing any affect of the probabilities of the conditional-only observations. In particular,

$$P(A|O=o)P(O=o) = P(A|O=o)$$

where A is an arbitrary random. The equation is true because essentially the CPT produces an effect where $P(O=o)=1$ for all values o. Note that this does not produce normalization irregularities because the variable O must not have any parents anywhere in the network, and it must be observed.

4.2.3 Sparse CPTS

In this section, you will learn about SparseCPT objects, a way of representing a CPT in a sparse way.

As defined above, a CPT (conditional probability table) is a table representation of a discrete conditional probability distribution. Suppose you are interested in representing the discrete distribution

$$p(x_1|x_2, x_3, \ldots, x_N)$$

where each of the random variables $X_1, \ldots, X_N$ are discrete. For this discussion, lets assume that all the random variables can have the same number of values (cardinality), the integers in the range between 0 and $K-1$ (so there are K values in total for each variable). Therefore, the table

above requires a total of K^N entries. For each set of values of the parents $X_2, \ldots, X_N$ (and there are K^{N-1} possible values), there are an additional K values needed for X_1 leading to K^N. If you were to represent this table using a DenseCPT, this could require a pretty large table, too big to store in memory (depending on the values of K and N). Furthermore, if you knew that for many parent values, and values of x_1, the probabilities are zero, it would be wasteful to store all the values (not to mention the fact that when doing inference, GMTK would need to iterate over all such values to check which ones are zero).

SparseCPTs solve this problem. A SparseCPT is a representation of $p(x_1|x_2, x_3, \ldots, x_N)$ where zeros are not represented. In other words, a SparseCPT is a way of representing a sparse singly-stochastic arbitrary-dimensional matrix (a matrix where one of the dimensions has "rows" which are probability mass functions). These individual mass functions have many entries which are zero. Sparsity is achieved in a SparseCPT using the following two methods.

1. A decision tree is used to map from the set of parent values $X_2, X_3, \ldots, X_n$ to integers. You can think of $x_2, x_3, \ldots, x_N$ as an $(N-1)$-dimensional discrete vector space. A decision tree carves this space up into a set of arbitrarily shaped regions of dimension $(N-1)$ or less. Each region is then mapped to some integer corresponding to the leaf nodes of a decision tree. Note that the regions need not be connected – this can be done by having multiple leaf nodes in the decision tree (Section 4.2.4) specify the same integer.

2. The integer results from 1) are used to index into a table of sparse 1-dimensional probability mass functions (SPMFs). An SPMF (see also Section 4.3.2) represents only the values of X_1 that have non-zero probability (for a given set of values of the parents $x_2, x_3, \ldots, x_N$). The zero probability values of x_1 are not represented, and therefore don't take up any resources.

Of course there is a little bit more to do than this implies, since we need to define the mappings, and how to create SPMFs. Also, SparseCPTs allow any of the SPMFs to be shared, and moreover, they allow different SPMFs to share the same underlying set of probabilities even if they correspond to different values of the random variable (this can also occur during EM training, and therefore could help to mitigate the effects of data sparsity issues).

Lets start off with SPMFs. An SPMF is a representation of a 1-dimensional probability mass function, where presumably many of the entries in this mass function are zero. Rather than using an array of probabilities with lots of zeros, an SPMF consists of two arrays. The first array contains the set of possible (i.e., non-zero probability) values of the random variable (the possible non-zero probability values of X_1 above) and the second array contains the probabilities themselves.

In the master parameter file, an SPMF can be specified as follows:

```
% SPARSE PMFs
SPMF_IN_FILE inline % SPMF keyword "SPMF_IN_FILE", and "inline"
                    % to indicate that they are contained right
                    % here rather than in another file.

2 % the number of SPMFs that will follow (only two in this case)

0           % SPMF index number 0 in this SPMF set.

spmf_name0 % SPMF_NAME: the name of this SPMF

42          % CARD: The cardinality of this SPMF (42 in this case). This
```

```
            % means that the random variable that this
            % SPMF is used for (e.g., X1 above) has 42 possible
            % values (including the values which might
            % have zero probability). Note that in this case
            % the values are integers in the range of 0 to 41.

16          % LENGTH: The length of this SPMF. This number gives
            % the number of non-zero random variable values
            % that this SPMF specifies. In this case,
            % it is saying that there are only 16 of the
            % 42 values which do not have zero probability.
            % Note that in some cases it might not be worth
            % using a SPMF if LENGTH >= CARD/2 since
            % a SPMF requires two tables.

              % TABLE: The actual set of values that this random variable
0   1  2  3   % might take on. This means that
5   6  7  8   % p(X1=4) = p(X1=9) = p(X1=14) = p(X1=19) = 0
10 11 12 13   % and that any other value not specified
15 16 17 18   % in the table (namely 19-41) also has
              % zero probability. Note that a value
              % can not be less than zero, nor can one
              % be greater (in this case) than 41.
              % Note that the fact that the values here are arranged
              % in a 4x4 matrix has no significance. The values
              % can be arranged in all one row, all one column, or
              % anything in between.

              % DPMF_NAME: the name of the DPMF (dense 1-D PMF)
dense_pmf     % which holds the 16 probability values
              % which are used for this SPMF and are associated
              % with each of the values in TABLE respectively. This name
              % can be any of the DPMF objects which
              % have been defined earlier. DPMFs also
              % have a cardinality, and in this
              % case the DPMF cardinality must be equal to 16 (not 42)
              % or you'll get an error message.

1           % SPMF index number 1. The next SPMF
            % was actually used for a language
            % modeling experiment. This is (a little) more
            % like the SPMFs that you are likely to encounter
            % when you start using SPMFs for real problems.
            % Of course, you probably would want to generate
```

```
          % it by script.
20000 604 % This r.v. has a card of 20000, and the length is 604.
          % What follows are the 604 (well almost) non-zero prob. values.
3 4 5 6 7 18 21 28 29 41 71 194 195 202 203 210 211 269 270 348 362
368 417 418 419 424 436 470 471 472 480 481 492 493 495 496 528 546
...
13108 13109 13110 13116 13117 13118 13119 13159 13161 13166 13189
13220 13248 13250 13251 13252
_COMMA  % finally the name of the corresponding DPMF, named '_COMMA'
```

Several SPMF objects can share the same DPMF object by specifying the same DPMF name. This is how parameter sharing of DPMFs by SPMFs are specified. During EM training, the contributions of both SPMF objects will increase the counts of the DPMF.

Also note, that the DPMF objects are exactly the same objects use for the probability coefficients of Gaussian mixture distributions. Therefore, a Gaussian mixture object can share its mixture DPMF with an SPMF. If you find a use for this (for some reason) make sure that you are aware that any Gaussian component vanishing and splitting will change the cardinality of the DPMF at which point it will no longer match the SPMF, and you'll get a run-time error.

Ok, so that describes SPMFs. We said earlier that decision tree leaf node integers map to the index entries in a large table of SPMFs. Entire SPMFs are shared in this way by having regions in $x_2, \ldots, x_N$ space refer to the same integer (and thereby an SPMF index).

Internal to GMTK, there is a global table of SPMF objects. In a master file, each time GMTK encounters a line of the form:

```
    SPMF_IN_FILE inline
    ...
```

or of the forms

```
    SPMF_IN_FILE filename.spmf ascii
    SPMF_IN_FILE filename.spmf binary
```

it will append any SPMFs that follow (either inline, or the ones contained in filename.spmf) to this global array of SPMF objects (this is the same way all GMTK objects work actually).

In order to have the decision tree integer indices of a SparseCPT specify particular SPMF objects, it must somehow specify the objects in this table. Rather than having these indices index directly into the global table (which would be difficult to manage when there are multiple sets of SPMF objects around), a SparseCPT uses a "name collection" object.

A name-collection object is just an array of names. If a collection gives a list of SPMF names, then a SparseCPT can use that collection. The SparseCPT does so by having its decision tree integer indices index directly into the collection. It therefore doesn't need to worry about the global index locations of the SPMF objects (there is, however, a special internal collection that is called "global" which you can use to index into the global table). Each SparseCPT can have its own collection, which makes managing the indices relatively easy.

So, that is basically it. Lets summarize in a simple ASCII-graphic tree, the objects that are required to specify a SparseCPT.

This will ultimately be a figure

```
SparseCPT
   |
```

```
+-- the SparseCPT's parameters
|     1) number of parents (e.g., N-1 in example above)
|     2) cardinalities of each parent (e.g., for vars X2,...,XN above)
|     3) cardinality of self (e.g., for var X1)
|
+---- Decision tree
|       A standard definition of a DT
|
+---- Collection object
          |   (A collection object that gives a list of SPMF names.
          |    Via these SPMF names, it refers to the following
          |    objects:).
          |
          +---- SPMF1 % the first SPMF refered to by the collection
          |      |
          |      +-- SPMF1's parameters
          |      |      1) the cardinality of self (e.g., for var X1)
          |      |      2) the length L (number of non-zero probabilities)
          |      |
          |      +-- DPMF
          |      |    |
          |      |    +--- The DPMF's parameters
          |      |    |        1) the length L (specified again)
          |      |    |
          |      |    +--- a list of L probability values
          |      |
          |      +-- list of L integer values giving the
          |          integers values of the random variable that
          |          do not have zero probability.
          |
          +---- SPMF2
          |      |
          |      + Same as above, but for SPMF2 ...
          |
          +---- SPMF2
          |      +...
          ...
          |
          +---- SPMFM (the last SPMF specified by the collection)
                 +...
```

Lastly, here is a complete example containing portions of all the definitions needed to set up a SparseCPT. These file portions will live in your master file, except for the final DPMF objects which are trainable and which could live either in a master file or a trainable file (a file containing objects which might change as a result of EM training).

First, the SparseCPT object itself:

```
SPARSE_CPT_IN_FILE infile
```

```
1 % one Sparse CPT

0 % index number

mySparseCPT   % name of 1st SparseCPT

2             % there are two parents, so this is a
              % SparseCPT for P(X1|X2,X3)

2 3           % cardinalities of parents. So the variable
              % X2 is binary valued (value 0 or 1), and
              % and the variable X3 is ternary valued
              % (values 0, 1, or 2).

5             % Cardinality of self, so X1 is a
              % quinary random variable (one with
              % five values)

myDT          % the name of the decision tree that
              % this SparseCPT will use.

myCol         % the name of the collection used by
              % this SparseCPT
```

Next, we'll give the definition of the decision tree. See section 4.2.4 for more details on decision tree syntax.

```
DT_IN_FILE inline

1 % one decision tree
0 % index number

myDT    % The name of the decision tree

% There are 6 possible values of the X2,X3.
% This decision tree will map
%    (X2=0,X3=(0,1)),(X2=1,X3=(2)) => the first SPMF
%    (X2=0,X3=2),(X2=1,X3=(0,1))   => the second SPMF

2       % two parents, must be same as mySparseCPT.
% The following line with '0 2 0 default' has four values meaning:
% (parent_specifier=0, specifying X2), (num_splits=2), (split 1) (default)
0 2 0 default
    1 2 2 default
```

```
       -1 1   % second SPMF
       -1 0   % first SPMF
    1 2 2 default
       -1 0   % first SPMF
       -1 1   % second SPMF
```

Here is the definition of the collection. It is quite simple as it only has two entries, one for each of the SPMF objects.

```
NAME_COLLECTION_IN_FILE inline

1 % one collection
0 % index 0

myCol % Collection name.
2     % length, only two entries.
      % The length-two set of spmf names.
mySpmf0
mySpmf1
```

Note that the names "mySpmf0" and "mySpmf1" refer to the global set of SPMFs. In other words, if another collection of SPMFs used the same names, then they would refer to the same SPMF objects (this is one way, in fact, to achieve parameter sharing across different SparseCPTs).

Next come the definitions of the two spmfs.

```
SPMF_IN_FILE inline

2 % two SPMFs

0 % First spmf.
mySpmf0 % SPMF name.
5       % Cardinality of X1.
3       % Number of values with non-zero probability.
0 1 2   % So in this case, X1 can only take values 0 1 or 2
        % with non-zero probability.
myDpmf0 % Name of corresponding length 3 DPMF.


1 % Second spmf.
mySpmf1 % SPMF name.
5       % Cardinality of X1.
2       % Number of values with non-zero probability.
0 4     % So in this case, X1 can only take values 0 or 4
        % with non-zero probability.
myDpmf1 % Name of corresponding length 2 DPMF.
```

And finally, the definition of the two DPMFs needed above.

```
DPMF_IN_FILE inline

2 % two DPMFs

0 % First DPMF.
myDpmf0 % Name.
3       % DPMF length.
0.25 0.5 0.25 % The three probability values.

1 % Second DPMF.
myDpmf1 % Name.
2       % DPMF length.
0.25 0.75 % The two probability values.
```

4.2.4 GMTK Decision Trees

GMTK uses decision trees for a variety of different functions. These functions include:

1. Deterministic CPTs, which are used when for each set of values of a collection of parent random variables, there is one and only one possible value having non-zero probability for the child variable.

2. Sparse CPTs, to specify arbitrary regions in the space possible values of a collection of parent variables, and map from those regions to SPMFs. Essentially the decision tree maps to a particular row entry in the CPT.

3. To map from a collection of discrete hidden parent random variables to a value for a child observation variable. The value for the child is then used to determine a Gaussian mixture object for that particular combination of parent values.

4. To map from the values of a collection of switching parent variables to one of a set of conditional parent variables.

These uses are elaborated upon further in Section 4.2.4.

Like all parameter objects, the decision tree syntax was designed for easy and fast parsing, while including redundancy so that the chance reading in an erroneous decision tree is reduced.

In general, a decision tree is used in GMTK to map from a values of a collection of discrete random variables down to a single integer value. That integer value is then used as appropriate for whatever function the decision tree is needed. Decision trees are regular GMTK objects, defined in the same way as all GMTK objects, and as described in Section 3.2.

In general, a decision tree is a tree where each node of the tree contains or uses some query, and the answer to the query at each node determines which single branch of the tree to descend. Note that there are two parts to any query: 1) the object (or set of objects) that are being queried, and 2) the actual question about that object. This is important since the GMTK decision tree syntax separates the two. At the decision tree leaf nodes, the final "answer" of the decision tree (i.e., the set of questions) are found. In a GMTK decision tree, all leave nodes have as their answer a single

integer (which is used for different purposes depending on where the decision tree is used, see above).

A general recursive definition of a GMTK DT (decision tree) is given in the following:

```
DT :=
% A decision tree is either a leaf node ...
  -1 integer_formula   % leaf nodes start with '-1'
% ... or is a branching node, a recursively defined decision tree
| parent_id N split_range_1 split_range_2 ... split_range_{N-1} "default"
   % parent_id says which parent to query. N = number of splits
     DT_1           % DT_1 used if split_range_1 is true.
     DT_2           % DT_2 used if split_range_1 is true.
     ...            % ...
     DT_{N-1}
     DT_{default}   % DT_{default} used if no split ranges are true.
```

This means that a decision tree is either a leaf node ("-1" followed by an integer formula) or alternatively is a branching node.

In the case of a branching node, `parent_id` is an integer in the range $0, 1, \ldots, K$ giving the index of the parent (or feature using the terminology of decision trees) to query, where K is one less than the total number of possible parents. This means that if the decision tree is used with a random variable having $K+1$ parents, then `parent_id` specifies which parent should be queried. The `parent_id` specifies what the query at a given DT node is about, but does not specify the actual set of questions which are used to determine which branch to descend. The questions are determined by the range specifications, as described below.

N is the number of splits in that node in the tree. This means that a GMTK decision tree can have any branching factor at each node. The string `split_range_i` is an integer range specification, and it specifies an integer range of values to use for that branch in the tree. A given integer range is really a syntax for specifying a set of integers, and if the parent (as given by `parent_id`) has a value that lies within a given range's set, then the answer to that particular question is yes.

An integer range is either a list of comma separated integers, or a matlab-like range specification (see Section 5.1). The integers specified, however, must be non-negative. The "default" string is the catch-all case, which is used if all of the split ranges fail.

The `DT_i` tags are recursively defined sub-decision trees trees having exactly the same syntax: they can either be another decision tree or can be a leaf node. There must be as many `DT_i` sub-decision trees as there are splits (N in the above). The last tag `DT_default` is used for the catch-all case.

In the case of a leaf node, the special value "-1" specifies this is the case, and `integer_formula` gives the integer formula corresponding to that leaf node. An integer formula may either be a simple solitary integer (giving the value of the leaf node), or it may be a simple additive integer formula (enclosed in parentheses) which can contain any of the special variables p_i (parent i's value), c_i (parent i's cardinality), or m_i (parent i's value multiplied by its cardinality).

Finally, a decision tree must be given a **name** (any textual string, like all GMTK objects). Following the name must be an integer that specifies the maximum **number of parents** (also called number of "features" in decision tree lingo) that the decision tree might use. In GMTK, 'features' and 'parents' are used synonymously when used in the context of decision trees, meaning that

queries made of a DT feature is really made on parent random variable values. For a GMTK DT, it is only parent random variable values which can be the inputs to DTs. Note that the number of parents of a decision tree should therefore match the number of parents that a random variable has when it tries to use that decision tree. Typically, decision trees have a number of possible features (i.e., multiple parents) that they can query. In GMTK, the different set of instantiated random variable parent values exactly constitute the set of features that the decision tree can query.

Note also that the number of parents specified with a decision tree must match the context in which it is used. If it is attempted to use a DT with a different number of parents than the number of parents in a CPT (or set of random variable parents), then a GMTK run-time error will occur.

Lets give a few examples, starting with the easiest of decision trees, one that returns a constant.

```
myDT 1 % name of the DT 'myDT', and its number of parents (one)
-1 0
```

The decision tree first specifies its name `myDT`, and then the number of parents (one, in this case). Then, there is only one leaf node (specified by the "-1" tag), and the value of the leaf node is 0 regardless of any of the parent values.

For our next example, we copy the value of the first parent to the child, ignoring the value of the second parent.

```
myDT2 2 % name of DT 'myDT2', and number of parents (two)
-1 (p0)
```

Here, there are two parents. The leaf node is the value `p0` surrounded in parentheses, saying that the value of the decision tree is whatever is the current value of the 1st parent (i.e., parent zero). This first parent corresponds to the left most parent in the corresponding structure file.

The next example implements the function that adds one to the first parent if its value is less than 10 but otherwise just copies that parent value.

```
myDT3 1 % name 'myDT3', and 1 parent.
0 2 0:9 default % query parent 0, two splits '0:9' and 'default'
  -1 (p0+1)     % if first split criterion true, go here
  -1 (p0)       % else, if second split criterion true, go here
```

This decision tree says that if the first parent (parent 0) is in the range of values between 0 and 9 inclusive, then the result of the decision tree is `(p0+1)`, and is otherwise just `(p0)`.

As another example, suppose there are three random variables A, B, and C all discrete, and we wish to specify the following mapping from jointly A, B, C to a single resulting integer.
If $A = 0$, and $C = 0$ then 0.
If $A = 0$, and $C \neq 0$ then 1.
If $A = 1$, then 2.
If $A \neq 0$, $A \neq 1$, and $B = 1$, then 3.
If $A \neq 0$, $A \neq 1$, and $B \neq 1$, then 4.

This can be done using a decision tree as follows:

```
myDT4 3 % name 'myDT4', 3 parents, parents 0,1,2 are A,B,C respectively.
0 3 0 1 default % parent 0 (A), 3 splits: '0', '1', and 'default'
```

```
  % if A = 0
  2 2 0 default    % parent 2 (C), 2 splits, '0' and 'default'
      -1 0         % leaf node returning 0
      -1 1         % leaf node returning 1
  % if A = 1
 -1 2              % leaf node returning 2
  % if A is not 0 or 1
  1 2 1 default    % parent 1 (B), 2 splits, '1' and 'default'
     -1 3          % leaf node returning 3
     -1 4          % leaf node returning 4
```

The next example queries the 2nd parent first, and if its value is 0 returns the value of the 3rd parent. Otherwise (i.e., if the 2nd parent does not have value 0), then the decision tree queries the 1st parent. If that first parent has value 1 then the tree returns the value 0, otherwise (if the 1st parent does not have value 1) then the tree returns the value of the 3rd parent plus one.

```
myDT5 3 % name 'myDT5', 3 parents
1 2 0 default
         -1 (p2)
         0 2 1 default
                 -1 0
                 -1 (p2+1)
```

A decision tree file consists of some number of decision trees. The first number in a DT file indicates how many DTs there are, and each DT is preceded by its relative order in the file (starting at number zero for the first DT, one for the second, and so on). This is the same as for all GMTK objects as described in Section 3.2. Here is an example of a DT file:

```
% this is a DT file, typically has extension .dt or .dts
3 % there are three decision trees in this file

0 % the first DT
the_first_DT % this is the name of the DT
1            % one parent
0 0:9 default
  -1 (p0+1)
  -1 (p0)

1 % the second DT
foo_DT % this is the name of the second DT
1      % also one parent
-1 (p0)

2 % the third DT
bar_DT % this is the name of the third DT
3 % 3 parents in this case
```

```
0 3 0 1 default
   2  2 0 default
       -1 0
       -1 1
  -1 2
   1 2 1 default
      -1 3
      -1 4
```

Per-Utterance Decision Trees

Sometimes, when building a mapping from one set of variables (say A,B,C) to another variable (say D), there is one and only one DT for all times that is used to implement that mapping. So that DT is specified once, and we're done.

Other times, the DT mapping will need to change once for each utterance. For example, during training of an ASR system, we know the training set transcription (i.e., what was said, we have both a speech waveform and the string of words indicating precisely the utterance that was spoken). DTs are used to implement the mapping between variables that encodes the transcription for each utterance. For example, the DT might specify the set of phones that corresponds to the utterance.

Here is a toy example. Suppose we have two possible utterances "CAT" and "DOG", where we use the transcriptions `/C/` (phone 0, i.e., phone zero), `/A/` (phone 1), `/T/` (phone 2) for "CAT" and `/D/` (phone 3), `/O/` (phone 4) and /G/ (phone 5) for "DOG". In the structure file, suppose that there is one random variable Q_t that indicates the current phone for each time t. Q_t must evolve over time in a way that traces out the appropriate transcription of the current utterance.

For "CAT", a decision tree might look something like:

```
cat_DT 1 % name, and number of parents (=1)
0 2 0 default
    -1 1
    -1 2
```

which implements the mapping:

```
      C -> A
      A -> T
```

But for DOG, we might have a DT that implements:

```
dog_DT 1 % name, and number of parents (again =1)
0 2 3 default
    -1 4
    -1 5
```

which implements the mapping:

```
      D -> O
      O -> G
```

There is a special syntax for decision trees that need to change for each utterance, and is given in the following master file (note the keyword `DT_IN_FILE` which introduces an inline section containing one decision tree).

```
DT_IN_FILE inline 1
% a DT to map from a word counter to a particular word.
% This takes its implementations from a file named 'transcription.dts',
% each utterance will have a different DT implementation.
Q_DT      % name of DT 'Q_DT'
counterToWordMap.dts   % file to take DT implementation.
```

In this case, the file `counterToWordMap.dts` must contain as many decision trees as there are utterances in the utterance file. The i^{th} decision tree is used for the i^{th} utterance. If there were two utterances, then this decision tree file might look like:

```
2 % number of decision trees
0 % DT 1
cat_DT 1 % name, number of parents

0 2 0 default
    -1 1
    -1 2
1 % DT 2
dog_DT 1 % name, number of parents
0 2 3 default
    -1 4
    -1 5
```

Note that in a per-utterance decision tree file, one may use the same name for all of the decision trees. For better error message reporting, however, it is advisable to use different names for each such DT.

Uses of Decision Trees

Now that we are comfortable with the syntax of decision, we elaborate upon the uses of decision trees that were sketched earlier in this section.

1. DTs are used to map from a collection of hidden variables to a particular Gaussian mixture. Suppose we are given the following structure:

```
variable : obs  {
   type: continuous observed OBSERVATION_RANGE ;
   switchingparents: nil;
   conditionalparents: wholeWordState(0) using mixGaussian
     collection("global")
     mapping("directMappingWithOneParent");
}
```

In this case, `obs` is an observation variable which has a hidden parent `wholeWordState`. It uses a DT named `directMappingWithOneParent` to map from values of `wholeWordState` to an index into the global (program wide) collection of Gaussians for the entire ASR system.

2. DTs are used to map to a specific row entry in a sparse CPT. Suppose we are interested in implementing $P(A|B,C,D)$. Sometimes CPTs can be sparse, which means that for each value of the tuple B,C,D there might be only a small subset of possible values for A which have non-zero probability. E.g., suppose that when $B = 1$, the event $A = 2$ is impossible.

   A sparse CPT uses a DT to map from regions in the tuple (B,C,D) to sparse 1-dimensional arrays that give the possible values of A that have non-zero probability.

   A DT is used to map from the collection of parent random variables to an integer which then maps indirectly to a Sparse CPT (i.e., a multi-dimensional Sparse CPT).

   Sparse CPTs are described in Section 4.2.3.

3. DTs are also used to map to a specific collection of conditional parents from a set of switching parents. Suppose we have the following structure segment:

```
variable : word {
   type: discrete hidden cardinality 100 ;
   switchingparents: wordTransition(-1)
     using mapping("directMappingWithOneParent");
   % if there was a word transition, use the bigram, otherwise,
   % this is a copy of previous self
   conditionalparents:
       word(-1) using DeterministicCPT("copyMTCPT")
     | word(-1) using DenseCPT("wordBigram");
}
```

   Here, `wordTransition(-1)` is a switching parent. We use the DT named `directMappingWithOneParent` to map from values of `wordTransition(-1)` to either 0 or 1 (since there are only two sets of conditional parents) indicating if either `word(-1)` should be used as a parent (with `DeterministicCPT("copyMTCPT")` as the CPT), or if `word(-1)` should be used as a parent with `DenseCPT("wordBigram");` as the CPT. Note that switching parents need not just switch the parents, but they can also switch the CPT that is used for the parents. In the example above, the parent `word(-1)` is the same, but the CPT used for that parent switches. Another example might switch the actual set of parents.

4. Finally, DTs are used to implement deterministic dependencies between variables (when A is a deterministic function of B). Here a DT is used in conjunction with a `DeterministicCPT`. Given the following structure fragment:

```
variable : wholeWordState {
   % number of words X number of states per word
   type: discrete hidden cardinality NUM_WHOLE_WORD_STATES;
   switchingparents: nil;
   conditionalparents: word(0), wordPosition(0) using
              DeterministicCPT("wordWordPos2WholeWordState");
}
```

Here, we have a variable `wholeWordState` which is a deterministic function of parents `word(0)`, `wordPosition(0)` using a DeterministicCPT implementation `wordWordPos2WholeWordState` which in turn uses the DT that lives in the file `wordWordPos2WholeWordState.dts` and is named `wordWordPos2WholeWordState`. Deterministic CPTs are described in Section 4.2.5.

4.2.5 Deterministic CPTs

After having mastered sparse CPTs, deterministic CPTs are quite easy. As specified in Section (NOT-YET-WRITTEN), it is quite beneficial (both for computational and memory reasons) for relations between certain random variables in GMTK to be specified deterministically, rather than producing large tables containing an overwhelming majority of zeros.

A deterministic CPT object uses a decision tree to specify the deterministic relationships between sets of parent random variables and a child "random" variable. A deterministic CPT is just another GMTK object which specifies how many parents the CPT has, the cardinalities of the parents and the self, and then gives a reference to an existing DT which implements the dependencies.

Here is an example of a deterministic CPT contained in a master file:

```
DETERMINISTIC_CPT_IN_FILE inline 1
0         % first DETERMINISTIC_CPT
myDETCPT  % name of the CPT
1     % num parents of the corresponding random variable
2 3   % cardinalities. cardinality of parent = 2, of self = 3
myDT  % The name of an existing DT to implement the deterministic
      % mapping
```

In this example, the CPT specifies its order in a list (the list has only one CPT in this case), its name 'myDETCPT', the number of parents corresponding to this CPT (for example, this could implement a CPT for say $P(B|A)$), the cardinalities of the parents and self (meaning, for example, that A is a binary random variable and B is a ternary random variable), and finally giving the name 'myDT' referring to a decision tree that implements the dependency.

The next example is a deterministic CPT which more than one parent. Suppose we wish to create a dependency of the form $P(D|A, B, C)$. This can be declared using the following structure fragment:

```
variable : A { type: discrete hidden cardinality 2 ; ... }
variable : B { type: discrete hidden cardinality 3 ; ... }
variable : C { type: discrete hidden cardinality 4 ; ... }
variable : D {
      type: discrete hidden cardinality 5 ;
      conditionalparents: A(0),B(0),C(0) using
          DeterministicCPT("D_given_ABC"); }
```

The deterministic CPT `D_given_ABC` could be declared as follows:

```
D_given_ABC     % name of the CPT
3               % three parents
```

```
2 3 4 5          % cardinalities of A,B,C, and D (self) respectively.
D_given_ABC_DT   % The name of the DT to use for the mapping
```

Note that the cardinalities of the CPT must match the cardinalities given the structure file for the variable that uses the CPT. This CPT could then use the following decision tree.

```
D_given_ABC_DT % DT name
3              % number of parents
0 3 0 1 default
   2 2 0 default
       -1 0
       -1 1
  -1 2
   1 2 1 default
      -1 3
      -1 4
```

The decision tree in this case does not specify the cardinalities of the random variable. A DT can therefore be used for multiple deterministic relationships amongst random variables having different cardinalities It is the job of the deterministic CPT to state its cardinalities (just like sparse and dense CPTs), which must match that of the corresponding random variables.

You must ensure that it is not possible for the resulting leaf nodes to produce a value that is not in the range 0 through $c-1$ where c is the cardinality of the child variable – if this occurs, a run-time error will occur. This could potentially be frustrating as such an error is not possible to detect until possibly long after the program has started. Therefore, particular care should be placed into ensuring that the DTs used can not have out of bounds leaf values.

4.3 Simple 1-D distributions: SPMFs and DPMFS

There are certain GMTK objects that implement one-dimensional CPTs that have a special functions, including the probabilities for sparse CPTs, or the mixture coefficients (responsibilities) for Gaussian mixtures. This section describes the syntax for these objects, dense probability mass functions (DPMFs) and sparse probability mass functions (SPMFs).

4.3.1 Dense probability mass functions (DPMFs)

DPMFs are simple one-dimensional CPTs, meaning CPTs with no parents, so they implement, say, $p(A)$ for some random variable A. They are declared in the way described in Section 3.2. Here is an example of a couple of DPMFs inline in a master file.

```
DPMF_IN_FILE inline 3
0       % DPMF number
mydpmf1 % DPMF name 'mydpmf1'
1 1.0   % The number of probabilities (one) and the probability

1       % DPMF number
```

```
mydpmf2 % DPMF name 'mydpmf2'
% The number of probabilities (4) and the probabilities
4 0.25 0.25 0.25 0.25

2       % DPMF number
mydpmf3 % DPMF name 'mydpmf2'
% The number of probabilities (3) and the probabilities
3 0.2 0.2 0.6
```

As can be seen, a DPMF consists of its name, followed by the number of probabilities that will follow, and then followed by that many probabilities. The number of probabilities corresponds to, for example, the number of mixture components in a Gaussian mixture in which case the probabilities indicate the component responsibilities.

Note that the number of probabilities for a DPMF might change when it is being used for Gaussian mixture responsibilities. This is because GMTK has a Gaussian component splitting/vanishing algorithm (described in Section 5.3) which can split or remove a particular component, and therefore grow or shrink the size of a DPMF. Care, therefore, should be exercised not to use a DPMF both for mixture responsibilities and sparse CPTs at the same time. If this does occur for some reason, a run-time error will be reported.

4.3.2 Sparse probability mass functions (SPMFs)

Sparse PMFs are used only with sparse CPTs, and are therefore fully described in Section 4.2.3 along with their use. To summarize, SPMFs are like DPMFs except they contain two arrays, one array giving the values of the random variable that have non-zero probability, and another array (a reference to a DPMF) of the same size that gives the non-zero probabilities for those values.

4.4 Gaussians and Gaussian Mixtures

Since Gaussians and Gaussian mixtures are so widely used, and since during training the number of components in a mixture might grow or shrink (via GMTK's splitting and vanishing algorithm), both Gaussian and Gaussian mixtures are basic objects in GMTK. These distributions are the only way at the moment to specify distributions over continuous observation vectors.

A basic Gaussian component contains a mean and a diagonal covariance vector. More advanced Gaussian components might have means that are conditional on other portions of the feature vector (either before, during, or after the current time frame). Other Gaussians might have full covariance matrices specified in this way. We begin, however, by describing simple diagonal covariance Gaussian mixtures.

4.4.1 Mean and Diagonal-Covariance Vectors

Gaussian components require a dimensionality, mean vector, and a covariance vector.

Mean vectors are quite simple. They require a name, a dimensionality, and a vector with that many mean values. Here is an example of several mean vectors.

```
MEAN_IN_FILE inline 3 % 3 mean vectors
0 % mean 1
```

```
% name, dimensionality (2), and values
mean0 2 1.0 1.0

1 % mean 2
% name, dimensionality (3), and values
mean1 3 -3.0 2.0 1.5

2 % mean 3
% name, dimensionality (5), and values
mean2 5 0.0 0.0 0.0 0.0 0.0

MEAN_IN_FILE file.means binary
```

The example first declares 3 mean vectors, all inline in a master file, and then declares some number of additional means which presumably live in a binary file named `file.means`. Of the means that are declared inline, the first one (with name `mean0`) is 2-dimensional with unity values, the second one (with name `mean1`) is three-dimensional with mixed positive and negative values, and the third one (with name `mean2`) is a five-dimensional zero-mean vector.

Mean vectors can be any dimensionality, and as can be seen the dimensionality must be specified along with the definition of the mean vector.

Diagonal covariance matrices (or actually vectors in this case) are defined using a similar syntax. Here is an example of two diagonal covariance matrices:

```
COVAR_IN_FILE inline 2 % two diag covariance matrices
0 % covar 1
covar0 10  % name and dimensionality (10)
% values
1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0

1 covar1 5
20 20 20 20 20
```

The first matrix `covar0` is a ten-dimensional unity-covariance vector, and the second is a five-dimensional covariance vector each with value 20. Note again that the dimensionality of the vector must be specified along with the covariance vector itself.

4.4.2 Gaussian Components

A Gaussian component groups together a mean and a covariance vector of the same dimension, and gives that grouping a name which is used to refer to the Gaussian component being defined. For example, here is a list of two Gaussian components (GCs).

```
GC_IN_FILE inline 2

0  % first GC
26 % dimensionality of the GC
0  % the 'type' of the Gaussian component. 0 = just mean & covar
```

```
gc_0 % the name of the component, here 'gc_0'
mean_0 covar_0 % the mean and variance of this component

1  % second GC
26 % dimensionality of the GC
0  % the 'type' of the Gaussian component. 0 = just mean & covar
gc_1 % the name of the component, here 'gc_0'
mean_1 covar_1 % the mean and variance of this component
```

These components assume the existence of the corresponding mean and covariance vectors *which must have the same dimensionality*. A Gaussian component also specifies a type (0 in the examples above) which indicate which type of component it is. The type of component specifies what additional objects are required to complete the component's definition. A component of type 0 requires only a mean and a diagonal covariance object, as given in the examples above. More on the type of component occurs in Section 4.4.4.

4.4.3 Special Internal Names: Zero and Unity Score Components

GMTK implements two special internal "Gaussians" components. They are not actually Gaussian at all but can be useful in certain circumstances for doing multi-stream and mult-rate models.

The first special component is called `internal:UnityScore`. It produces a component that will always return the value 1 (unity) regardless of the values of its arguments. I.e., $p(x) = 1, \forall x$. This therefore is not a true density, and one must be careful when using this density to ensure that the resulting model is probabilistically valid. Nevertheless, having this component can be indispensable for specifying certain models.

Another internal component density is defined as `internal:ZeroScore`. It is identical to the unity-score Gaussian except rather than always returning unity, it always returns zero.

4.4.4 Mixtures of Gaussians

Gaussian components (of any type) along with a DPMF can be bundled together to form a Gaussian mixture. Note that since different types of components can exist in a mixture, a mixture is able to consist a heterogeneous collection of components if desired.

A mixture of Gaussians is declared as in the following:

```
MG_IN_FILE inline 2

0   % mixture number
26  % dimensionality of the mixture
gm0 % the name of the Gaussian mixture.
2   % the number of components in this mixture
mx0_dpmf   % the name of the DPMF containing the component
           % responsibilities. This DPMF must have
           % dimensionality 2.
gc_0 gc_1 % a list of 2 components for this mixture

0   % mixture number
```

```
26   % dimensionality of the mixture
gm0  % the name of the Gaussian mixture.
3    % the number of components in this mixture
mx1_dpmf    % the name of the DPMF containing the
            % component responsibilities
            % the dimensionality of this DPMF must have value 3
gc_2 gc_3 gc_4 % a list of 3 components for this mixture
```

A Gaussian mixture definition must be given a dimensionality, and that dimensionality must match that of all of its corresponding components (and therefore which must match all of the corresponding means and covariance matrices).

4.5 Dlink matrices and structures

Dlink matrices and dlink structures are the one place where true graphical model structure is specified outside of the normal textual structural file. The structure that dlinks determine are those over observation vectors. Because is is often the case that such structure will switch for different values of hidden variables, that the structure will be automatically learned in some way, and that the amount of information contained in observation structures is quite large, it was decided that such structure should be kept out of the main structure file and specified using normal GMTK objects. Let us start with with dlink structures.

Dlink structures specify the directed dependencies that are to exist between the individual elements of vector observation vectors. Dlink structures do not specify the implementation of these dependencies, and state only that such dependencies might exist in some way.

Given a particular element of an observation vector, a dependency might exist from a parent that exists either 1) in the past of the current element, 2) in the same frame as the current element, and 3) in the future relative to the current element. This is depicted in Figure 4.1. These dependencies can co-exist with each other, so that a particular element might have multiple parents in the past, present or future.

There are a number of features of GMTKs dlink structures. First, each individual element of an observation vector can have its *own set of dependencies* which can be to any set of parents in the past, present, or future. Note, however, that an element can not be its own parent (Bayesian networks do not allow for directed cycle, and this undefined dependency would be such a cycle). Note also that a given element can not specify the same parent twice, as that would indicate a double edge, also not allowed in a Bayesian network. Second, dependency patterns can be *sparse* since there is no restriction on the set or number of parents an element can have. Third, the dependency patterns can switch as a function of a hidden parent. When a hidden variable which is a parent of an observation vector changes value, the corresponding Gaussian mixture will also (potentially) change. The Gaussian mixture can be set up so that the pattern of dependencies changes (switches) for each hidden variable value.

The dlink structure facility is quite flexible, and it allows GMTK to support Gaussians that have either full, banded diagonal, and factored sparse covariance matrices. It also allows multi-stream processing and cross-stream dependency modeling. More generally, it allows the representation of buried Markov models [9]. Let us start with a simple example first.

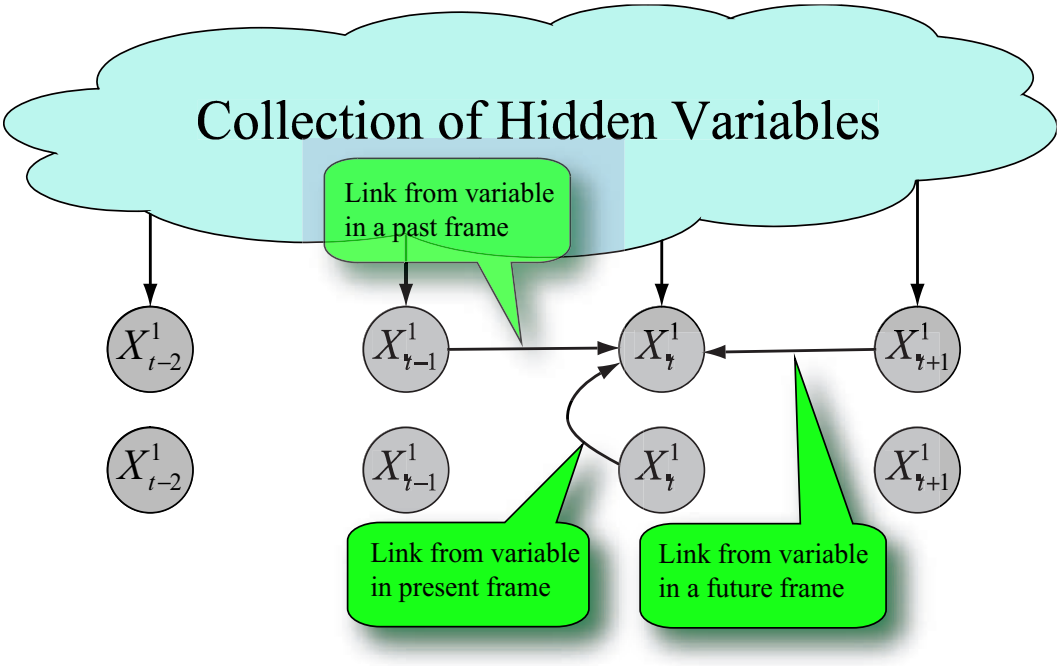

Figure 4.1: This figure shows a particular element of the current observation vector at time t can have a dependency either into the past, the present, or the future.

4.5.1 Full-covariance Gaussians

In Section TO-BE-INCLUDED (also given in [6]), it was shown how either a Gaussian can be represented by either a directed or an undirected graphical model. GMTK represents all Gaussians as directed graphical model. Repeating here part of the derivation given in Section **??**, the inverse covariance matrix $K = \Sigma^{-1}$ of a Gaussian can be Cholesky factored as follows as $K = R'R$, where R is upper triangular, $D^{1/2} = \text{diag}(R)$ is the diagonal portion of R, and $R = D^{1/2}U$. A Gaussian density can therefore be represented as:

$$p(x) = (2\pi)^{-d/2}|D|^{1/2}e^{-\frac{1}{2}(x-\mu)'U'DU(x-\mu)}$$

the exponent of which can further be represented as:

$$(x-\mu)'U'DU(x-\mu) = (x - Bx - \tilde{\mu})'D(x - Bx - \tilde{\mu})$$

where $U = I - B$, I is the identity matrix, B is an upper triangular matrix with zeros along the diagonal. Therefore, a full covariance Gaussian can be specified by giving 1) a mean, 2) a diagonal covariance component, and 3) a B matrix. The B matrix contains the coefficients of the linear dependencies of a directed graphical view of a Gaussian. In particular, the i^{th} row of B specifies the regression coefficients for a Gaussian when it is factorized according to the chain rule:

$$p(x) = \prod_{i=1}^{d} p(x_i|x_{i+1:D}) \tag{4.1}$$

where d is the dimensionality of the Gaussian.

A dlink structure given so that it specifies parents for individual elements of a feature vector to come from the same vector (the present) can therefore be used to determine the structure of a B

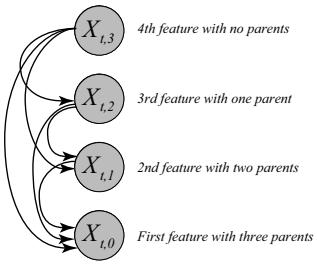

Figure 4.2: Figure showing one possible structure for the B matrix. In this case, we are defining a directed model over the individual features of a Gaussian. The first feature (feature number 0, or $X_{t,0}$) has three parents, $X_{t,1}$, $X_{t,2}$, and $X_{t,3}$. This given by the first structure specification line in the dlink structure file, which states `3 0 3 0 2 0 1`, meaning that there are 3 parents, the first parent has time lag 0 (so the current frame) and is to parent 3, the second parent has time lag 0 and is to parent 2, and the third parent has time lag 0 and is to parent 1. The specification of the parents for features 1, 2, and 3 are specified in a similar way. Note that to satisfy the chain rule of probability, care must be taken to ensure that no circularities are created in the dlink structure specification. For example, feature 3 in the figure has no parents, but if it did a circularity would exist.

matrix. Moreover, since the structure can be sparse or banded, GMTK supports full, sparse, and banded diagonal matrices.

Lets go over a few examples. Suppose we wish to produce a $d = 4$ dimensional full-covariance Gaussian. Furthermore, let us use the Gaussian factorization given in Equation **??**. This means that element i has as parents all other elements in the observation vector at the same time that have higher element numbers, i.e., $i + 1, i + 2, \ldots, d$. The dlink structure for this Gaussian is depicted in Figure 4.2. To specify the dlink structure for this Gaussian, one would specify (inline within a master file) the following:

```
DLINK_IN_FILE inline 1
0 % first dlink structure
dlink_str0 % name of the dlink structure 'dlink0'.
4       % Number of features for which this dlink structure
        % refers to. This must correspond to the dimensionality
        % of the Gaussian.
% The parents for feature element 0, which has 3 parents.
% The numbers consist of:
% <num links> <1st time lag> <1st offset> <2nd time lag> <2nd offset> ...
3 0 3 0 2 0 1
% Next, the dlinks for feature 1
2 0 3 0 2
% Next, the dlinks for feature 2
1 0 3
% Finally, the dlinks for feature 3, and there are no dependencies.
0
```

The dlink structure contains the following. First, the dlink structure name `dlink0`. Next comes an integer indicating the number of features that this dlink can be used with. When a

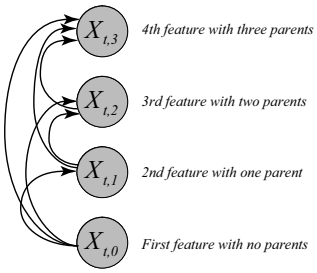

Figure 4.3: Figure showing another possible structure for the B matrix.

Gaussian component uses this dlink structure, the dimensionality of the Gaussian must correspond to the number of features in the dlink structure. Therefore, we could call this number d in general. Next comes the specification of the structure itself. This consists of d lists of parents, one for each feature starting at the lowest feature of the dlink structure (e.g., feature 0) and ending at the highest feature number (e.g., feature $d-1$).

In the example above, feature element 0 has three parents, all from relative time lag 0 (meaning all parents come from the same frame). The three parents consists of feature elements 3, 2, and 1 respectively (see Figure 4.2). Feature element 1 has two parents in the same frame, features 3 and 2. Feature 2 has one parent, feature 3. Finally, the fourth feature, feature 3, has no parents.

The dlink structure is flexible, so different chain rule factorizations can be specified just as easily. For example, given the chain rule factorization of the Gaussian as follows:

$$p(x) = \prod_{i=1}^{d} p(x_i | x_{1:i-1}) \tag{4.2}$$

This can be specified as follows.

```
DLINK_IN_FILE inline 1
0 % first dlink structure
dlink_str1
4       % Number of features.
0
1 0 0
2 0 0 0 1
3 0 0 0 1 0 2
```

This says that feature 0 has no parents, feature 1 has one parent (at element 0), feature 2 has two parents (elements 0 and 1) and lastly feature 3 has three parents (elements 0, 1, and 2). This is depicted in Figure 4.3. Note that there are many other ways of factorizing a Gaussian and a dlink structure can represent all of them (there are $d!$ possible factorizations). Regardless of the order of factorization, a full-covariance Gaussian can be specified.

Once a dlink structure for a particular ordering of the factorization of a B matrix is specified, it is necessary to also specify the actual regression coefficients of the B matrix for the Gaussian, and this is done using a dlink matrix. A dlink matrix has a coefficient for each dependency specified in a dlink structure.

```
DLINK_MAT_IN_FILE inline 1
0 % first dlink matrix
```

```
dlink_mat0 % name of the dlink matrix.
dlink_str0 % Name of the dlink structure to be used with
           % this dlink matrix.
4          % number of features, must match dlink structure
% the regression coefficients for the dlink structure.
% For each vector element, an integer giving
% the number of coefficents and the coefficients themselves
% must be specified.
3  0.1 0.2 0.3
2  0.4 0.5
1  0.6
0
```

The dlink matrix first consists of its name `dlink_mat0`. Next, the name `dlink_str0` of the dlink structure to be used with this matrix is given. After that, the number of features is given. This number must match that of the number of features of the dlink structure. Finally, the list of coefficients is given, one for each feature element, meaning one each for the corresponding element given in the dlink structure.

We are at last ready to specify a full-covariance Gaussian. We will create a full-covariance Gaussian using the Cholesky factorization describe above, but we will use upper-triangular B matrices that are dense in the upper-triangular portion. This produces a full-covariance Gaussian as described in Section TO-BE-INCLUDED (also see [6]).

Lets use the factorization given in Equation 4.1 and specify the dlink structure, dlink matrix, a mean vector, a covariance vector, and a Gaussian component for a $d = 12$ dimensional full-covariance Gaussian component. What follows is the complete annotated specification.

```
% specify a 12-D zero mean vector
MEAN_IN_FILE inline 1 % 1 mean vector
0 % mean 1
mean0 12 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
COVAR_IN_FILE inline 1

% specify a 12-D unity diagonal covariance matrix
COVAR_IN_FILE inline 1 % 1 diagonal covariance matrix
0 % covar 1
covar0 12 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0

% specify a dlink structure corresponding to element
% i having parents all elements with index greater than i.
DLINK_IN_FILE inline 1
0      % dlink structurr 1
dlink0 % name of dlink structure = 'dlink0'
12     % dimensionality
11 0 11 0 10 0 9 0 8 0 7 0 6 0 5 0 4 0 3 0 2 0 1
10 0 11 0 10 0 9 0 8 0 7 0 6 0 5 0 4 0 3 0 2
9  0 11 0 10 0 9 0 8 0 7 0 6 0 5 0 4 0 3
```

```
8  0 11 0 10 0 9 0 8 0 7 0 6 0 5 0 4
7  0 11 0 10 0 9 0 8 0 7 0 6 0 5
6  0 11 0 10 0 9 0 8 0 7 0 6
5  0 11 0 10 0 9 0 8 0 7
4  0 11 0 10 0 9 0 8
3  0 11 0 10 0 9
2  0 11 0 10
1  0 11
0

% The dlink matrix corresponding to dlink structure dlink0
DLINK_MAT_IN_FILE inline 1
0      % dlink matrix 1
dlink_mat0 % name of dlink matrix
dlink0     % name of dlink structure to use
12         % dimensionality
% regression coefficients, all zeros for now.
11 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
10 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
9  0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
8  0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
7  0.0 0.0 0.0 0.0 0.0 0.0 0.0
6  0.0 0.0 0.0 0.0 0.0 0.0
5  0.0 0.0 0.0 0.0 0.0
4  0.0 0.0 0.0 0.0
3  0.0 0.0 0.0
2  0.0 0.0
1  0.0
0

% The gaussian component using the above.
GC_IN_FILE inline 1
0  % first GC
12 % dimensionality of the GC
1  % the 'type' of the Gaussian component. 1 = use dlink structure
gc_0 % the name of the component, here 'gc_0'
mean0 covar0 dlink_mat0 % the mean, variance and dlink matrix to use.
```

The above defines a component named `gc_0`. Note that the type of the component in this case is 1 (rather than 0 as was done in Section 4.4.2). Type 1 indicates that a mean, covariance, *and* dlink matrix should follow. Note that factorizations other than Equation 4.1 can be produced just as easily, by specifying a different dlink structure and matrix. Moreover, it is possible to produce a mixture of Gaussians that have components with using different factorizations. This feature will become more interesting in the next section.

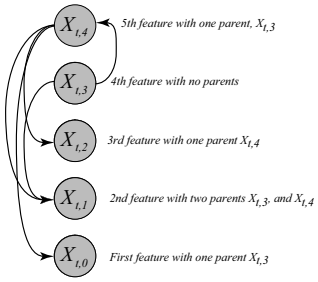

Figure 4.4: A directed model for a sparse 5-dimensional Gaussian.

4.5.2 Banded Diagonal and/or Sparse Factored Inverse Covariance Matrices

Dlink matrices and structures can do more than specify full-covariance matrices. They may also be used to specify a variety of different directed structures for the Gaussian densities. This can lead to a wide variety of possible sparse directed Gaussian structures.

The way this is done is similar to the way that full covariance matrix Gaussians are formed, but rather than giving the full set of possible parents for a given Gaussian factorization, instead only a subset is given. For example, as stated above, any density function (including a Gaussian) can be factorized according to the chain rule of probability. Suppose the dimensionality of the Gaussian is d, and that $\pi(i), i = 1 \ldots d$ is an arbitrary permutation of the integers from 1 to d inclusive. The chain rule states the following:

$$p(x) = \prod_{i=1}^{d} p(x_{\pi(i)}|x_{\pi(1:i-1)}) \tag{4.3}$$

where $\pi(1 : j)$ represents the first j integers in the permutation, i.e., $\pi(1 : j) = \{\pi(1), \pi(2), \ldots, \pi(j)\}$. By making conditional independence assumptions about the distribution, we may change the above factorization as follows.

$$p(x) = \prod_{i=1}^{d} p(x_{\pi(i)}|x_{\text{pa}(i)}) \tag{4.4}$$

where $\text{pa}(i)$ denote the parents of element i, and since it is a directed model, we have that the parents are a subset, meaning that $\text{pa}(i) \subseteq \pi(1 : i - 1)$. This factorization corresponds to the set of independence statements for each i

$$X_{\pi(i)} \perp\!\!\!\perp (X_{\pi(1:i-1)} \setminus X_{\text{pa}(i)}) | X_{\text{pa}(i)}.$$

Note again that the permutation π can be arbitrary. GMTK supports all possible permutations in its representation of sparse Gaussians. Moreover, in light Equation **??** in Section **??**, the sparseness corresponds to zeros in the B matrix. Therefore, such a sparse pattern can be set up using the dlink structure and matrix mechanism mentioned above. Lets run through a few examples.

Suppose that we have a 5-dimensional Gaussian distribution, and we wish to represent the sparse directed graph given in Figure 4.4. We need only to create a dlink structure and a corresponding matrix for this purpose, as follows:

```
DLINK_IN_FILE inline 1
0 % first dlink structure
```

```
sparse_dlink0 % name
5       % Number of features.
% The parents for feature element 0, 1 parent, number 4
% The numbers consist of:
% <num links> <1st time lag> <1st offset> <2nd time lag> <2nd offset> ...
1 0 4
% Next, the dlinks for feature 1, two parents, 3 and 4.
2 0 3 0 4
% Next, the dlinks for feature 2, one parent, feature 4
1 0 4
% Next, dlinks for feature 3, no dependencies.
0
% Finally, dlinks for feature 4, one parent, feature 3
1 0 3

% Next, we have a dlink matrix for the above structure.
DLINK_MAT_IN_FILE inline 1
0       % dlink matrix 1
sparse_dlink0 % name of dlink matrix
sparse_dlinkmat0     % name of dlink structure to use
5          % dimensionality
% regression coefficients, all zeros for now.
1  0.4
2 -0.4 1.3
1  1.2 1.0
0
1 -2.4
```

Note that in the above structure, there are dependencies not only from lower numbered elements to higher numbered ones (namely, element 0 and 2 both have element 4 as a parent, element 2 has elements 3 and 4 as parents) but an element can have a lower numbered parent as well (namely, element 4 has element 3 as a parent). Note that this is OK since the structure does not specify a directed cycle, meaning the structure is a directed acyclic graph (DAG). To see this, place $X_{t,3}$ at the top of the DAG, where it has children $X_{t,4}$ and $X_{t,1}$. $X_{t,4}$ has three children $X_{t,2}$, $X_{t,1}$, and $X_{t,0}$.

In general, any combination of child and set of parents is a valid configuration in GMTK, except that a child may not be a parent of itself (note that specifying self as parent will cause GMTK to signal an error). Also, note that it is important to produce a structure that corresponds to a valid permutation and therefore factorization as specified in Equation 4.3. Note that using the representation for dlinks given above it is not only possible to specify all permutations, but it is also possible to specify graphs that have directed cycles, as in the following dlink structure over a 3-dimensional Gaussian.

```
DLINK_IN_FILE inline 1 0 circularity_dlink 3
1 0 1
1 0 2
1 0 0
```

The above structure states that that element 1 is a parent of element 0, element 2 is a parent of element 1, and element 0 is a parent of element 2. Such a model *does not* correspond to a valid chain rule factorization for any permutation, and is therefore not a DAG, but is nevertheless allowed in the current version of GMTK. Care must be taken, therefore, to ensure that your sparse structures do not contain such a circularity.[1]

4.5.3 Sparse Global B Matrix

It is possible with GMTK to set up a structure to learn a sparse global B matrix. Having and learning the parameters for such a B matrix is similar to learning a global scaling and rotation matrix that is applied to all feature vectors, similar to a matrix transform (such as a discrete cosine transform) which is applied to the feature vectors before they are used in the ASR system. In this case, however, the transform can be learned in a maximum likelihood setting.

4.5.4 Buried Markov Models with Linear Dependencies

Using the above structures, it is just as easy to specify structures corresponding to buried Markov models ([9]), but where the edge implementations between observation elements are linear.

In all of the above dlink structures, you might have noticed the zero that comes in between element indices. For example, in a line `3 0 3 0 2 0 1` defining a dlink structure, the zeros specify that the parents come from the the current frame. Specifically, these zeros are the current relative frame offset from the current frame at which to obtain the parent elements. If all of the offsets are always zero, then this will correspond to a certain permutation of a Gaussian factorization [2]. If, on the other hand, the offset is negative then the parent will come from the past relative to the current frame, and will implement a conditional Gaussian distribution, where the mean of the Gaussian is a function of the parents. If the offset is positive, the parents will come from the future relative to the current frame. There may be any mix between negative, zero, and positive temporal offsets.

Using dlinks structures and matrices, it is possible to derive BMMs where the structure switches as a function of one of the discrete parents. The parent simply needs to specify a Gaussian mixture that uses a different set of dependency links.

Lets consider the following BMM structure specified in Figure 4.5. This BMM uses 5-dimensional feature vectors, and has the sparse pattern as shown. Also shown is the fact that the BMM changes its structure depending on the value of the hidden variable at the time (lets assume that the hidden variable is binary for now). When the hidden variable is zero ($Q_t = 0$), we have the first sparse pattern on the left, and when it is one ($Q_t = 1$), we have the second pattern on the right.

These dlink structure for this case can be specified as follows. This requires several objects. First, two dlink structures are defined, one for each hidden variable value. Next, two dlink matrices are defined, one for each dlink structure. In general, that multiple dlink matrices can share the same dlink structure, which would correspond not to a switching structure, but rather switching parameter values. For illustrative purposes, in this example each dlink matrix uses a distinct dlink structure for each dlink matrix (and corresponding hidden variable value).

```
DLINK_IN_FILE inline 2
0 % first dlink structure
```

[1] The authors would be interested to hear if anyone obtains useful results using such circularities, which are clearly invalid probabilistic structures. My guess is that this will result in numerical oddities, such as zero valued variances.

[2] Again, assuming it is a valid factorization, see the previous section on sparse covariances

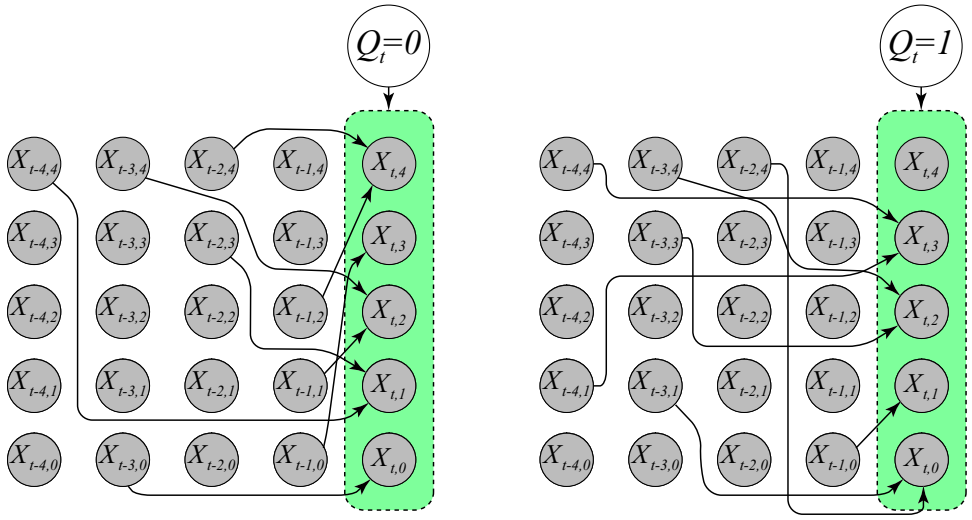

Figure 4.5: A simple BMM over 5-dimensional feature vectors. Note that the structure changes (switches) depending on the value of the hidden variable Q_t. Specifically, when the hidden variable $Q_t = 0$ the structure on the left (and its parameters) are active. when $Q_t = 1$ the structure on the right is active. The dlink structures and matrices for this graph is specified in the text.

```
bmm_struc_0 % name
5       % Number of features.
% note that the offsets are no longer zero.
1 -3 0
2 -2 3 -4 4
2 -3 4 -1 1
1 -1 0
2 -1 2 -2 4

1 % next dlink structure
bmm_struc_1 % name
5       % Number of features.
2 -3 1 -2 4
1 -1 0
2 -3 3 -3 4
2 -4 1 -4 4
0

% Next, we have a dlink matrcies for the above structures.
DLINK_MAT_IN_FILE inline 2
0       % dlink matrix 1
bmm_mat_0   % name
bmm_struc_0 % name of dlink structure to use
5          % dimensionality
% regression coefficients, all zeros for now.
1 0.3
```

```
2 0.4 -0.4
2 0.1  0.0
1 1.0
2 1.3 -1.4

1      % dlink matrix 2
bmm_mat_1    % name
bmm_struc_1 % name of dlink structure to use
5            % dimensionality
% regression coefficients, all zeros for now.
2 -2.0 4.0
1  1.0
2  1.0 2.0
2 -3.0
0
```

Now that the dlink structures and matrices are set up, it is possible to specify two Gaussian components which utilize these two structures (we assume that the mean and variance objects have been defined somewhere, but are not specified in what follows).

```
GC_IN_FILE inline 2

0  % first GC
5  % dimensionality of the GC
1  % the 'type' of the Gaussian component. 1 = use dlink
gc_0 % the name of the component, here 'gc_0'
mean_0 covar_0 bmm_mat_0 % the mean, variance, and bmm mat of this component

1  % second GC
5  % dimensionality of the GC
1  % the 'type'
gc_1 % name of this component, here 'gc_1'
mean_1 covar_1 bmm_mat_0 % the mean and variance of this component
```

It is also interesting to note that parents need not come only from the past, but may also come from the future (or any combination of past, present, and future) as the following simple dlink structure demonstrates (as also given in Figure 4.6).

```
DLINK_IN_FILE inline 1
0 % first dlink structure
bmm_struc_0 % name
5      % Number of features.
% note that the offsets are no longer zero.
3 -3 0 3 1 2 4
3 -2 3 -4 4 1 0
4 -3 4 -1 1 3 3 3 4
```

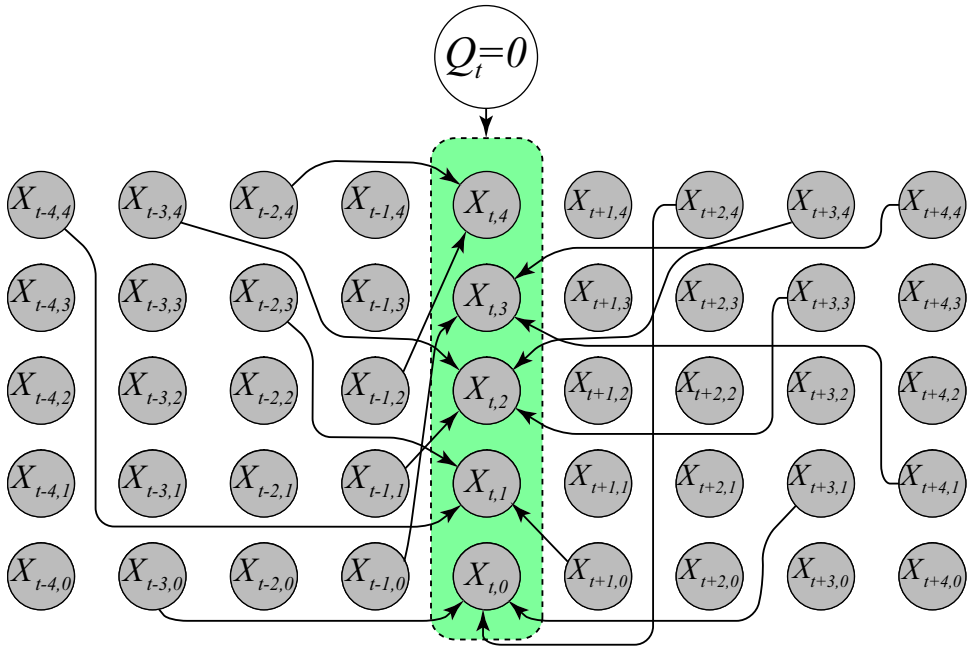

Figure 4.6: A BMM with parents both in the past and future relative to the the current frame.

```
3 -1 0 4 1 4 4
2 -1 2 -2 4
```

Note that for readability, it is of course always possible to reformat the above dlink structure specification. It is only white space that needs to separate the characters. For example:

```
DLINK_IN_FILE inline 1
0 % first dlink structure
bmm_struc_0 % name
5       % Number of features.
% note that the offsets are no longer zero.
3 -3 0
   3 1
   2 4
3 -2 3
  -4 4
   1 0
4 -3 4
  -1 1
   3 3
   3 4
3 -1 0
   4 1
   4 4
2 -1 2
  -2 4
```

Again, however, it is important to realize that when using structure with dependencies in the past, present, and future, the resulting graph might contain directed cycles. Care should therefore be taken to ensure that structures are not specified as such, or otherwise unexpected results may follow. Once again, GMTK does *not* signal this *a prior* as an error, and it is only after training might the results of such invalid structures be discovered. As mentioned above, the most likely result, however, will be numerical instabilities (e.g., IEEE floating point infinities, and/or messages about dividing by zero).

4.6 GMTK Parameter Sharing/Tying

GMTK supports a flexible array of options for parameter sharing and parameter tying. Any mean can be tied to any other mean (of the same dimension), any diagonal covariance may be tied to any other diagonal covariance (again of the same dimension), and any dlink matrix may be tied to any other dlink matrix (as long as the structures of the two dlink matrices are the same).

Sharing need not only be specified during testing, but may also be specified before training begins. GMTK uses a GEM algorithm for training in this, an algorithm that retains the convergence Guarantees that normal EM possess, as long as training is done in the right way. The GEM algorithm is fully described in [**?**].

Parameter sharing is specified very simply in GMTK. When an object in GMTK is defined that consists of several constituent sub-objects, each of those sub-objects is specified using the sub-object name. If two objects wish to share the same sub-object, they need only specify the same name. For example, if two Gaussian components (which each normally consist only of a mean and a diagonal covariance vector) wish to share the same mean, they need only specify the same mean name. For example,

```
GC_IN_FILE inline 2
0  % first GC
26 % dimensionality of the GC
0  % the 'type' of the Gaussian component. 0 = standard
gc_0 % the name of the component, here 'gc_0'
mean_0 covar_0 % the mean and variance of this component

1  % second GC
26 % dimensionality of the GC
0  % the 'type' of the Gaussian component. 0 = standard
gc_1 % the name of the component, here 'gc_0'
mean_0 covar_1 % the mean and variance of this component
```

In the example above, `gc_0` and `gc_1` have unique diagonal covariance matrices `covar_0` and `covar_1` respectively), but they share the same mean vector `mean_0`. When the Gaussian is evaluated, that same mean will be used. Moreover, even Gaussians of different types can share a mean vector. This implies that a Gaussian of type '0' (having just a mean and diagonal covariance) and a Gaussian of type '1' (having a mean vector, diagonal covariance, and B matrix) can share the same mean (or diagonal covariance as well).

Note that during training, GMTK figures out when an object is shared, and decides based on the particular type of sharing if an EM algorithm can be used, or if for that particular object, a GEM

algorithm must be run. This means that EM and GEM might be used simultaneously to train the set of parameters in GMTK, each will be used to match the appropriate parameter sharing. GMTK itself figures out when it should run EM and when it should run GEM based on the sharing pattern that is specified by the user.

The example above demonstrates only how mean vectors can be shared. GMTK however can share or tie many basic object in the same way. In particular, means, covariances, and dlink matrices can be shared in the same way simply by specifying the name of the sub-object so desired. When two or more different objects specify the same sub-object, then that sub-object is shared. Objects that can be shared this way include decision trees, named collections, dense probability mass functions (DPMFs), sparse probability mass functions (SPMFs), dlink structures, and Gaussian components (i.e., different Gaussian mixtures can share the same Gaussian component). Lets give a few examples.

First, here is an example where a decision tree, named `myDT`, is shared between two different deterministic CPTs. This can be done as follows:

```
DETERMINISTIC_CPT_IN_FILE inline 2
0        % first DETERMINISTIC_CPT
detcpt1  % name of the CPT
1    % num parents of the corresponding random variable
2 3  % cardinalities. cardinality of parent = 2, of self = 3
myDT % The name of an existing DT to implement the deterministic
     % mapping

1       % second DETERMINISTIC_CPT
detcpt2 % name of the CPT
1       % num parents of the corresponding random variable
10 20   % cardinalities. cardinality of parent = 2, of self = 3
myDT % The name of an existing DT to implement the deterministic
     % mapping
```

Note that the only difference between the two CPTs is that they have different cardinalities. The cardinalities therefore are associated with a CPT, not with a decision tree. The same decision tree can be used for any purpose where it might be useful. In the example above, the two deterministic CPTs implement the same function, but the first and the second requires a parent cardinality of 10 (and self cardinality of 20).

As our next example, suppose it is desired to share a dlink structure across multiple dlink matrices (this, for example, can be used to set of a system with a globally shared B matrix, as described above). This can be set up as follows.

```
% specify a dlink structure corresponding to element
% i having parents all elements with index greater than i.
DLINK_IN_FILE inline 1
0      % dlink structure 1
dlink0 % name
3     % dimensionality
2  0 2 0 1
1  0 2
```

```
0

% The dlink matrix corresponding to dlink structure dlink0
DLINK_MAT_IN_FILE inline 2
0          % dlink matrix 1
dlink_mat0 % name of dlink matrix
dlink0     % name of shared dlink structure to use
3          % dimensionality
% regression coefficients, all zeros for now.
2 0.0 0.0
1 0.0
0

1          % dlink matrix 1
dlink_mat1 % name of dlink matrix
dlink0     % name of shared dlink structure.
3          % dimensionality
% regression coefficients, all zeros for now.
2 0.0 0.0
1 0.0
0
```

In this example, the two dlink matrices `dlink_mat0` and `dlink_mat1` share the same dlink structure `dlink0`, simply by specifying the name `dlink0` in both instances.

Mixtures of Gaussians and the various forms of CPT objects (dense, sparse, and deterministic) can also be shared, but are done so in a different way, namely via the structure file and/or the current name collection object being used.

To share a CPT object of some sort, the name of the CPT is specified whenever it is needed. The following is an example:

```
variable : var1 {
   type: discrete hidden cardinality 2 ;
   conditionalparents :
         foo(0) using DenseCPT("sharedCPT");
}
variable : var2 {
   type: discrete hidden cardinality 2 ;
   conditionalparents :
         bar(0) using DenseCPT("sharedCPT");
}
```

In this case, the dense CPT named `sharedCPT` is specified twice by both random variables `var1` and `var2`. Note that the cardinalities of the variables that share a CPT and the cardinalities of the corresponding parents must be equivalent to each other, and must be the same as the corresponding CPT. For example, assuming that `foo` has cardinality 3, then `bar` must also have cardinality 3, and the shared CPT named `sharedCPT` must be a 3×2 table.

Mixture of Gaussian objects can also be shared between multiple sets of continuous observed random variables. Again, this is done via a combination of the structure file and the collection objects. Consider the following example, first a few definitions of collection and decision tree objects which is then followed by a structure file segment.

```
NAME_COLLECTION_IN_FILE inline 3
0     % first
col1  % The name of the collection, 'col1'
3     % The collection length, 3 entries.
gm1
gm2
gm3
1     % second
col2  % The name of the collection, 'col2'
3     % The collection length, 3 entries.
gm4
gm3
gm2
2     % third
col3  % The name of the collection, 'col3'
3     % The collection length, 3 entries.
gm3
gm5
gm4

DT_IN_FILE inline 3
0 % first
map1     % name of decision tree, 'map1'
1        % only one parent.
-1 (p0)  % just copy value of parent
1 % second
map2     % name of decision tree, 'map2'
1        % only one parent.
-1 (p0)  % just copy value of parent
2 % third
map3     % name of decision tree, 'map3'
1        % only one parent.
-1 (p0)  % just copy value of parent
```

```
   variable : obs1  {
      type: continuous observed 0:12 ;
      conditionalparents: foo(0) using mixGaussian
        collection("col1")
        mapping("map1");
   }
   variable : obs2  {
```

```
    type: continuous observed 0:12 ;
    conditionalparents: bar(0) using mixGaussian
      collection("col2")
      mapping("map2");
}
variable : obs3  {
    type: continuous observed 13:25 ;
    conditionalparents: baz(0) using mixGaussian
      collection("col3")
      mapping("map3");
}
```

In this case there are three observed variables `obs1`, `obs2`, and `obs3`, each using its corresponding collection objects and mapping objects.

The first thing to note is that for continuous observation variables to share Gaussians, the (necessarily vector) variables must have the same dimensionality (vector length). In the case above, each of the variables have dimensionality 13, feature elements 0 through 12 for `obs1` and `obs2`, and feature elements 13 through 25 for `obs3`.

Each of the observation vectors use both a mapping decision tree and a collection object. The decision trees `map1`, `map2`, and `map3` map from the parent random variables, `foo`, `bar`, and `baz` respectively to positions within the collection objects (`col1`, `col2`, `col3` respectively).

Note that any of the variables should share either the collections or the maps, and in the example above, the definition of the mapping decision trees are in fact the same. For simplicity, the example shows each variable with its own collection and mapping objects.

As defined above, each of the collection objects have length three. That means that the decision tree objects should never have a leaf node which has value greater than two (the values can only be between 0 and 2). As defined above, however, the decision trees copy the parent random variable value, which means that the corresponding random variables `foo`, `bar`, and `baz` should never have a value that is greater than 2. Note that in GMTK, there can be a value greater than 2 **only** if all such values have zero probability.

The three collection objects define lists of names of Gaussian mixtures, `gm1` through `gm5`. From the file above, it means, for example, that `col1`'s first (i.e., zero position) mixture is `gm1`, `col2`'s first mixture is `gm4`, and `col3`'s first mixture is `gm5`. The example above, for example, means that when `obs1`'s parent `foo` has value 2, it uses Gaussian mixture `gm3`. This same mixture is also used for `obs2` when its parent `bar` has value 1, and used for `obs3` when its parent `baz` has value 0. This, therefore, is the general way that Gaussian mixtures can be shared.

Note that there are actually two ways that mixtures could be shared, either via the collection object, or via the decision tree. In the above example, all of the sharing was done via the collection objects since the decision trees were identical and simply mapped the parent value to the leaf value. Other decision trees could be used, however, which maps collections of values of parent variables to the same position in the collection object.

Lastly, note that variable `obs3` shares Gaussians with variables `obs1` and `obs2`, even though `obs3` corresponds to a different feature range. This is allowed in GMTK, meaning that it is possible to share the same Gaussian distribution over a different and disjoint set of features. There can even be overlap between the two ranges, if the ranges as given overlap. It is moreover possible to share portions of a Gaussian, for example, by having the Gaussian components share the same means (as specified in Section xxx). In sum, these abilities give GMTK a great deal of flexibility in how it is able to share Gaussian densities or portions of Gaussian densities.

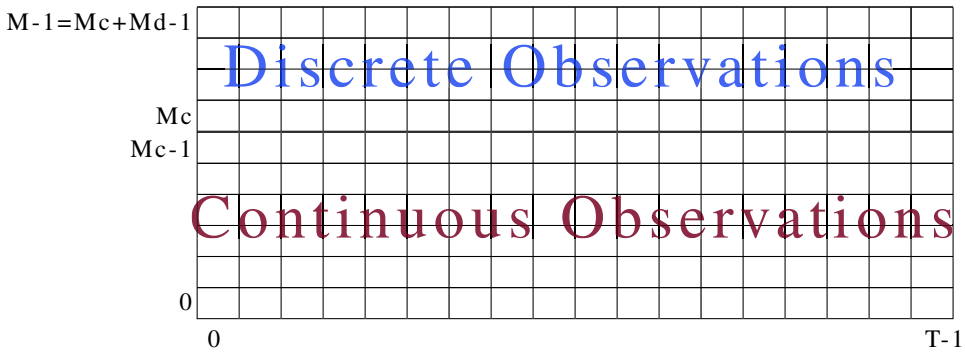

Figure 4.7: The Global Observation Matrix

4.6.1 GEM training and parameter Sharing/Tying

GMTK supports both EM training and GEM training. The underlying equations for GEM training is described in Section TO-BE-WRITTEN. It should be noted here, however, that the GMTK general training program determines when either EM or GEM training is appropriate, depending on the degree and type of sharing that occurs. In fact, it might be that GEM and EM training are running internally at the same time, for different parameter values.

4.7 The Global Observation Matrix

GMTK uses a global observation matrix, which is read in for all files. The global observation matrix contains, say, T frames (for a T frame utterance) and some number M of features per frame. It can contain both continuous M_c and discrete M_d features, where $M = M_c + M_d$. The lower portion of the matrix contains only continuous features and the upper portion contains only discrete features. The matrix might have only continuous ($M_d = 0$) or only discrete features ($M_c = 0$). Figure 4.7 shows an example of an $M \times T$ sized matrix, and how it is organized.

Feature range specifications for observed random variables in a structure file determines the range in the global observation to obtain the observation values for the current frame. For example, once the template network is unrolled so that it is T frames long, each observed random variable at, say, frame t obtains its values from the elements within column t of the matrix. It is important therefore to specify feature ranges that map to the appropriate type of feature (i.e., a Gaussian should not specify a range that includes discrete features, and a discrete variable should not specify a continuous feature). If this occurs, GMTK will produce an error message.

4.7.1 Dlink Structures and the Global Observation Matrix

Another situation in which portions of columns of the global observation matrix is specified is when a dlink structure (see Section 4.5) is used. Consider the following example:

```
DLINK_IN_FILE inline 1
0
dlink_str0
4      % Number of features for which this dlink structure
3 0 3 0 2 0 1
% Next, the dlinks for feature 1
```

```
2 0 3 0 2
% Next, the dlinks for feature 2
1 0 3
% Finally, the dlinks for feature 3, and there are no dependencies.
0
```

For the specification of *feature* position elements in a dlink structure, the dlink structure specifies **absolute** positions within the global observation matrix rather than positions relative to the Gaussian that is using the Gaussian mixture. In other words, the dlink structure specifies the same dependencies regardless of what Gaussian uses it. For example, in the example above, the first feature of the dlink structure requires absolute feature position 3, 2, and 1 in the current frame of the observation matrix. If a Gaussian using this dlink structure was defined over features 0 through 3 (i.e., the structure file specifies the range 0 through 3), then this would correspond to a full covariance Gaussian, say $p(x)$ where x is the 4-dimensional vector corresponding to features 0 through 3 . If, on the other hand, a Gaussian using this dlink structure was defined over features 4 through 7, this would correspond to a linear conditional Gaussian, say $p(y|x)$ where y is a 4-dimensional vector over features 4 through 7 of the current frame, and x is again features 0 through 3 of the current frame.

Given the above, when specifying non-diagonal covariance Gaussians, care should be taken to ensure that the dlink structure used corresponds to the desired conditioning set. On the other hand, with this ability, it is easy to create a variety of conditional multi-stream Gaussian models, say $p(y|x)$ where y comes from one stream, and x comes from another stream. This is particularly easy using multiple observation files described below.

Note that for temporal elements, the dlink structure specifies **relative** rather than absolute positions within the observation matrix. For example, a dlink structure line of the form

```
3 0 3 0 2 0 1
```

would select feature elements (i.e., parents) 3, 2, and 1 in the current frame (the three zeros), but in

```
3 -1 3 -2 2 -3 1
```

this selects parent 3 in the previous frame (-1), parent 2 two frames in the past, and parent 1 three frames in the past.

4.7.2 Multiple Files and the Global Observation Matrix

The global observation matrix is formed from the vertical concatenation one or more observation files. There is a specific command line syntax to use, and is described in the section on the programs (see below).

Each observation file used must contain the same number (say T) of frames, but the files may have as many features per frame (say M^i) as is desired. Moreover, each observation file may have any number of continuous M_c and/or discrete M_d features, where $M = M_c + M_d$. Supposing that some number of files are concatenated together, where the i'th file has M^i features, the resulting virtual internal global observation matrix will have $\sum_i M^i$ features. The continuous and discrete features are reorganized, however (see Figure **??**). In particular, if M_c^i (resp. M_c^i) is the number of continuous (resp. M_d^i) features in the i'th file, then once the files are concatenated together,

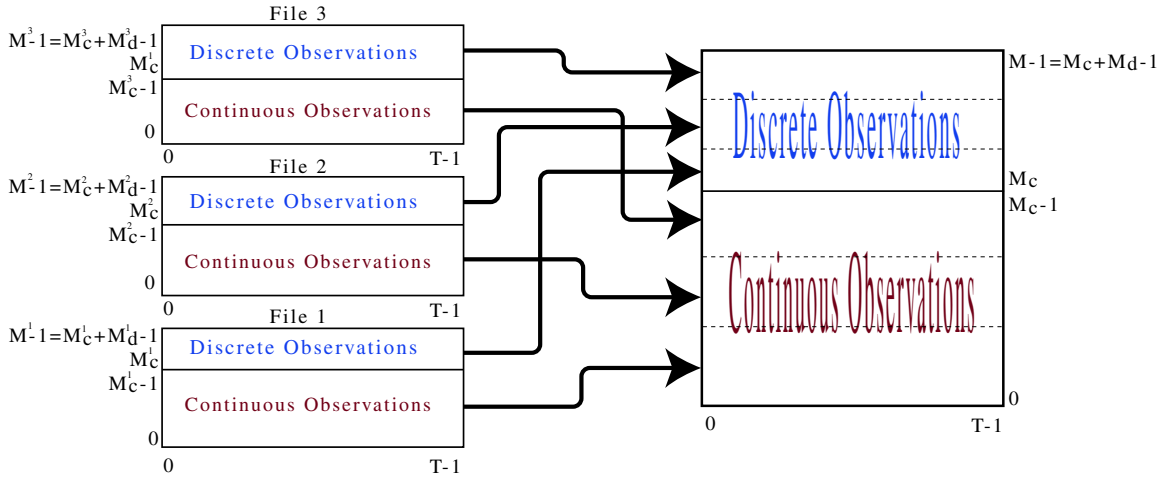

Figure 4.8: The combination of three feature files on the left consisting each of both continuous and discrete observations. On the right is the virtual global observation matrix, combined from the three feature files so that the continuous observations come first followed by the discrete observations.

the first $M_c = \sum_i M_c^i$ features (i.e., features 0 through $M_c - 1$) are continuous, and the remaining $M_d = \sum_i M_d^i$ features are discrete (i.e., features M_c through $M_c + M_d - 1$). This way, it is always the case that the lower indexed features are continuous and the higher indexed features are discrete.

Note that selection of the feature stream files, and the orders that the feature streams are combined, can be changed on the fly using GMTK's command line arguments (see below). GMTK structure files and dlink structures, however, specify absolute locations in the virtual observation matrix. Therefore, if an experiment exists and a new file to merge is added at the beginning, the range of the feature values will change. For example, suppose there are two observation files at the moment, the first (file A) has 5 features and the second (file B) has 3 features. The global observation matrix will have 8 features, and the first five (features 0-4) will correspond to file A, and the last 3 (features 5-7) will correspond to file B. If another file (file C) is used as the first file (and suppose file C has two features) so that the file order used is now C A B, that will effectively shift up by 2 the indices of the features corresponding to file A and file B. This could significantly change the behavior of a program, especially if one is using Gaussians that have already been trained using just the file A and file B features.

In general, it is better to have any new feature files added at the end rather than the beginning (e.g., order A B C above) so that the ranges in the structures and dlinks that have already been specified will correspond to the same features and therefore still be valid.

Chapter 5

The Main GMTK Programs

GMTK consists of a number of different programs that allow you to do anything from train the parameters of a graphical model based system, to convert parameters from ASCII to binary format.

5.1 Integer range specifications

Integer ranges are used for a number of purposes in GMTK programs: decision tree range specifications, ranges of features to include in the global observation matrix, the set of utterances to include in training. Integer ranges are really just a way of specifying a general set of integers. GMTK uses a particular syntax for specifying these sets as follows:

In general, an integer range specifies a set of integers in the range 0 through $N-1$, where N is the total number of possible integers. In some cases, N is known and there are shortcut range specifications that can be used (e.g., the number of utterances, the number of accumulator files, etc.). In other cases, N is unknown at the time the range is given (e.g,. a decision tree range, since a given decision tree can be used with different random variables having different cardinalities). In the latter case, N is a very large value (larger than any cardinality).

An integer range specification consists of one of of the following forms: `b` indicates element `b` with 0 the least and $N-1$ the greatest possible value; `b-e` or `b:e` inclusive beginning at `b` ending at `e`; `b:s:e` inclusive from `b` striding by `s` and ending no more than `e`. Either `b` or `e` can be omitted specifying an implicit 0 or $N-1$ (e.g., `b-` indicates an inclusive range starting at `b` ending at $N-1$). Either `b` or `e` can be preceded by a `^` character indicating the value $N-n-1$ (i.e., `^0` is the last element $N-1$, `^2` is $N-1-2$, etc.). A full range specification must indicate a sorted list of elements where each element is specified at most once. A range may also start with the `/` character indicating a negated (i.e., complemented) range, i.e., all elements between zero and one less than N will be selected except for those that match the range specification following the `/` character.

For example, if $N = 20$ the range `0,2,4-8,9,11:4:^0` corresponds to the set `0,2,4,5,6,7,8,9,11,15,19`, the range `1:7:14,10-12,^3-^0` indicates `1,8,10,11,12,16,17,18,19`, and the range `/3,5,8-17` indicates `1,2,4,6,7,18,19`. Also, `0:2:` specifies the even elements and `1:2:` specifies the odd ones. A range can also be `nil` or `none` specifying the empty set, or can be `all`, `full`, `-`, or `:` specifying the range $0-(N-1)$ (alternatively `0-^0`).

[Include text about max value for DTs is unknown, since a etc.. value to be equal to the cardinality of the random variable here. We can't do that, however, because the DT is generic, and could be used with multiple different RVs with different cardinalities.]

5.2 Observation/feature file formats

GMTK supports four different types of feature file formats, ICSI PFILEs, HTK files, lists of raw binary features, and lists of plain ASCII features. The type of file format used is determined by the command line arguments given to the program.

The format of ICSI PFILEs is not described in this document. Please refer to the ICSI Sprach distribution [15] for details about PFILEs. Similarly, HTK files are not described in this document; the format of these files is described in the HTK book [21]. It is worth noting here, however, that an HTK feature file is an ASCII file containing a list of files each of which contains the features of an utterance. The format of the feature files for each utterance is binary data consisting of a 12-byte header followed by the features. ICSI PFILEs place all of the utterances and any header information into one large file. PFILEs are advantageous when the training corpus is not very large, as the utterances for an entire training or test set can be very easily manipulated. HTK files are better for larger corpora.

Binary and ASCII files are similar to HTK files, except that the underlying feature files for each utterance are raw binary or ASCII respectively. In other words, an ASCII or binary file consists first of a single ASCII file that specifies a list of other files. Each file in that list must exist, and must contain a raw binary matrix of features (binary file) or a raw ASCII matrix of features. Note that ASCII and binary files can not be intermixed.

Several GMTK programs use the same command line arguments to specify these files, and so are described in this section. They take the form as follows:

`-of1 str`	*Observation File 1 {}*
`-nf1 unsigned`	*Number of floats in observation file 1 {0}*
`-ni1 unsigned`	*Number of ints in observation file 1 {0}*
`-fr1 str`	*Float range for observation file 1 {all}*
`-ir1 str`	*Int range for observation file 1 {all}*
`-fmt1 str`	*Format (htk,bin,asc,pfile) for observation file 1 {pfile}*
`-iswp1 bool`	*Endian swap condition for observation file 1 {F}*

Note that there is the digit "1" after each command line option. This means that these options are for the first file. There are other command line options with a suffix "2" and "3" corresponding to the second and third file respectively. The first, second, and third may be in different formats (i.e., PFILEs, HTK files, etc.) but each such file specification must have the same number of utterances, and the number of frames per utterance must also be the same across files. Some of these files might contain floating point (continuous) value features, others might contain discrete features, and still others might contain a combination of both. The features are combined together into one global observation matrix as described in Section 4.7.1.

We now describe the options in detail.

- `-of1 str` Specifies the file name of the first feature file.
- `-nf1 str` The number of "continuous" floating point values per time frame contained in the feature file.
- `-ni1 str` The number of integer values contained in the feature file. Note that some file types might have only integers, and others might have only floating point values. HTK files, for example, may contain only integers or floats, but not both. PFILEs may contain floats

and/or integers. ASCII and binary files may also contain any number of either integers or floats. Because the program can not in general for all file types figure out how many values are in what format, the user must specify this on the command line.

- `-fr1 str` This option is a string specifying the subset of floating point values, contained in the file, that should be used and placed into the global observation matrix. The default option is to use all values. This means that any arbitrary (possibly sparse) subset of features may be chosen and placed together in consecutive array elements in the global matrix. The format of the range string is a matlab-like list of integers (see Section 5.1).
- `-ir1 str` This option is similar to `-fr1` except it chooses from the integers in the file.
- `-fmt1 str` This option specifies the format of the file: PFILE, an HTK file, an ASCII file, or a binary file.
- `-iswp1 bool` This option specifies whether to swap the byte order of the binary data as it is read in. This is needed as the data might have been generated on a machine with a different Endian as the one that is currently being used. This option allows files with differing byte orders to be used together. GMTK does not figure out automatically the Endian of the file (which would not be possible anyway for raw binary data).

5.3 `gmtkEMtrain`

`gmtkEMtrain` is the main EM training program for GMTK. It is used for training the parameters of a graphical model based on the data that is contained in a list of observation matrices. This section describes the command line options for this program in detail. Observation feature files and their command line options are described in Section 5.2. First, here is a complete list of `gmtkEMtrain`'s command line options and their default values.

```
<-of1 str>                          Observation File 1 {}
<-inputMasterFile str>              Input file of multi-level master CPP processed GM input parameters {}
<-strFile str>                      Graphical Model Structure File {}
[-nf1 unsigned]                     Number of floats in observation file 1 {0}
[-ni1 unsigned]                     Number of ints in observation file 1 {0}
[-fr1 str]                          Float range for observation file 1 {all}
[-ir1 str]                          Int range for observation file 1 {all}
[-fmt1 str]                         Format (htk,bin,asc,pfile) for observation file 1 {pfile}
[-iswp1 bool]                       Endian swap condition for observation file 1 {F}
[-of2 str]                          Observation File 2 {}
[-nf2 unsigned]                     Number of floats in observation file 2 {0}
[-ni2 unsigned]                     Number of ints in observation file 2 {0}
[-fr2 str]                          Float range for observation file 2 {all}
[-ir2 str]                          Int range for observation file 2 {all}
[-fmt2 str]                         Format (htk,bin,asc,pfile) for observation file 2 {pfile}
[-iswp2 bool]                       Endian swap condition for observation file 2 {F}
[-of3 str]                          Observation File 3 {}
[-nf3 unsigned]                     Number of floats in observation file 3 {0}
[-ni3 unsigned]                     Number of ints in observation file 3 {0}
[-fr3 str]                          Float range for observation file 3 {all}
[-ir3 str]                          Int range for observation file 3 {all}
[-fmt3 str]                         Format (htk,bin,asc,pfile) for observation file 3 {pfile}
[-iswp3 bool]                       Endian swap condition for observation file 3 {F}
[-cppCommandOptions str]            Additional CPP command line {}
[-outputMasterFile str]             Output file to place master CPP processed GM output parameters {}
[-inputTrainableParameters str]     File of only and all trainable parameters {}
[-binInputTrainableParameters bool] Binary condition of trainable parameters file {F}
[-objsNotToTrain str]               File listing trainable parameter objects to not train. {}
[-outputTrainableParameters str]    File to place only and all trainable output parametes {}
[-binOutputTrainableParameters bool] Binary condition of output trainable parameters? {F}
[-wpaeei bool]                      Write Parameters After Each EM Iteration Completes {T}
[-seed bool]                        Seed the random number generator {F}
[-maxEmIters unsigned]              Max number of EM iterations to do {3}
[-beam float]                       Beam width (less than max*exp(-beam) are pruned away) {1.000000e+10}
[-allocateDenseCpts]                Automatically allocate any undefined CPTs.
                                    arg = 1 means use random initial CPT values. arg = 2, use uniform values {0}
[-mcvr double]                      Mixture Coefficient Vanishing Ratio {1.000000e+20}
[-botForceVanish unsigned]          Number of bottom mixture components to force vanish {0}
```

```
[-mcsr double]                    Mixture Coefficient Splitting Ratio {1.000000e+10}
[-topForceSplit unsigned]         Number of top mixture components to force split {0}
[-meanCloneSTDfrac double]        Fraction of mean to use for STD in mean clone {1.000000e-01}
[-covarCloneSTDfrac double]       Fraction of var to use for STD in covar clone {1.000000e-01}
[-dlinkCloneSTDfrac double]       Fraction of var to use for STD in covar clone {0.000000e+00}
[-cloneShareMeans bool]           Gaussian component clone shares parent mean {F}
[-cloneShareCovars bool]          Gaussian component clone shares parent covars {F}
[-cloneShareDlinks bool]          Gaussian component clone shares parent dlinks {F}
[-varFloor double]                Variance Floor {1.000000e-10}
[-floorVarOnRead bool]            Floor the variances to varFloor when they are read in {F}
[-lldp float]                     Log Likelihood difference percentage for termination {1.000000e-03}
[-mnlldp float]                   Absolute value of max negative Log Likelihood difference
                                  percentage for termination {1.000000e-02}
[-trrng str]                      Range to train over segment file {all}
[-storeAccFile str]               Store accumulators file {}
[-loadAccFile str]                Load accumulators file {}
[-loadAccRange str]               Load accumulators file range {}
[-llStoreFile str]                File to store previous sum LL's {}
[-accFileIsBinary bool]           Binary accumulator files (def true) {T}
[-startSkip integer]              Frames to skip at beginning (i.e., first frame is buff[startSkip]) {0}
[-endSkip integer]                Frames to skip at end (i.e., last frame is buff[len-1-endSkip]) {0}
[-cptNormThreshold double]        Read error if |Sum-1.0|/card > norm_threshold {1.000000e-02}
[-random bool]                    Randomize the parameters {F}
[-enem bool]                      Run enumerative EM {F}
[-showCliques integer]            Show the cliques after the network has been unrolled k times. {0}
[-numSplits integer]              Number of splits to use in logspace recursion (>=2). {3}
[-baseCaseThreshold integer]      Base case threshold to end recursion (>=2). {1000}
[-gaussianCache bool]             Cache Gaussians evaluations during EM training.
                                  Will speeds things up, but uses more memory. {T}
[-argsFile <str>]                 File to obtain additional arguments from {}
```

- `-of1 str` The name of the first feature file. Since feature file arguments are described generically above in Section 5.2, this option, and the options for the other two feature files are not describe here.

- `-cppCommandOptions str` This is a string that gives additional commands to the CPP command line. This makes it possible, for example, to make CPP defines on the GMTK command line. This can be useful for specifying paths of certain files, ranges to use, and so on. See your local `cpp` manual page to see the syntax of options it supports, but note that typically, if you want to define a CPP variable `FOO` to have value, say, 5, you would give the option `-DFOO=5`.

- `-outputMasterFile str` This option gives the file containing the output master file, as described in Section 3.4.3.

- `-inputTrainableParameters str` The file containing trainable parameters in a predefined format, as described in Section 3.4.3.

- `-binInputTrainableParameters bool` A boolean specifying whether or not the input trainable parameters file should be in binary or ASCII format. Like all boolean arguments, it can be `T/F`, `0/1`, `true/false` and so on.

- `-objsNotToTrain str` This option specifies a file that contains a listing of all GMTK objects that should be held fixed during training, rather than be updated at the end of each training iteration. The format of this file is ASCII, and is pre-processed by CPP. The file itself consists of a collection of keywords followed by object names. The keywords specify the type of object (the namespace if you will), and the object name specifies the name of the object in that object name space to not train. The keywords may be any of the following, corresponding to the object types listed in Table 3.1.

  ```
  DPMF
  SPMF
  MEAN
  COVAR
  ```

```
DLINKMAT
WEIGHTMAT
GAUSIANCOMPONENT
DENSECPT
SPARSECPT
DETERMINISTICCPT
MIXGAUSSIAN
```

The name of the object can either be the object name, or the special string `*` which states that all objects of that kind should not be trained. For example, in the following file

```
MEAN foo
COVAR bar
MEAN baz
DENSECPT *
```

it states that the mean objects called `foo` and `baz`, the covariance object called `bar`, and all dense CPTs should be held fixed during EM training.

- `-outputTrainableParameters str` This option specifies a file in which to place all *trainable* output parameters. The parameters are written in a specific order, as described in Section 3.4.3.
- `-binOutputTrainableParameters bool` A boolean specifying if the output trainable file should be ASCII or binary.
- `-wpaeei bool` A boolean that determines if all of the parameters should be written after each EM iteration completes, rather than only once at the end after EM has converged.
- `-seed bool` Boolean stating if the random number should be seeded.
- `-maxEmIters unsigned` An integer stating the maximum number of EM iterations that should be used during training.
- `-beam float` The beam width parameter, which is a floating point value that controls how GMTK does pruning. During each message pass between cliques in a graphical model, values that are less than $me^{-\text{beam}}$ are pruned away, where m is the maximum clique probability. This is a generalization of standard beam pruning in speech recognition systems.
- `-allocateDenseCpts unsigned` Normally, all CPTs must be declared in parameter files for them to be usable as objects in a structure file. With this option, it is possible to automatically allocate dense CPTs that do not exist in the parameter file, just by naming them in a structure file. For setting up initial experiments, this option therefore might make it easier to get going. The unsigned argument to this command determines how the parameter values are initialized. If the value is zero (0), then CPTs are not allocated (the default). If the value is one (1), it means that random initial CPT values should be used. If the argument is two (2), then uniform initial values are used. See also Section 5.7 for information on how to allocate initial values for these parameters without needing to run any EM training.
- `-mcvr double` This is the mixture coefficient vanishing ratio (MCVR) which determines when components in a Gaussian mixture should "vanish" when the component responsibility gets too small. The MCVR works as follows. Let the i^{th} component's responsibility be

p_i. At the end of an EM iteration, the component will vanish if $p_i < 1/(N \times \text{MCVR})$ where N is the number of components in the distribution. Therefore, each mixture has its own vanishing threshold depending on the number of components that it currently uses.

The larger MCVR becomes, the less chance that a vanish will occur. To turn off vanishing completely, set this parameter to something in the range of 10^{10}. A typical value to allow for some vanishing is to set MCVR to about 50. To allow for aggressive vanishing, set MCVR to about 5 or so.

- `-botForceVanish unsigned` This next option is another way to vanish Gaussian components. This option specifies the number of mixture components that should be forced to vanish, irregardless of the vanishing ratio. The parameter that is specified here (say V) determines the number of components that should vanish, and it vanishes the components with the smallest V responsibilities.

- `-mcsr double` The mixture coefficient splitting ratio (MCSR) is the analog to the MCVR. GMTK's splitting algorithm is similar to the HTK algorithm for mixing up Gaussian components. In GMTK, the criterion can be controlled by the MCSR. A Gaussian component is split if its responsibility becomes larger than a threshold. Again, if p_i is the i^{th} component's responsibility, the component is split if $p_i > \text{MCSR}/N$ where again N is the current number of components.

  Like the MCVR, the larger MCSR becomes, the less the chance will be that a split occurs. To turn off splitting, set MCSR to be about 10^{10}. To force a split of all Gaussian components, set MCSR to be something very small (or see the `-topForceSplit` option.

  In general, the MCVR/MCSR ratios should be set differently for each EM training iteration. I.e., one should **not** set them to be the same for each such training iteration, as that would lead to an ever splitting/vanishing set of Gaussians. When training up a system, it is best to split at one EM iteration, and then allow things to stabilize for a few EM iterations without having any splitting or vanishing occur (or perhaps having mild vanishing during this time). After things have stabilized, it is often useful to repeat the cycle: aggressive splitting for 1 EM iteration, no splitting and very mild vanishing for about 4-6 EM iterations, and then repeat. In general, the set of values that MCVR/MCSR are set to at each EM iteration is called the splitting/vanishing *schedule*.

  Here is a schedule[1] that was successfully used on the Aurora 2.0 noisy speech task with a phone-based GMTK recognizer. The schedule is not "optimized" in any way (meaning that other schedules might do better) but this one seems to work fairly well.

  Step 0: Start the training with 1 Gaussian component per mixture

  Step 1: Train multiple EM iterations with no splitting or vanishing until you have reached 2% convergence. This means that the relative difference in the global log likelihood is less than 2%. You can get the global log-likelihood at the end of each EM iteration using the option `-llStoreFile` which will store that number (in ASCII) to a file.

  Step 2: Do an iteration that splits all Gaussians, and then continue training with no splitting/vanishing until 2% convergence ratio is achieved.

  Repeat steps 1 and 2 a few times (typically 5). At this point it becomes dangerous to split all Gaussians because some of them will start to have determinants that reach the minimum, which will kill the job. So, then we start vanishing:

[1] Courtesy of Karen Livescu from MIT

Step 3: Run one EM iteration with mcvr = 10 and mcsr = 1.

Step 4: Continue training with no splitting/vanishing until 2% convergence.

Repeat steps 3 and 4 a few times. After a while some of the Gaussians again might start to have determinants that hit the minimum, so we then do:

Step 5: Set mcvr = 10, mcsr = 1 for 1 EM iteration.

Step 6: Set mcvr = 10, mcsr = 1e10 for 1 iteration (just to kill off some of the weak Gaussians from step 5)

Step 7: Continue training with splitting/vanishing until 2% convergence.

Repeat steps 5, 6, and 7 a few times. Finally to do the final following steps to achieve convergence:

Step 8: Set mcvr = 10, mcsr = 5 for 2 iterations.

Step 9: Train with no splitting/vanishing until 0.2% convergence.

It was found that one can pretty much go on and on with this schedule, and performance keeps improving at least until 15k Gaussians (or so) are used.

Note that testing for relative convergence between an EM iteration when splitting and/or vanishing is allowed will not work, because if the two iterations you're comparing have different numbers of Gaussians then the EM likelihood is not guaranteed to increase (and is therefore not comparable). Therefore, when testing for relative convergence, make sure that it is between two EM iterations in which no splitting or vanishing has occurred.

In general, the splitting/vanishing schedule is more art than science. If you happen to discover a schedule that works well, then please let me (JB) know about it.

- `-topForceSplit unsigned` In some cases, it can be useful to have the components with the top (largest) responsibility split. This option does that, and forces, at the end of an EM iteration, the splitting of those most probable components.

- `-meanCloneSTDfrac double`, `-covarCloneSTDfrac double`, `-dlinkCloneSTDfrac double`. When a split of a component Gaussian occurs, the component splits into two components. One of the two new components is the same as the old component. The other new component (the cloned component) is a random function of the old component, and these options determine the parameters of the random transformation. Essentially a clone is made of the Gaussian G, so the original G is left alone and its clone G' is randomly perturbed. As mentioned in Section 4.5.1, GMTK represents Gaussians in terms of their means, variances, and dlink matrices. Based on this Gaussian decomposition, there are a number of options when this occurs.

  The `-meanCloneSTDfrac double` option determines the fraction of the original mean to use for the standard deviation in a Gaussian that is used to form the new mean as a function of the old mean. In other words,

$$\mu' = \mu + \text{meanCloneSTDfrac}\mu\mathcal{N}(0,1)$$

  where μ' is the new mean, μ is the old mean, and $\mathcal{N}(0,1)$ is a sample from a zero mean unity variance Gaussian. This means that setting `meanCloneSTDfrac` to zero causes the mean of the cloned Gaussian to be equal to the original. A typical value for this parameter that has worked in the past is anywhere from 0.1 to 0.25, but this will not necessarily be the "best" for a given application.

The option `-covarCloneSTDfrac double` is the fraction of the original variance to use for the standard deviation in a Gaussian that is used to form the new variance as a function of the old variance. Therefore, this option is similar to the case for the mean, specifically:

$$D' = D + \text{covarCloneSTDfrac} D\mathcal{N}(0,1)$$

So far, the value of 0.0 has been tried for this parameter.

Lastly, for the dlink matrix B, the `-dlinkCloneSTDfrac double` option exists. Again, only value 0.0 has been tried for this parameter.

- `-cloneShareMeans bool`, `-cloneShareCovars bool`, `-cloneShareDlinks bool`. When a split occurs and a clone is made of a component, GMTK provides the option to have the mean, variance, and/or dlink matrix of the clone to be shared with the original component, and these options determine if any of the Gaussian portions should be shared after a clone. Of course, if a portion is shared, there is no random transformation between the old and new portion.

  Note that if any of these options are on, that implies that the Gaussian component and its clone will have a portion of itself (i.e., either a mean, diagonal covariance matrix, or B matrix) tied for the remainder of the life of the two components. If one of the components vanishes, then the shared portion will remain attached to the component that survives.

  In general, these options provide a way of sharing portions of Gaussians without having to explicitly or by-hand specify how to do sharing in the parameter files. With these options turned on, sharing can start occur while growing the Gaussians from one component to multi-component mixtures.

- `-varFloor double` Controls the variance floor, meaning that if any of the diagonal variances of a Gaussian falls below this value, then the variance is "floored" (prohibited from falling below the floor value). The variance, in this case, is set to the corresponding variance from the previous EM iteration.

- `-floorVarOnRead bool` In certain cases, it is useful to have the variances floored to the floor value right when the parameters are read in (if for example the parameters were trained using a lower `-varFloor` value, but a larger value is now desired. This option controls this behavior.

  Note that neither `-varFloor` nor `-floorVarOnRead` will directly change and/or affect the vanishing algorithm. It is in general the case, however, that when the variances get very small, the corresponding mixture coefficient responsibility will have a very low value, and is more likely to be vanished. Therefore, if `varFloor` is decreased, that can have the indirect effect of allowing components to have smaller responsibilities (smaller at least than if `varFloor` was larger). This might then encourage more vanishing. One must be careful, however, as if `varFloor` is very small, and the variances themselves get too small, then other numerical issues will arise (GMTK will produce numerous warning messages about them).

  Similarly, more aggressive vanishing (lower MCVR) can be performed to try to eliminate those small variance components, something that might allow for a larger `-varFloor`.

  In general, the vanishing and variance floor options will operate together in interesting and mysterious ways.

- `-lldp float` This option determines the log likelihood difference percentage (lldp) for termination of an EM iteration. If the overall percent difference between the previous EM iteration's log likelihood and the current iteration's log likelihood falls below this threshold, EM training will end.
- `-mnlldp float` NOT CURRENTLY USED.
- `-trrng str` Gives an integer range of training utterance numbers to train over, using the standard integer range specification (Section 5.1).
- `-storeAccFile str` After an EM iteration or a partial EM iteration, this option gives the file name in which to store all the EM accumulators for all trainable objects. If this option is specified, then after forming the accumulators by doing a pass through the training data, the accumulators are written to a file, and then the program exits (i.e., it does not update the parameters). The reason for this is that the accumulators are then used in another run of the training program which accumulates together all of the accumulators that were created by different runs of `gmtkEMtrain`, and updates the parameters. This option is useful for parallel training, as described in Section 5.3.1.
- `-loadAccFile str` Before starting an EM iteration, it is possible to load initial accumulators. This option is used with parallel training.
- `-loadAccRange str` This option provides an integer range (Section 5.1) that is used to choose and automatically load a number of different accumulator files, each of which have been generated by different partial runs of `gmtkEMtrain`. If the file name given by the `-loadAccFile` option contains the string `@D` then for each integer given in the range, the `@D` is substituted by each range integer, and the corresponding accumulator file is loaded.

  As an example, suppose the `-loadAccFile` file is given as `foo@D.acc` and that the range is given as `0:3`. Then the following four accumulator files will be assumed to exist and will be loaded: `foo0.acc`, `foo1.acc`, `foo2.acc`, `foo3.acc`
- `-llStoreFile str` Specifies the file to store the overall log likelihood of the training data. This is used to determine convergence during external and parallel EM training (see Section 5.3.1).
- `-accFileIsBinary bool` Set to true if the accumulator files are in binary rather than ASCII. Since the accumulator files can be large, this option should almost always be true (the default).
- `-startSkip integer` When using dlinks into the past, it is important that the parents of children at early frames do not reference to frames before the beginning of the global observation matrix. If a dlink dependency specifies a frame before the global matrix, then a run-time error will occur. This option says that some number of frames should be skipped at the beginning of each utterance. Those skipped frames may be parents in a dlink matrix (since elements in the global matrix exist for these now negative frame indices).
- `-endSkip integer` This is the analog to `-startSkip` except it operates on the end of the observation matrix, since dlinks are allowed to specify parents in the future as well as the past.

- `-cptNormThreshold double` GMTK will check to ensure that, after they are read in, the CPTs are appropriately normalized (i.e., sum to unity). This option controls the unity tolerance. A read error will occur if there exists a CPT such that $|S - 1.0|/C > \tau$ where S is the sum over the CPT probabilities (which should be unity), C is the cardinality of the random variable (which means that the tolerance automatically increases for greater cardinality variables), and τ is the threshold given as a command line argument.
- `-random bool` True if all parameters should be randomized after being read in.
- `-enem bool` True if enumerative EM should be run. This will be extremely slow as it does not use junction trees at all. This option is there only to verify probability values.
- `-showCliques integer` This option instructs GMTK to unroll the graph some number of times (given by the command line integer argument), triangulate the graph, and then print out the resulting cliques and their sizes. This can be useful to examine the cliques and thereby obtain information on the running time of the graph that you are using.
- `-numSplits integer` During log-space inference (Section XXX), this argument determines the number of splits to use when training. Essentially, this argument determines the base of the logarithm when we say that the running time goes from $O(ST)$ (and respectively space goes from $O(ST)$) during exact inference to $O(ST \log T)$ time (respectively $O(S \log T)$ space) in log-space inference. In general, the larger the value, the less space will be required.
- `-baseCaseThreshold integer` When the number of frames falls below a threshold, log-space inference starts using exact space inference. In other words, this option specifies the base case threshold to end the log-space recursion.
- `-gaussianCache bool` This option determines if Gaussian probability evaluations are cached during EM training. If this is true, the program will run faster, but will use more memory. The amount of extra memory is not that large, however, so there is almost no reason not to keep this set to true.
- `-argsFile <str>` Any of the command line arguments can be obtained from a file rather than from the command line. This option specifies a file from which options are to be obtained. The file consists of keyword-value pairs separated by a colon, where the keywords are the same as the command line parameters without the dash. Argument files may be interspersed with command line arguments, and the order of processing is sequentially through the command line, descending into files where argument files are given. For example, the following is a valid argument file:

```
strFile: foo.str
inputMasterFile: foo.master
of1: foo.htk
nf1: 39
fmt1: htk
```

5.3.1 `gmtkEMtrain`, EM iterations, and parallel training

As was alluded to above, `gmtkEMtrain` supports parallel EM training. In fact, there are two ways to run EM training, both internal to the program and external. In internal EM training, multiple

EM iterations are run for one single invocation of the `gmtkEMtrain` program. In external EM training, each EM iteration corresponds to running `gmtkEMtrain` $M+1$ times. It is run M times to create M different accumulator files, where each accumulator file corresponds to a different set of observation files, and is run a final time to bundle together all the accumulator files, update the parameters, and write the new parameters out to a file. In this latter mode of processing, each of the M runs of `gmtkEMtrain` can be run simultaneously on different processors, which is how parallel EM training works.

The following is a simple example of a script that runs one iteration of EM externally using `gmtkEMtrain`. It runs `gmtkEMtrain` $M+1=3$ times (so that $M=2$). It does not run the programs in parallel, but it can easily be seen where the parallelism arises.

```
GMTK Script using external EM.
```

5.3.2 `gmtkEMtrain` tips

In this section, we list a number of tips for using `gmtkEMtrain`.

Determinants becoming too small

In some cases, you might find that `gmtkEMtrain` reports that determinants of Gaussians become smaller than is possible for the finite precision IEEE floating point arithmetic to handle. This happens only when the determinant of a covariance matrix (something that `gmtkEMtrain` needs to compute for doing an EM update) becomes smaller than the minimum possible double precision floating point value (i.e., there is no threshold that can be adjusted in this case).

In general, when this happens there are several possible solutions:

- use more training utterances, as this often occurs when a component does not get enough counts.
- make sure the observations are scaled appropriately. For example, if you take a set of observation features, and divide each feature by, say, 1e10, there is no theoretical information loss, but the the variances of Gaussians (and therefore the determinants) will become much smaller thereby potentially leading to a minimum variable problem. If such a thing happens, you can try simply multiplying all your features by, say, 1000 so that the features are in the range more appropriate for finite precision arithmetic.
- Make sure that the features were generated with the appropriate options. There are many options when using MFCCs, for example. Also, it is often the case that brand new feature sets (something that GMTK with BMMs and dlink matrices is uniquely suitable for) are not appropriately scaled at first.

5.4 `gmtkViterbi`

`gmtkViterbi` is the main GMTK decoding program. Many of the command line options are the same as for `gmtkEMtrain`, so only the ones that are unique to `gmtkViterbi` will be described here.

Currently, `gmtkViterbi` works by running a single pass of max-product inference over the network. This means that the procedure finds the single most probable configuration of all hidden variables. Once these hidden variables are instantiated (such as a word variable), the variables over time may optionally be written out to disk. The time frame that the instantiated hidden variable is written out may also be determined using another hidden variable (such as a transition variable). At the moment, GMTK does not allow for some variables to be integrated (via summation) and others to be maxed, but this will change in future versions of GMTK.

Here is the complete set of command line options.

```
<-of1 str>                      Observation File 1 {}
<-strFile str>                  GM Structure File {}
<-inputMasterFile str>          Input file of multi-level master CPP processed GM input parameters {}
[-nf1 unsigned]                 Number of floats in observation file 1 {0}
[-ni1 unsigned]                 Number of ints in observation file 1 {0}
[-fr1 str]                      Float range for observation file 1 {all}
[-ir1 str]                      Int range for observation file 1 {all}
[-fmt1 str]                     Format (htk,bin,asc,pfile) for observation file 1 {pfile}
[-iswp1 bool]                   Endian swap condition for observation file 1 {F}
[-of2 str]                      Observation File 2 {}
[-nf2 unsigned]                 Number of floats in observation file 2 {0}
[-ni2 unsigned]                 Number of ints in observation file 2 {0}
[-fr2 str]                      Float range for observation file 2 {all}
[-ir2 str]                      Int range for observation file 2 {all}
[-fmt2 str]                     Format (htk,bin,asc,pfile) for observation file 2 {pfile}
[-iswp2 bool]                   Endian swap condition for observation file 2 {F}
[-of3 str]                      Observation File 3 {}
[-nf3 unsigned]                 Number of floats in observation file 3 {0}
[-ni3 unsigned]                 Number of ints in observation file 3 {0}
[-fr3 str]                      Float range for observation file 3 {all}
[-ir3 str]                      Int range for observation file 3 {all}
[-fmt3 str]                     Format (htk,bin,asc,pfile) for observation file 3 {pfile}
[-iswp3 bool]                   Endian swap condition for observation file 3 {F}
[-inputTrainableFile str]       File of only and all trainable parameters {}
[-binInputTrainableFile bool]   Binary condition of trainable parameters file {F}
[-cppCommandOptions str]        Command line options to give to cpp {}
[-varFloor double]              Variance Floor {1.000000e-10}
[-floorVarOnRead bool]          Floor the variances to varFloor when they are read in {F}
[-dcdrng str]                   Range to decode over segment file {all}
[-beam float]                   Beam width (less than max*exp(-beam) are pruned away) {1.000000e+10}
[-showVitVals bool]             Print the viterbi values?? {F}
[-printWordVar str]             Print the word var - which has this label {}
[-varMap str]                   Use this file to map from word-index to string {}
[-transitionLabel str]          The label of the word transition variable {}
[-startSkip integer]            Frames to skip at beginning (i.e., first frame is buff[startSkip]) {0}
[-endSkip integer]              Frames to skip at end (i.e., last frame is buff[len-1-endSkip]) {0}
[-cptNormThreshold double]      Read error if |Sum-1.0|/card > norm_threshold {1.000000e-02}
[-showCliques integer]          Show the cliques after the netwok has been unrolled k times. {0}
[-dumpNames str]                File containing the names of the variables to save to a file {}
[-ofilelist str]                List of filenames to dump the hidden variable values to {}
[-numSplits integer]            Number of splits to use in logspace recursion (>=2). {3}
[-baseCaseThreshold integer]    Base case threshold to end recursion (>=2). {1000}
[-argsFile <str>]               File to obtain additional arguments from {}
```

Let us now go though each option. Again, the ones that are already described above are not repeated here.

- `-dcdrng str` This option provides the integer range (see Section 5.1) over the utterance file to decode.
- `-showVitVals bool` This is a boolean that, if true, states the the Viterbi values (the maximum probability values) of the hidden variables should be printed out.
- `-printWordVar str` Gives the name of the variable (typically a word variable) that should be printed out.
- `-transitionLabel str` Gives the name of a binary transition variable that, when unity, determines when (i.e., for which frames) the word variable given by `-printWordVar` should be printed out.
- `-varMap str` Gives a name of an ASCII file that contains a list of words. This is the mapping from integer word variable value to actual textual word string, and these strings are used for printing rather than the integers.

- `-dumpNames str` The file that contains a list of the names of all the variables that should be saved to a file.
- `-ofilelist str` The name of the file that contains a list of filenames to which the variables values listed in `-dumpNames` should be written. The number of file names listed in the file should be the same as the number of utterances decoded.

5.5 gmtkScore

`gmtkScore` is the main GMTK re-scoring program, for producing a score from a given model. I.e., it simply integrates over all hidden variables, and computes $P(X)$, where X are all the observed variables, and then prints out $\log P(X)$.

Here is the complete list of `gmtkScore`'s options.

```
 <-of1 str>                      Observation File 1 {}
 <-strFile str>                  GM Structure File {}
 <-inputMasterFile str>          Input file of multi-level master CPP processed GM input parameters {}
 [-nf1 unsigned]                 Number of floats in observation file 1 {0}
 [-ni1 unsigned]                 Number of ints in observation file 1 {0}
 [-fr1 str]                      Float range for observation file 1 {all}
 [-ir1 str]                      Int range for observation file 1 {all}
 [-fmt1 str]                     Format (htk,bin,asc,pfile) for observation file 1 {pfile}
 [-iswp1 bool]                   Endian swap condition for observation file 1 {F}
 [-of2 str]                      Observation File 2 {}
 [-nf2 unsigned]                 Number of floats in observation file 2 {0}
 [-ni2 unsigned]                 Number of ints in observation file 2 {0}
 [-fr2 str]                      Float range for observation file 2 {all}
 [-ir2 str]                      Int range for observation file 2 {all}
 [-fmt2 str]                     Format (htk,bin,asc,pfile) for observation file 2 {pfile}
 [-iswp2 bool]                   Endian swap condition for observation file 2 {F}
 [-of3 str]                      Observation File 3 {}
 [-nf3 unsigned]                 Number of floats in observation file 3 {0}
 [-ni3 unsigned]                 Number of ints in observation file 3 {0}
 [-fr3 str]                      Float range for observation file 3 {all}
 [-ir3 str]                      Int range for observation file 3 {all}
 [-fmt3 str]                     Format (htk,bin,asc,pfile) for observation file 3 {pfile}
 [-iswp3 bool]                   Endian swap condition for observation file 3 {F}
 [-inputTrainableFile str]       File of only and all trainable parameters {}
 [-binInputTrainableFile bool]   Binary condition of trainable parameters file {F}
 [-cppCommandOptions str]        Command line options to give to cpp {}
 [-varFloor double]              Variance Floor {1.000000e-10}
 [-floorVarOnRead bool]          Floor the variances to varFloor when they are read in {F}
 [-dcdrng str]                   Range to decode over segment file {all}
 [-beam float]                   Beam width (less than max*exp(-beam) are pruned away) {1.000000e+10}
 [-startSkip integer]            Frames to skip at beginning (i.e., first frame is buff[startSkip]) {0}
 [-endSkip integer]              Frames to skip at end (i.e., last frame is buff[len-1-endSkip]) {0}
 [-cptNormThreshold double]      Read error if |Sum-1.0|/card > norm_threshold {1.000000e-02}
 [-showCliques bool]             Show the cliques of the not-unrolled network {F}
 [-numSplits integer]            Number of splits to use in logspace recursion (>=2). {3}
 [-baseCaseThreshold integer]    Base case threshold to end recursion (>=2). {1000}
 [-argsFile <str>]               File to obtain additional arguments from {}
```

All of the options for `gmtkScore` have already been described.

5.6 gmtkSample

`gmtkSample` allows you to sample from a graphical model. It currently only samples discrete variables. In the next release, GMTK will allow sampling of both discrete and continuous variables.

Here is the complete list of command line options.

```
Required: <>; Optional: []; Flagless arguments must be in order.
 <-of1 str>                      Observation File 1 {}
 <-strFile str>                  GM Structure File {}
 <-inputMasterFile str>          Input file of multi-level master CPP processed GM input parameters {}
 <-dumpNames str>                File containing the names of the variables to save to a file {}
 <-ofilelist str>                List of filenames to dump the hidden variable values to {}
 [-nf1 unsigned]                 Number of floats in observation file 1 {0}
 [-ni1 unsigned]                 Number of ints in observation file 1 {0}
 [-fr1 str]                      Float range for observation file 1 {all}
```

```
[-ir1 str]                    Int range for observation file 1 {all}
[-fmt1 str]                   Format (htk,bin,asc,pfile) for observation file 1 {pfile}
[-iswp1 bool]                 Endian swap condition for observation file 1 {F}
[-of2 str]                    Observation File 2 {}
[-nf2 unsigned]               Number of floats in observation file 2 {0}
[-ni2 unsigned]               Number of ints in observation file 2 {0}
[-fr2 str]                    Float range for observation file 2 {all}
[-ir2 str]                    Int range for observation file 2 {all}
[-fmt2 str]                   Format (htk,bin,asc,pfile) for observation file 2 {pfile}
[-iswp2 bool]                 Endian swap condition for observation file 2 {F}
[-of3 str]                    Observation File 3 {}
[-nf3 unsigned]               Number of floats in observation file 3 {0}
[-ni3 unsigned]               Number of ints in observation file 3 {0}
[-fr3 str]                    Float range for observation file 3 {all}
[-ir3 str]                    Int range for observation file 3 {all}
[-fmt3 str]                   Format (htk,bin,asc,pfile) for observation file 3 {pfile}
[-iswp3 bool]                 Endian swap condition for observation file 3 {F}
[-inputTrainableFile str]     File of only and all trainable parameters {}
[-binInputTrainableFile bool] Binary condition of trainable parameters file {F}
[-cppCommandOptions str]      Command line options to give to cpp {}
[-samplerng str]              Range to decode over segment file {all}
[-startSkip integer]          Frames to skip at beginning (i.e., first frame is buff[startSkip]) {0}
[-endSkip integer]            Frames to skip at end (i.e., last frame is buff[len-1-endSkip]) {0}
[-argsFile <str>]             File to obtain additional arguments from {}
```

All of these options have been described above for different programs.

5.7 gmtkParmConvert

`gmtkParmConvert` is the program that can convert ASCII to binary files and vice versa, and can automatically allocate and print dense CPTs for you.

Note that it is crucial to convert parameter files to binary as soon as possible since reading them in ASCII is doubly slow – ASCII files are first processed by CPP, and it further takes time to convert the parameter values from ASCII to binary. It is not unheard of, in fact, for CPP to take 20 minutes to process large ASCII files.

Once files are converted to binary, they can easily be reconverted to ASCII for quick viewing using `gmtkParmConvert`. Also, the special file name "-" indicates that that parameters should be written to (respectively, read from) standard output `stdout` (respectively, standard input `stdin`), depending on if the file is an input or output file.

As mentioned in Section 5.3, GMTK has the ability to automatically allocate the parameter objects for dense CPTs. Both the `gmtkEMtrain` and the `gmtkParmConvert` program can do this for you, using the `-allocateDenseCpts` option. `gmtkParmConvert` is a particularly convenient way of doing this as a script need not produce the paramter file objects needed for all of the dense CPTs.

Here is the complete set of command line options:

```
[-inputMasterFile str]               Input file of multi-level master CPP processed GM input parameters {}
[-outputMasterFile str]              Output file to place master CPP processed GM output parameters {}
[-allocateDenseCpts]                 Automatically allocate any undefined CPTs.
[-inputTrainableParameters str]      File of only and all trainable parameters {}
[-binInputTrainableParameters bool]  Binary condition of trainable parameters file {F}
[-outputTrainableParameters str]     File to place only and all trainable output parametes {}
[-binOutputTrainableParameters bool] Binary condition of output trainable parameters? {F}
[-cppCommandOptions str]             Command line options to give to cpp {}
[-varFloor double]                   Variance Floor {1.000000e-10}
[-floorVarOnRead bool]               Floor the variances to varFloor when they are read in {F}
[-cptNormThreshold double]           Read error if |Sum-1.0|/card > norm_threshold {1.000000e-02}
[-argsFile <str>]                    File to obtain additional arguments from {}
```

All of these options have been described above in `gmtkEMtrain`.

Part III

Tutorial

Part IV

Reference

Part V

Bibliography

Bibliography

[1] The BUGS project. http://www.mrc-bsu.cam.ac.uk/bugs/Welcome.html.

[2] CMU Sphinx: Open source speech recognition. http://www.speech.cs.cmu.edu/sphinx/Sphinx.html.

[3] The ISIP public domain speech to text system. http://www.isip.msstate.edu/projects/speech/software/ind

[4] S. M. Aji and R. J. McEliece. The generalized distributive law. *IEEE Transactions in Information Thoery*, 46:325–343, March 2000.

[5] J. Bilmes. *Natural Statistical Models for Automatic Speech Recognition*. PhD thesis, U.C. Berkeley, Dept. of EECS, CS Division, 1999.

[6] J. Bilmes. Graphical models and automatic speech recognition. Technical Report UWEETR-2001-005, University of Washington, Dept. of EE, 2001.

[7] J. Bilmes, O. Cetin, H. Nock, K. Kirchhoff, G. Zweig, and M. Ostendorf. The gmtk-based spine evaluation. *to be submitted to Proc. Int. Conf. on Spoken Language Processing*, 2002.

[8] J. Bilmes and G. Zweig. The Graphical Models Toolkit: An open source software system for speech and time-series processing. *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, 2002.

[9] J.A. Bilmes. Buried Markov models for speech recognition. In *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, Phoenix, AZ, March 1999.

[10] J.A. Bilmes. Dynamic Bayesian Multinets. In *Proceedings of the 16th conf. on Uncertainty in Artificial Intelligence*. Morgan Kaufmann, 2000.

[11] J.A. Bilmes. Factored sparse inverse covariance matrices. In *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, Istanbul, Turkey, 2000.

[12] J. Binder, K. Murphy, and S. Russell. Space-efficient inference in dynamic probabilistic networks. *Int'l, Joint Conf. on Artificial Intelligence*, 1997.

[13] T. Dean and K. Kanazawa. Probabilistic temporal reasoning. *AAAI*, pages 524–528, 1988.

[14] D. Geiger and D. Heckerman. Knowledge representation and inference in similarity networks and Bayesian multinets. *Artificial Intelligence*, 82:45–74, 1996.

[15] Icsi sprach tools. http://www.icsi.berkeley.edu/ dpwe/projects/sprach/sprachcore.html.

[16] F.V. Jensen. *An Introduction to Bayesian Networks*. Springer, 1996.

[17] S.L. Lauritzen. *Graphical Models*. Oxford Science Publications, 1996.

[18] K. Murphy. The Matlab bayesian network toolbox. http://www.cs.berkeley.edu/~murphyk/Bayes/bnsoft.html.

[19] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 2nd printing edition, 1988.

[20] P. Smyth, D. Heckerman, and M.I. Jordan. Probabilistic independence networks for hidden Markov probability models. Technical Report A.I. Memo No. 1565, C.B.C.L. Memo No. 132, MIT AI Lab and CBCL, 1996.

[21] S. Young, J. Jansen, J. Odell, D. Ollason, and P. Woodland. *The HTK Book*. Entropic Labs and Cambridge University, 2.1 edition, 1990's.

[22] G. Zweig. *Speech Recognition with Dynamic Bayesian Networks*. PhD thesis, U.C. Berkeley, 1998.

[23] G. Zweig, J. Bilmes, T. Richardson, K. Filali, K. Livescu, P. Xu, K. Jackson, Y. Brandman, E. Sandness, E. Holtz, J. Torres, and B. Byrne. Structurally discriminative graphical models for automatic speech recognition — results from the 2001 Johns Hopkins summer workshop. *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, 2002.

[24] G. Zweig and M. Padmanabhan. Exact alpha-beta computation in logarithmic space with application to map word graph construction. *Int. Conf. on Spoken Lanugage Processing*, 2000.

[25] G. Zweig and S. Russell. Speech recognition with dynamic Bayesian networks. *AAAI-98*, 1998.

[26] G. Zweig and S. Russell. Probabilistic modeling with bayesian networks for automatic speech recognition. *Australian Journal of Intelligent Information Processing*, 5(4):253–260, 1999.

Index