PROBABILISTIC GRAPHICAL MODELS FOR PATTERN RECOGNITION AND OPTICAL MOTION CAPTURE TRACKING

by

Stjepan Rajko

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

ARIZONA STATE UNIVERSITY

December 2009

PROBABILISTIC GRAPHICAL MODELS FOR PATTERN RECOGNITION AND OPTICAL MOTION CAPTURE TRACKING

by

Stjepan Rajko

has been approved

July 2009

Graduate Supervisory Committee:

Gang Qian, Co-Chair
Hari Sundaram, Co-Chair
Baoxin Li
Goran Konjevod

ACCEPTED BY THE GRADUATE COLLEGE

ABSTRACT

This dissertation discusses several probabilistic graphical models which address problems in pattern recognition and optical motion capture tracking. It first introduces the augmented hidden Markov model with equivalence classes (AHMM+EC), which provides a unifying framework for a large number of existing state-based probabilistic generative models. For example, the AHMM+EC can be used to represent regular hidden Markov models (HMMs), terminating HMMs, hierarchical HMMs, semi-Markov models, reduced-parameter models, Markov models of order larger than 1, and conceptual combinations thereof. The advantage of such a unifying framework is that an algorithm specified or implemented for the AHMM+EC is immediately applicable to any of the models it can represent. The dissertation then shows how the AHMM can be extended into the semantic network model (SNM), in which states of probabilistic models can be marked as semantic states. States are marked when they carry some special meaning to the application, for example the beginning or end of a gesture. Defining semantic states allows formulating and solving problems specifically related to semantic states, which is shown useful in segmentation of an unknown observation sequence, event-driven application frameworks, on-line learning, and finding multiple likely explanations of the data. Both the AHMM and SNM models and related algorithms have been implemented in the open source AME Patterns library, which is presented in a broad overview, and with an accompanying analysis of various tradeoffs and design decisions involved. Finally, the dissertation presents a comprehensive method aimed at making optical motion capture more robust and less time consuming. The first part is an autonomous algorithm for the real-time creation of a moving subject's kinematic model from optical motion capture data and with no a priori information. The second part shows how the automatically built kinematic model can be matched to known persistent models to provide consistent labeling of its elements. The net effect is that the tedious subject calibration phase typically associated with motion capture is completely eliminated.

To Jessica, who was the best imaginable companion on this journey. And to my parents, for being the best parents in the world.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No.0403428. and Grant No. 0504647.

Many thanks go to Gang Qian, Bo Peng and Todd Ingalls, with whom I collaborated in developing the content presented in this dissertation, and to many other students, staff and faculty of the Arts, Media and Engineering Program, who have helped me along the way.

I would also like to thank Gang Qian, Hari Sundaram, Baoxin Li, and Goran Konjevod for serving as members of my doctoral committee and providing their guidance.

Finally, my deepest gratitude goes to Gang Qian, for being a wonderful mentor.

TABLE OF CONTENTS

Page

LIST OF TABLES

Table Page

Table Page

LIST OF FIGURES

Figure Page

Figure Page

Figure Page

1. INTRODUCTION

This dissertation discusses two probabilistic graphical models which address problems in pattern analysis and optical motion capture tracking. The main part of the discussion is divided in four chapters. Chapters 2, 3, and 4 deal with problems in pattern analysis, specifically pattern modeling, recognition, and synthesis, however most of the focus is on pattern recognition applications. The chapters introduce a new graphical model that unifies a number of existing models, extend it to allow patterns to be marked with semantic meaning, and present an open source implementation of the models and algorithms involved. Chapter 5 presents a separate model that addresses optical motion capture tracking.

The model introduced in Chapter 2 is titled augmented hidden Markov model with equivalence classes (AHMM+EC), and provides a unifying framework for a large number of existing state-based probabilistic generative models. Such models have been proposed for the modeling of temporal patterns, and the recognition of patterns such as gestures and speech. Specifically, the AHMM+EC can be used to represent regular hidden Markov models (HMMs) [37], terminating HMMs [2], hierarchical HMMs (HHMMs) [11], hidden semi-Markov models (HSMMs) [31], reduced-parameter models [43], and Markov models of order larger than 1 [48]. This is accomplished through generalized non-emitting states, as well as the use of equivalence classes in training.

Non-emitting states are sometimes used, for example, as "dummy" or "null" start states in regular HMMs, or used as terminal states of an HHMM abstract state. In AHMM+ECs, non-emitting states are treated generally and can be used flexibly for a number of purposes. The equivalence classes are related to models in which training must preserve equality between different parts of the model, either in equality of certain transition probabilities or in equality of certain observation probability distributions. Again, equivalence classes in AHMM+ECs provide a general mechanism which can be used flexibly for the needs of each of the individual models. It is by using the common language of non-emitting states and equivalence classes that AHMM+ECs are able to describe a wide range of models.

Compared to other unifying frameworks such as Dynamic Bayesian Networks (DBNs) [31], or Hierarchical Multi-Channel Hidden Semi Markov Models [33], the AHMM+EC is less general in that it doesn't directly account for factorization of states. However, it has a desirable property that most common algorithms are only marginally more difficult to specify for the AHMM+EC than they are for a regular HMM. Once implemented, these algorithms work without modification for any supported model. This allows researchers and developers to easily evaluate algorithms across a wide range of models, and evaluate a wide range of models for any given task or problem domain. This allows for a more holistic understanding of the properties of different algorithms and models, and facilitates finding the best model / algorithm combination for a given problem domain. It can also facilitate fast prototyping of new algorithms and models. All this makes AHMM+ECs a valuable unifying framework for models that do not employ factorization of states.

The value of AHMM+ECs is further supported by Chapters 3 and 4, which present material based on the AHMM+EC model and are thereby simultaneously applicable to all of the special case model types that the AHMM+ECs can represent.

Chapter 3 introduces the semantic network model (SNM), in which states of generative probabilistic models can be marked as semantic states. States are marked when they carry some special meaning to the application, for example when they represent the beginning or end of a gesture.

Defining semantic states allows us to formulate and solve problems specifically related to semantic states. For example, given a sequence of observations, we can find what semantic states were likely visited, and when they were visited. If the semantic states were used to mark the beginning and end of pattern submodels, that corresponds to finding which patterns occurred in the data, and when they occurred. This gives a more direct way of obtaining the important high-level information, without the need to find the full correspondence between all states and observations.

Obtaining the high-level information more efficiently is especially important in real-time scenarios, such as those involving continuous user interaction via gesture or speech. In such scenarios, we continuously receive new data that can change our perspective on what has happened in the past. For example, what we first interpret as the word "forever" may turn out to more likely correspond to the words "for everyone" as we receive new data. The SNM allows us to pinpoint higher-level events such as recognitions of new words or undoing recognitions of previously hypothesized occurrences, by concisely describing them as changes in the hypothesized sequence of visited semantic states. An application using speech and gesture recognition can then be implemented as an event driven system that responds appropriately to new visits of semantic states, revised times of visitation, and undoing of previously assumed visits to semantic states. In addition to gesture segmentation and event-driven application frameworks, SNMs can also be used to facilitate on-line learning, as well as finding multiple likely explanations of the data.

Because the SNM is presented as an extension of the AHMM+EC, it is applicable to all types of models that are special cases of AHMM+ECs. This in turn validates the usefulness of the AHMM+EC, because rather than having to present how semantic states and related algorithms need to be applied to each of the special case models, the presentation is done only once in the context of AHMM+ECs.

Chapter 4 presents the AME Patterns library, which is a C++ implementation of AHMM+EC and SNM concepts and algorithms, and uses generic programming principles. Because of the use of generic programming, the algorithms can easily be applied to any modality (e.g., speech, gesture, or purely numerical data). This also allows ideas and algorithms developed in one field (e.g., speech recognition) to be immediately usable in other fields, as long as the implementation is generic.

Furthermore, the use of generic programming allows the library to be used with data types that provide different tradeoffs, e.g. between space and time complexity.

As the flexibility of generic C++ programming comes at a cost of high programming complexity and increased compile times, the library also provides higher-level application programming interfaces (APIs). For common uses such as classification, on-line recognition, synthesis, or training, the library provides various *task* class templates which are easier to use and can be somewhat customized via template arguments. We also provide a collection of purely object-oriented C++ classes for a few of the most common usage scenarios and modalities, provided in a pre-compiled library: ofxPatterns. The advantage of ofxPatterns are reduced compile times, and a (sometimes preferred) object-oriented C++ interface. Finally, we have recently started developing an additional library offering a C interface, with the goal of the same functionality as ofxPatterns. The C interface allows the library to be used in environments where C++ is not an option, thus increasing portability. For example, it can be loaded as a shared library into C#. The chapter discusses all such tradeoffs involving programming complexity, flexibility, portability, and compile times in more detail.

Just as SNMs supported the usefulness of AHMM+ECs by validating that it allows a broadly applicable theoretical extension, the AME Patterns library supports the usefulness of AHMM+EC by validating that it allows a broadly applicable software library implementation.

Finally, Chapter 5 presents a comprehensive method aimed at making optical motion capture more robust and less time consuming. While marker-based optical motion capture is a reliable and mature technique to capture body kinematics such as joint angles and joint locations, and has proved useful for many applications, such as rehabilitation, kinesiology, sports training, animation, and interactive arts, using such motion capture is usually expensive, cumbersome, and time consuming. These drawbacks stem from the requirements of marker placement on the subject's body, manual entry of a marker set model, calibration of the subject, imperfect real-time data caused by marker occlusion and noise, and high financial cost of quality optical motion capture system software.

To address these issues, we present a completely autonomous algorithm for the real-time creation of a moving subject's kinematic model from optical motion capture data and with no a priori information. Our approach solves marker tracking, the building of the kinematic model, and the tracking of the body simultaneously. The novelty lies in doing so through a unifying Markov random field framework, which allows the kinematic model to be built incrementally and in real-time. Then, we show how the automatically built kinematic model can be matched to known persistent models to provide consistent labeling of its elements. The net effect is that the time consuming subject calibration phase typically associated with motion capture is completely eliminated. We validate the potential of this method through experiments in which the system is able to accurately track the movement of the human body without an a priori model, as well as through experiments on synthetic data. We also provide an example of a persistent model, to which our system is able to match an automatically constructed kinematic model.

2. THE AUGMENTED HIDDEN MARKOV MODEL WITH EQUIVALENCE CLASSES

2.1 INTRODUCTION

Over the years, many state-based probabilistic models have been proposed for the modeling of temporal patterns, and the recognition of patterns such as gestures and speech. Examples of such models are hidden Markov models (HMMs) [37], terminating HMMs [2], hierarchical HMMs (HHMMs) [11], semi-Markov models [31], reduced-parameter models [43], and Markov models of order larger than 1 [48].

On one hand, the large number of models is an advantage - having models of different properties makes the availability of an adequate model for any given problem more likely. On the other hand, the diversity of the models poses a challenge to research and software development communities - to holistically compare the models, one would have to implement the required algorithms and experimental code for each model in turn.

Fortunately, all of these models share common properties. For example, most of them can be described as instances of dynamic Bayesian networks (DBNs) [31], or as one of the Hierarchical Multi-Channel Hidden Semi Markov Models [33]. However, because DBNs and Hierarchical Multi-Channel Hidden Semi Markov Models allow for state factorization (i.e., the concept that the current state of the model is composed of several factors), algorithms that deal with these general models must take state factorization into account. Since none of the models presented in the opening paragraph use state factorization, this is an unnecessary complication when dealing with these models alone.

The augmented hidden Markov model with equivalence classes (AHMM+EC) proposed in this chapter is also capable of representing each of the models listed in the opening paragraph. This is accomplished through generalized non-emitting states, as well as the use of equivalence classes in training. Non-emitting states are not uncommon in state-based generative models. They are sometimes used, for example, as "dummy" or "null" start states in regular HMMs, or used as terminal states of an HHMM abstract state. In AHMMs, non-emitting states are treated generally and can be used flexibly for a number of purposes. The equivalence classes are related to models in which training must preserve equality between different parts of the model, either in equality of certain transition probabilities or in equality of certain observation probability distributions. Again, equivalence classes in AHMMs provide a general mechanism which can be used flexibly for the needs of each of the individual models. By using the common language of non-emitting states and equivalence classes, AHMMs are able to describe a wide range variations of HMMs, including HMMs with termination probabilities, HHMMs, semi-Markov models, reduced parameter models, and larger order Markov models.

In addition, such common descriptive language of AHMMs based on non-emitting states and equivalence classes leads to simple and generalized training and inference algorithms for AHMMs, independent of the actual model structure such as model topology and number of states. These training and inference algorithms for AHMMs are straightforward extensions from those of the traditional HMMs and they are easy to state and to implement. Once implemented, these algorithms work without modification for any supported model. This allows researchers and developers to easily evaluate algorithms across a wide range of models, and evaluate a wide range of models for any given task or problem domain. This allows for a more holistic understanding of the properties of different algorithms and models, and facilitates finding the best model / algorithm combination for a given problem domain. It can also facilitate fast prototyping of new algorithms and models. In particular, it makes it easy to evaluate conceptual combinations of any of the supported models, enabling discovery of new models which may be more suitable for particular scenarios.

We have already implemented the models and algorithms discussed in this chapter and made them available in the AME Patterns library (see Chapter 4), available to the public in the open source AMELiA distribution [40] under the GNU General Public License. Since its release, several individuals and development groups have downloaded and tried this library with positive feedback.

This chapter is organized as follows. In Section 2.1.1, we review some existing work that is related to the models discussed in this chapter. In Section 2.2, we will begin with our presentation of the augmented hidden Markov model (AHMM), and in Section 2.2.1 our presentation of the equivalence classes used in conjunction with the model. Section 2.3 will describe how a number of other models such as HMMs, HHMMs, etc., can be described as special cases of AHMM+ECs. Section 2.4 adapts the basic inference and training algorithms used with pattern modeling to AHMM+ECs. Finally, Section 2.5 presents a number of experimental results obtained by our implementation of AHMM+ECs. Some of the results merely reproduce results reported on existing literature, and support the correctness of our models and implementation. Other results expand on previously reported experiments by comparing a wider range of models, and show how a wider set of tested models, facilitated by the AHMM+EC framework, can provide a fuller understanding of the problem.

2.1.1 Related Work

There are many bodies of work related to the models discussed in this chapter. Murphy [31] provides an excellent treatment of various HMM-related models, all cast as special cases of dynamic Bayesian networks (DBNs). As a unifying framework, the DBN is a much more powerful model than the AHMM because it allows factorization of the state space. This makes the general DBN more complex than necessary when the needing is to represent only models that do not utilize state factorization. The Hierarchical Multi-Channel Hidden Semi Markov Models [33], introduced by Natarajan and Nevatia, also takes into account factorization of states. Their presentation offers an example of how taking into account state factorization can increase the complexity of algorithms.

E.g., compare the two algorithms presented in [33] to the Viterbi algorithm used for AHMMs, which essentially applies equations (20) and (21) for each time step and for all states.

The AHMM+EC does not take into account factorization of states into account, but for many useful models this is not necessary. The resulting framework is consequently much simpler than DBNs or Hierarchical Multi-Channel Hidden Semi Markov Models, at the expense of generality.

A comprehensive review of hidden Markov models is presented in [37]. The hierarchical hidden Markov model (HHMM) [11] in particular is commonly used for pattern recognition. An HHMM can be viewed as a model that is itself composed of multiple HMMs. This is very useful in cases where the system being modeled is hierarchical, such as in speech recognition. In speech, a sentence can be modeled as a sequence of words, and each word in turn as a sequence of phonemes. An HHMM speech recognition system could model each phoneme by a separate HMM, and then bring the individual HMMs together in an HHMM hierarchy. The AHMM offers equivalent support for representation of hierarchical models. An abstract state in the HHMM corresponds to a pair of non-emitting states bounding a subcomponent of the model in the AHMM.

The non-emitting states used in AHMMs are analogous to the auxiliary states presented in [38]. As described above, they are also related to abstract states in AHMMs. Special cases of non-emitting states can also be found in "dummy" or "null" states which are sometimes used as unique starting or terminating states of an HMM [53].

Before HMMs became the ubiquitous tool for gesture recognition, simpler dynamic programming alignment (DPA) methods were commonly used. These methods solve the problem by comparing a known sequence of observations (a training sample) with an unclassified sequence. This can be done, for example, using the edit distance algorithm [25], or dynamic time warping [19, 8]. Our adaptation of the Viterbi algorithm (Section 2.4.1) to AHMMs evolved from a joint treatment of HMMs and DPAs presented in [41].

Many of the algorithms used for training and inference in all of these models can be described as instances of the sum-product algorithm in factor graphs [20], or the general distributive law [1]. For example, the training of hidden Markov models is typically done using the expectation-maximization algorithm [28]. We adapt this and other algorithms for use with the proposed model.

2.2 The Augmented Hidden Markov Model

Like the HMM and other models mentioned above, the AHMM is a state-based probabilistic generative model. State-based probabilistic generative models are stochastic state machines in which entering some or all states emits an observation. We will refer to those states which emit an observation as *emitting* estates. We also assume that emitting states produce observations from an observation set $\mathbb{O}$, which can be either discrete or continuous. We will refer to those states which do not emit an observation as *non-emitting* states. As stated above, such non-emitting states are already used in different ways in related models. The version presented in the context of the AHMM generalizes their properties and can be used flexibly for a number of purposes.

As there are many different notation conventions in use for hidden Markov models, here we will utilize a convention we believe will be easy to remember while reading this paper.

As a starting point, we present a traditionally defined hidden Markov model using our notation convention. A hidden Markov model $\lambda_{HMM} = (S, B, T, O)$ is defined by a set of states S, a beginning state probability distribution B, a state transition probability function T, and the observation probability function O. The HMM begins its execution in a state chosen by B, i.e. it will begin in some state $s \in S$ with probability $B(s)$. At each *time step*, the HMM will make a state transition - if it is in a state $s \in S$ it will transition into state $s' \in S$ with probability $T(s, s')$. In each state, an observation is generated according to the observation probability distribution O.

We now define our augmented hidden Markov model (AHMM) $\lambda = (E, N, S_b, S_e, T, O)$ by a set of emitting states E, a set of non-emitting states N (and we denote the union of the two as

$S = E \cup N$), a set of beginning states $S_b \subset N$, a set of ending states $S_e \subset N$, a state transition probability function $T : (S \setminus S_e) \times (S \setminus S_b) \rightarrow [0, 1]$, and an observation probability function O. If $\mathbb{O}$ is discrete, then $O : E \times \mathbb{O} \rightarrow [0, 1]$. If $\mathbb{O}$ is continuous, then O is a probability density function and $O : E \times \mathbb{O} \rightarrow [0, \infty)$. In our discussion, we will place the word density in parenthesis to account for both scenarios - e.g., O is a probability (density) function.

For simplicity, this paper will only deal with the case where there is a single beginning state s_b ($S_b = \{s_b\}$), and either a single ending state s_e ($S_e = \{s_e\}$) or no ending states ($S_e = \emptyset$).

The augmented HMM behaves similarly to a regular HMM, but with a few points of departure. The states are divided into disjoint sets of emitting states E and non-emitting states N. When emitting states are entered, they emit an observation belonging to the observation set $\mathbb{O}$ according to the observation probability function O. Specifically, the probability (density) of state $e \in E$ emitting observation $o \in \mathbb{O}$ is given by $O(e, o)$. Unlike emitting states, non-emitting states do not emit an observation when entered.

Unlike the traditional HMM, the AHMM does not include the beginning state probability distribution in the model definition. Rather, the model always begins in state s_b, and until it arrives to a state in S_e it makes transitions according to the state transition probability function T (therefore, if $S_e = \emptyset$, the model will make transitions indefinitely). T must satisfy that the sum of transition probabilities out of any (non-ending) state is 1. Since only emitting states produce an observation, and we assume that observations are generated/collected at some consistent frame rate, entering an emitting state takes one *time step*, while all other transitions are executed instantaneously. Since the AHMM transitions instantly to non-emitting states, it can visit multiple states in the same time step (but in a well defined sequence). We use X_i to denote the set of states visited in time-step i. For example, we can say that $s_b \in X_0$.

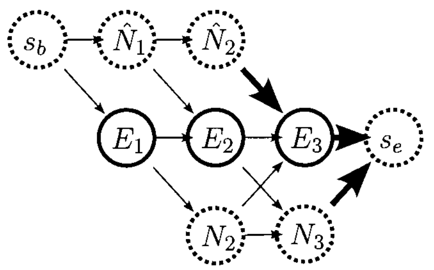

Fig. 2.1. A sample AHMM. Solid circles represent emitting states, and dashed circles represent non-emitting states. Arrows indicate positive transition probabilities. Thin arrows indicate a transition probability of $\frac{1}{2}$, and the thicker ones a transition probability of 1.

Figure 2.1 shows a sample AHMM. There are three emitting states (E_1, E_2, E_3) and six non-emitting states ($s_b, \hat{N}_1, \hat{N}_2, N_2, N_3, s_e$). The transition probabilities are such that for any state s, all non-zero transition probabilities out of s are equal.

We will now note a few definitions and restrictions. Let $S = E \cup N$. Let the (directed) transition graph of the AHMM λ be $G(\lambda) = (G_S, G_E)$ with vertices $G_S = S$ and edges G_E such that $(s, s') \in E_G$ if and only if $T(s, s') > 0$. We call a path $(s_1, s_2, \ldots, s_n)$ inside $G(\lambda)$ where $s_1, s_n \in S$ and $s_2, \ldots, s_{n-1} \in N$ an *internally non-emitting path*. If $s_n \in N$, we also call such a path a *non-emitting path*. If rather $s_n \in E$, we call such a path an *emission-terminating path*.

Please note that in this definition it is possible to have no intermediate nodes in the path, i.e. we can have $n = 2$. In the example from Figure 2.1, (s_b, E_1) is a emission-terminating path and (E_1, N_2, N_3) is a non-emitting path. (s_b, E_1, N_2) is neither (hence, not an internally non-emitting path).

Given $s \in S$ and $e \in E$, define $C^{in}_{s,e}$ to be the set of all internally non-emitting paths connecting s and e. For any such path $c = ((s_1 = s), s_2, \ldots, (s_n = e)) \in C^{in}_{s,e}$, define $p(c) = \prod_{i=1}^{n-1} T(s_i, s_{i+1})$, i.e. the product of T values along the path. Finally, define

$$T^{in}(s,e) = \sum_{c \in C^{in}_{s,e}} p(c) \tag{1}$$

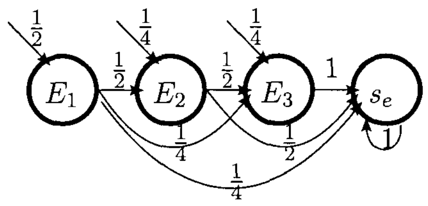

Fig. 2.2. An HMM corresponding to a slight modification of the AHMM in Figure 2.1 (the ending state s_e from Figure 2.1 was changed into an emitting state to allow the conversion). Arrows from one state to another indicate (and are labeled by) transition probabilities, whereas arrows not originating at a state indicate beginning state probabilities.

To be clear, if $C^{in}_{s,e} = \emptyset$ then $T^{in}(s, e) = 0$.

For simplicity, we will restrict AHMMs in this paper in requiring that for any states $s, s' \in S$, there is at most one internally non-emitting path from s to s'. This rule simplifies the discussion by disallowing alternate non-emitting paths between emitting states, as well as disallowing loops composed entirely of non-emitting states.

Note that we can convert an AHMM with no ending state to an equivalent HMM by limiting the states to emitting states. The beginning state probability distribution of the HMM is determined by emission-terminating paths originating in the single AHMM beginning state. The transition probabilities from a particular emitting state are determined by emission-terminating paths originating in that emitting state.

Formally, given an AHMM λ_{AHMM} s.t.

$$\lambda_{AHMM} = (E, N, \{s_b\}, \emptyset, T, O), \tag{2}$$

the equivalent HMM λ_{HMM} is s.t.

$$\lambda_{HMM} = (E, B, T_{HMM}, O), \tag{3}$$

where $B(e) = T^{in}(s_b, e)$ and $T_{HMM}(e, e') = T^{in}(e, e')$ for all $e, e' \in E$.

For example, Figure 2.2 shows an HMM obtained through conversion of a modified version of the AHMM given in Figure 2.1. Since HMMs do not support ending states, before applying the conversion we substitute the non-emitting state s_e shown in Figure 2.1 with an emitting state which always transitions to itself.

2.2.1 Equivalence Classes

To be able to represent certain models as special cases of an AHMM, we need to be able to enforce equality between transition probabilities and / or observation probability distributions. For example, the constant reduced parameter model [43] requires a number of transition probabilities be of the same value. In a semi-Markov model, we will need a number of emitting states to share the same observation probability distribution.

To this end, we supplement the AHMM with equivalence class information, which dictates which transition probabilities / observation probability distributions are constrained to be equal. The equivalence classes are related to the equivalence relation "is constrained to be equal to". Any mutating AHMM algorithm (any algorithm that modifies an AHMM, e.g. a training algorithm), must leave the AHMM in a state that satisfies the constraint. For non-mutating algorithms, such as an inference algorithm, this information is irrelevant. Since inference algorithms do not modify the model, it is meaningless to constrain parts of the model to be equal - they are either equal or they are not.

2.2.1.1 Transition Probability Equivalence

Transition probability equivalence is given over the set of all pairs of states which can have a transition between them. If $s \in S \setminus S_e$ and $s' \in S \setminus S_b$, we denote by $[(s, s')]_T$ the equivalence class of (s, s'). For any $(s'', s''') \in [(s, s')]_T$, $T(s, s')$ is constrained to be equal to $T(s'', s''')$. We denote the set of all transition equivalence classes C_T.

In some cases, no additional transition equivalence class information is needed. In this case, $C_T = \{\{(s, s')\} \,|s \in S \setminus S_e, s' \in S\}$.

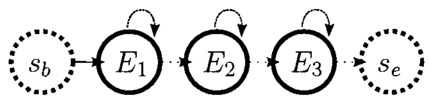

Fig. 2.3. A sample AHMM with transition probability equivalence classes. Transitions drawn with dotted lines form one equivalence class, and transitions drawn with dashed lines form another equivalence class. Transitions within the same equivalence class are constrained to have equal transition probabilities.

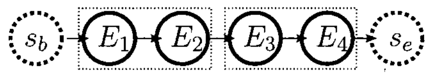

Fig. 2.4. A sample AHMM with observation probability distribution equivalence classes. Each dashed rectangle encloses an equivalence class. States within the same equivalence class are constrained to have equal observation probability distributions.

For an example of an AHMM+EC which uses transition equivalence classes, consider the model in Figure 2.3. There are three equivalence classes: $\{(s_b, E_1)\}$, $\{(E_1, E_1), (E_2, E_2), (E_3, E_3)\}$, and $\{(E_1, E_2), (E_2, E_3), (E_3, s_e)\}$. Any algorithm that modifies transition probabilities, such as a training algorithm, must leave the model in a state where transitions within the same equivalence class have equal transition probabilities.

2.2.1.2 Observation Probability Distribution Equivalence

Observation probability distribution equivalence is given over the emitting state set E. For $e \in E$ we denote by $[e]_O$ the equivalence class of e. For any $e' \in [e]_O$, $O(e)$ is constrained to be equal to $O(e')$. We denote the set of all observation equivalence classes C_O.

In some cases, no additional observation equivalence class information is needed. In this case, $C_O = \{\{e\} | e \in E\}$.

For an example of an AHMM+EC which uses observation equivalence classes, consider the model in Figure 2.4. There are two equivalence classes: $\{E_1, E_2\}$ and $\{E_3, E_4\}$. Any algorithm that modifies observation probability distributions, such as a training algorithm, must leave the model in a state where states within the same equivalence class have equal observation probability distributions.

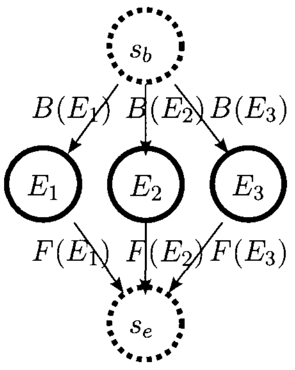

Fig. 2.5. HMM with termination probabilities modeled as an AHMM. Transitions between emitting states are omitted for clarity.

2.3 Special cases of AHMMs

2.3.1 Hidden Markov Model

Let a hidden Markov model be given as follows:

$$\lambda_{HMM} = (E, B, T_{HMM}, O) \tag{4}$$

The HMM is equivalent to the following AHMM:

$$\lambda = (E, \{s_b\}, \{s_b\}, \emptyset, T_{HMM} \cup T_B, O), \tag{5}$$

where T_B is such that $T_B(s_b, e) = B(e)$ for all $e \in E$.

No additional equivalence class information is needed.

2.3.2 Hidden Markov Model with Termination Probabilities

Al-Ohali et al. [2] present an extension of HMM in which each state has a termination probability, and show that such a model can be superior to the regular HMM. Terminating HMMs are particularly suited for modeling patterns of finite sequences of observations, since a non-terminating HMM is fundamentally a generative model that outputs an infinite sequence of observations. While the infinite sequence generated by a non-terminating HMM can be cut off at any point (using criteria external to the model) to give a finite sequence, the terminating HMM encodes termination probabilities directly into the model. Thus, when performing classification of segmented patterns, the terminating HMM is more flexible in modeling the termination of the pattern. Under reasonable assumptions, using a non-terminating HMM to classify patterns of finite length is equivalent to assuming that every emitting state has the same termination probability.

Suppose a terminating HMM is given as follows:

$$\lambda_{THMM} = (E, B, F, T_{THMM}, O), \tag{6}$$

where $F : E \rightarrow [0, 1]$ is the added termination (finishing) probability.

The terminating HMM is then equivalent to the following AHMM (see Figure 2.5 for an illustration):

$$\lambda = (E, \{s_b, s_e\}, \{s_b\}, \{s_e\}, T_{HMM} \cup T_B \cup T_F, O), \tag{7}$$

where T_B is defined as above and T_F is such that $T_F(e, s_e) = F(e)$ for all $e \in E$.

No additional equivalence class information is needed.

2.3.3 Hierarchical Hidden Markov Model

Hierarchical HMMs are well suited for modeling hierarchical patterns. For example, in speech, a sentence is composed of words, a word is composed of phonemes, and a phoneme might be described as a pattern of frequency spectra. Thus, a speech recognition hierarchical HMM might have emitting states that model frequency spectra; a number of such states might form a submodel that represents a phoneme (submodels are typically referred to as abstract states in hierarchical HMMs); a number of phoneme submodels might form a submodel that represents a word; and finally a number of word submodels might form the overall model that represents a sentence. This approach to modeling is helpful both conceptually, since the model structure reflects the pattern structure, and practically: using submodels can help with training, since it allows training examples

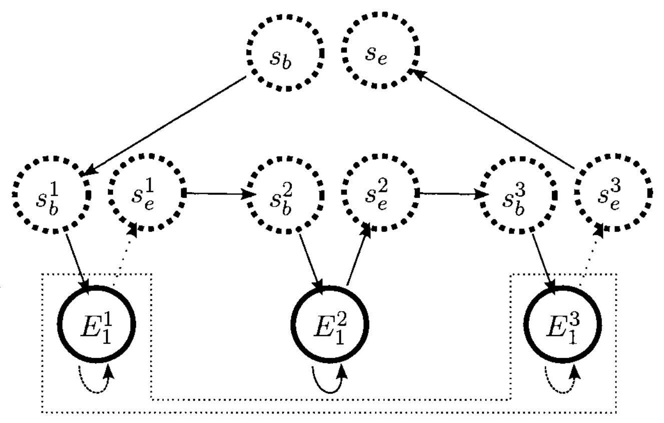

Fig. 2.6. HHMM representing the word "bob" modeled as an AHMM. Each subcomponent has a single begin state, a single end state, and for simplicity, a single emitting state. The resulting AHMM also has a single begin and single end state, so it could be used as a subcomponent of a larger hierarchical model. The two subcomponents connected by the dotted region both represent "b", and are constrained to be equal through equivalence classes during training. This applies to both the observation probability distribution, and the corresponding transition probabilities.

for, e.g., a particular phoneme to be gathered from multiple words, which helps increase the amount of training data per state. Also, submodels of hierarchical HMMs explicitly encode termination probabilities for the submodel (but there are no termination probabilities for the overall model).

A hierarchical hidden Markov model (HHMM) can be constructed by combining component AHMMs that have a single begin state and a single end state. For example, let a set of such component AHMMs $\Lambda = \{\lambda^i\}$ be given as follows:

$$\lambda^i = (E^i, N^i, \{s_b^i\}, \{s_e^i\}, T^i, O^i) \tag{8}$$

We can construct a higher-level AHMM λ that uses λ^i as submodels, by using with their begin and end states as their interface. λ will have its own begin and end states, s_b and s_e.

To specify the higher-level behavior of λ, we define a partial transition function $T' : (\cup_i \{s_e^i\} \cup \{s_b\}) \times (\cup_j \{s_b^j\} \cup \{s_e\}) \to [0, 1]$, where:

- $T'(s_b, s_b^i) = p$ means that λ will begin execution with λ^i with probability p.

- $T'(s_e^i, s_b^j) = p$ means that after completion of λ^i, λ will begin execution of λ^j with probability p.
- $T'(s_e^i, s_e) = p$ means that after completion of λ^i, λ will end its execution with probability p.

The overall AHMM λ can now be formally given as:

$$\lambda = (E, N, S_b, S_e, T, O) \text{ , where} \tag{9}$$

$$E = \cup_i E^i \tag{10}$$

$$N = \cup_i N^i \cup \{s_b, s_e\} \tag{11}$$

$$S_b = \{s_b\} \tag{12}$$

$$S_e = \{s_e\} \tag{13}$$

$$T = \cup_i T^i \cup T' \tag{14}$$

$$O = \cup_i O^i \tag{15}$$

Since the final result is an AHMM with a single begin and single end state, it can be combined with other such AHMMs in a higher level of hierarchy.

The above description of constructing hierarchical Markov models using AHMMs assumes that the component AHMMs are distinct. Suppose we are constructing a hierarchical model where multiple component AHMMs are modeling the same pattern. For example, suppose we are planning to model the pronunciation of the word "bob". We could choose to model the first and last pronunciation of "b" using the same AHMM submodel.

In this case, we would apply the above procedure to three component AHMMs: λ^1 (modeling "b"), λ^2 (modeling "o"), and λ^3, where λ^3 is a copy of λ^1 (that is, there is an isomorphism between the two that preserves transition probabilities and observation distribution probabilities).

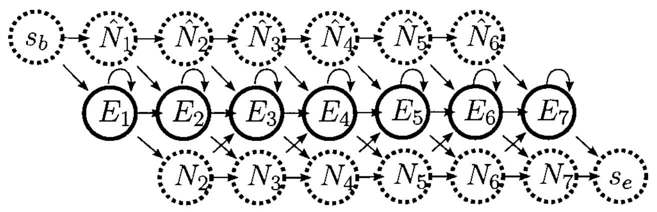

Fig. 2.7. Transitions for the reduced parameter models with 7 emitting states. The emitting states ($E_1 \dots E_7$) produce the gesture observations. Non-emitting states $\hat{N}_1 \dots \hat{N}_6$ serve to skip emitting states immediately after the gesture AHMM has started its execution in state s_b. Non-emitting states $N_2 \dots N_7$ serve to skip emitting states after an emitting state has been entered. The displayed transitions are determined differently for each of the reduced parameter models.

Additionally, we would specify equivalence classes between the corresponding states and transitions. For any $e^1 \in E^1$ and its corresponding copy $e^3 \in E^3$, $[e^1]_O = \{e^1, e^3\}$. For $s^1, s'^1 \in E^1$ and their corresponding copies $s^3, s'^3 \in E^3$, $[(s^1, s'^1)] = [(s^1, s'^1), (s^3, s'^3)]$. The equivalence classes would preserve the equality constraints during training or any other mutating algorithms. Figure 2.6 illustrates this model, assuming a single emitting state per component AHMM.

In general, suppose a set of component AHMMs $\Lambda = \{\lambda^i\}$ is given. Let a subset of isomorphic components AHMMs $\{\lambda^k | k \in I\}$ defined by the index set $I = \{k_1, k_2, \dots\}$ be given, where each component λ^k models the same subcomponent of the pattern. Finally, suppose $\lambda^k = (E^k, N^k, S_b^k, S_e^k, T^k, O^k)$. Then there would be $|E^{k_1}|$ observation equivalence classes, each with $|I|$ states (one from each isomorphic component). Similarly, there would be $|T^{k_1}|$ transition equivalence classes, each with $|I|$ state pairs (one pair from each isomorphic component).

2.3.4 Reduced Parameter Model

The reduced parameter model was originally presented in [43]. Its state and transition structure are depicted in Figure 2.7. The model provides an emission-terminating path from an emitting state E_i to any emitting state E_j, $j \geq i$, but at the same time allows an efficient computational complexity of inference algorithms ($\Theta(|E|)$ per time step). The resulting structure is a compromise between a HMM which allows the same transitions but imposes a higher computational complexity of inference ($\Theta(|E|^2)$ per time step), and HMMs with a chain-like topology (e.g., the ones shown in Figures 2.12 and 2.13), which offer the same low computational complexity but disallow transitions that skip more than a certain number of states.

In the reduced parameter model, the transition probabilities are labeled by the following symbols (we highlight in bold italics the letter that might help you remember the symbol due to visual similarity):

- τ_i is the probability of ***r***emaining in an emitting state ($T(E_i, E_i)$)
- η_i is the probability of going to the ***n***ext emitting state ($T(E_i, E_{i+1})$)
- ς_i is the probability of ***s***kipping at least one emitting state ($T(E_i, N_{i+1})$)
- κ_i is the probability of s***k***ipping an additional emitting state ($T(N_i, N_{i+1}) = T(\hat{N}_i, \hat{N}_{i+1})$)
- ρ_i is the probability of ending a ski***p*** sequence ($T(N_i, E_{i+1}) = T(\hat{N}_i, E_{i+1})$)

In these expressions, s_b would be treated as $\hat{N}_0$ and s_e as $N_{|E|+1}$. Note that transition equivalence classes would be established between two instances of the κ_i probabilities for each i, as well as the two instances of the ρ_i probabilities for each i.

The parameters pertaining to a local set of emitting / non-emitting states is shown in Figure 2.8. Note that given these parameters for each state, the probabilities of transition paths between emitting states are given as follows:

$$T^{in}(E_i, E_j) = \begin{cases} 0, & \text{if } j < i \\ \tau_i, & \text{if } j = i \\ \eta_i, & \text{if } j = i + 1 \\ \varsigma_i \prod_{k=i+1}^{j-2} \kappa_k \rho_{j-1}, & \text{if } j > i + 1 \end{cases} \tag{16}$$

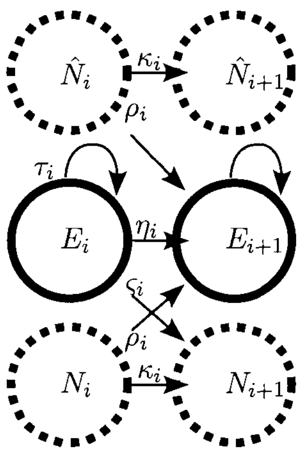

Fig. 2.8. Transition probability parameters for the *reduced* model.

2.3.5 Constant Parameter Model

The constant parameter model uses a constant number of parameters to determine transition probabilities between all states, regardless of the number of states in the model. It is an adaptation of the model presented in [38]. The states and transitions can again be seen in Figures 2.7 and 2.8, but the transition probabilities are determined slightly differently than with the reduced parameter model. In particular, each of the 5 parameters is constrained to have the same value for almost every state - i.e., $\tau_i = \tau$ for some value τ and all appropriate i, $\eta_i = \eta$ for some value η and all appropriate i, etc. By "all appropriate i", we mean all i with the exception of a few border cases. In the example AHMM shown in Figure 2.7, κ_7 is not necessarily equal to κ_6, because $\kappa_7 = 1$ while $\kappa_6 + \rho_6 = 1$ (since transition probabilities out of a state must sum to 1). The transition equivalence classes are established correspondingly. The border cases do not belong to the equivalence classes.

The probabilities of transition paths between emitting states are now given by:

$$T^{in}(E_i, E_j) = \begin{cases} 0, & \text{if } j < i \\ \tau, & \text{if } j = i \\ \eta, & \text{if } j = i+1 \\ \varsigma\kappa^{j-i-2}\rho, & \text{if } j > i+1 \end{cases} . \qquad (17)$$

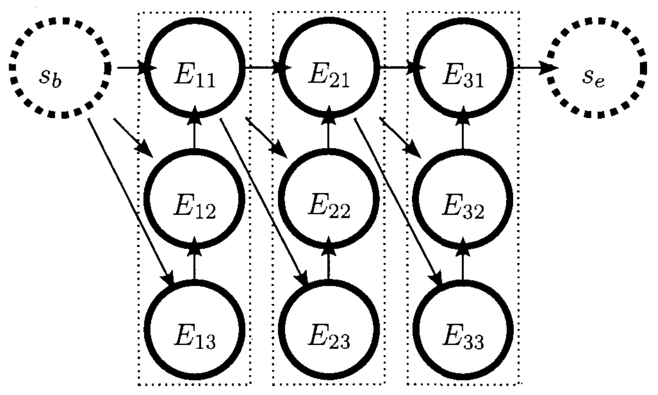

Fig. 2.9. A semi-Markov model with three groups of emitting states. Each group, indicated by the square dashed line, is constrained to have the same observation probability distribution through equivalence classes.

The benefit of the constant parameter model is that it shares training data among all transition probabilities, so that the resulting transition probabilities around each state are influenced by transitions inferred from the training data around every state. This can be helpful if the training data is limited [38].

2.3.6 Hidden Semi-Markov Model

In a semi-Markov model, the probability of transitioning between a state s and a state s' depends on the number of times state s has been consecutively visited immediately prior to the transition to s'. Thus, the probability of staying in a state can change based on how long the model stays in that state. In the context of patterns, staying in a state is usually analogous to staying in a particular segment or phase of the pattern, so the number of times a state is visited consecutively corresponds to the length of time spent in that portion of the pattern. For example, when saying the word "Wow", the speaker may spend a fraction of a second on the vowel, or several seconds. Semi-Markov models allow more precise modeling of the lengths of time that are probable for any given state. For example, a semi-Markov model could specify high probabilities for moderate lengths of time spent on the vowel in "Wow", while specifying very low probabilities for uncommonly short or uncommonly long lengths of time. A regular HMM can only represent a geometric probability distribution over the length of time (more precisely, over the number of consecutive times a state is visited).

If the number of times a state can be consecutively visited is bounded by m, a semi-Markov model can be represented by a regular HMM in which each state of the semi-Markov model is replaced by m states that share their observation probability distribution. Representing such a semi-Markov model with an AHMM has the added benefit of being able to define equivalence classes for the observation probability distributions.

Suppose there are n semi-Markov model states, and m is the maximum number of times a state can be visited consecutively. The set of emitting states E of our AHMM is then defined as $\{e_{ij}\}$ where $1 \leq i \leq n$ and $1 \leq j \leq m$. The overall model is defined as:

$$\lambda = (E, \{s_b\}, \{s_b\}, \emptyset, T, O) \tag{18}$$

$O(e_{ij})$ is constrained to be equal to $O(e_{ik})$ for all j, k. We make use of observation equivalence classes $[e_{ij}]_O = \cup_k e_{ik}$. Figure 2.9 shows an example.

2.3.7 Larger Order Hidden Markov Models

In a regular HMM, the probability of the next state is completely determined by the most recent (current) state. In an HMM of order n, the probability the next state is determined by the n most recent states. The benefit is that dependencies between state visitations can be expressed across wider temporal spans (rather than only between adjacent state visitations). This can be useful, for example, with gestures where the way a gesture is completed is influenced by the way the gesture was started.

A higher order HMM can be represented by a regular HMM (of order 1) by using Cartesian products of states. For example, if E^2 is the set of emitting states of an HMM of order 2, then we

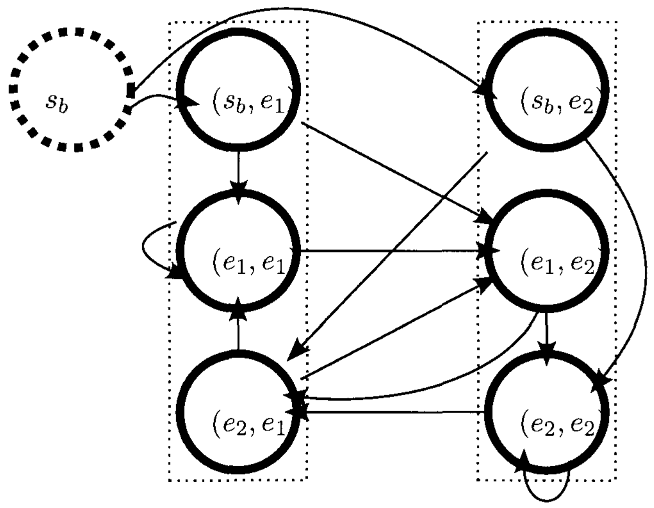

Fig. 2.10. A 2-state, second order HMM converted to an AHMM. Each of the two groups of emitting states, indicated by the dashed square, represents one of emitting states of the HMM (e_1, e_2). In the AHMM, each group of emitting states is constrained to have the same observation probability distribution through equivalence classes.

could represent the model in a regular HMM by using a set of states $E = (E^2 \times E^2)$. Being in state $(e_i^2, e_j^2) \in E$ is equivalent to being in state $e_j^2 \in E^2$ immediately after visiting $e_i^2 \in E^2$.

With an AHMM, we can define equivalence classes that correctly perform training. In the above example, $[(e_i^2, e_j^2)]_O = \cup_k(e_k^2, e_j^2)$. Figure 2.10 shows an example.

2.4 AHMM Inference and Training

The following subsections will provide an overview of the inference (Sections 2.4.1 and 2.4.2) and training (Section 2.4.3) algorithms used for the AHMM+EC.

2.4.1 AHMM Viterbi Inference

One commonly addressed problem in the context of HMMs is one of state estimation: given a sequence of observations $o_1, o_2, \ldots, o_n$, what is the most likely sequence of states to have produced it? At the same time, we would like to know the likelihood of the observations being produced by the most likely state sequence.

To solve the stated problem, we adapt the Viterbi algorithm, which is the usual way of finding the most likely state sequence in traditional HMMs. Formally, given an AHMM $\lambda =$

(E, N, S_b, S_e, T, O) and a particular sequence of n observations $o_1, o_2, \ldots, o_n$, we seek to find the most likely state sequence $s_1, s_2, \ldots, s_m$ of states $s_i \in E \cup N$ that would have produced it (since only emitting states emit observations, $m \geq n$ and there are exactly n emitting states in the state sequence).

The Viterbi algorithm uses a dynamic programming variable $\delta_i(s)$, which captures the probability of partial state sequence generating a partial observation sequence. We specify the partial state sequence in terms of the set of states visited at each time step:

$$\delta_i(s) = \max_{x_{1:i-1}} p\left(X_1 = x_1, X_2 = x_2, \ldots, X_{i-1} = x_{i-1}, s \in X_i, o_{1:i} | \lambda\right) \tag{19}$$

$\delta_i(s)$ represents the likelihood of the most likely state sequence ending in s to have generated the observations $o_{1:i}$. For $s \in E$, this assumes that state s generated observation o_i. For $s \in N$, it is the highest possible value that can be obtained by starting with $\delta_i(e)$ at an emitting state $e \in E$, and multiplying the values along a non-emitting path in $C^{in}_{e,s}$, in which case it assumes e generated observation o_i. This can be written as $\delta_i(s) = \max_{e,c \in C^{in}_{e,s}} \delta_i(e)p(c)$.

δ values for a state are calculated from δ values among all states that we can transition from:

$$\delta_i(s) = \begin{cases} \max \delta_{i-1}(s')T(s', s)O(o_i, s) & \text{if } s \in E \\ \max \delta_i(s')T(s', s) & \text{if } s \in N \end{cases} \tag{20}$$

From (20), we can see that the δ values should be calculated in a certain order. Specifically, before calculating $\delta_i(s)$ for emitting states $s \in E$ we should have calculated $\delta_{i-1}(s')$ for all states $s' \in S$. Also, before calculating $\delta_i(s)$ for some $s \in N$, we should have calculated $\delta_i(s')$ for each $s' \in E \cup N$ such that $T(s', s) > 0$.

The order of calculation for a particular value of i is given by a topological sort of $G(\lambda)$ limited to non-emitting paths. Given such a topological sort, the algorithm is simple. To initialize, we set $\delta_0(s_b) = 1$. The remaining δ_0 values for all states that are reachable from s_b by non-emitting paths can then be calculated using (20) in the order of the topological sort. For example, in the AHMM shown in Figure 2.1 the order would be $\hat{N}_1, \hat{N}_2$. For any state s' not reachable from s_b through non-emitting paths, $\delta_0(s') = 0$. Given each $o_i, i = 1, 2, \ldots$, we can first calculate δ_i values for all emitting states (in any order), and then the δ_i values for non-emitting states (reachable from emitting states) in the order of the topological sort. For the AHMM shown in Figure 2.1, the order would be N_2, N_3, s_e.

To extract the most likely state sequence, we also keep track of the optimal route taken to calculate the δ values. We define the *traceback* function, denoted by δ^*, as follows:

$$\delta_i^*(s) = \begin{cases} (\arg\max_{s'} \delta_{i-1}(s')T(s', s)O(o_i, s) \ , & \\ \quad i-1) & \text{if } s \in E \\ (\arg\max_{s'} \delta_i(s')T(s', s) \ , & \\ \quad i) & \text{if } s \in N \end{cases} \tag{21}$$

These δ^* values allow us to perform a traceback from any state and recover the most likely state sequence ending with that state. At any time i, the overall most likely state sequence ends with state s given by $\arg\max_s \delta_i(s)$. If $\delta_i^*(s) = (s', i')$, then the previous state in the sequence is s', and is visited at time i'. We can then continue the traceback by examining $\delta_{i'}^*(s')$, etc.

2.4.2 Forward and Backward Algorithms

Forward and backward algorithms are used to find the probability (density) of an observation sequence given a model. They are adapted similarly to the Viterbi algorithm.

Now, consider the forward variable α and the backward variable β defined as:

$$\alpha_i(s) = p(o_{1\ldots i}, s \in X_i|\lambda) \tag{22}$$

$$\beta_i(s) = p(o_{i+1\ldots n}|s \in X_i, \lambda) \tag{23}$$

Intuitively, $\alpha_i(s)$ represents the probability (density) of $o_{1\ldots i}$ being generated by the model λ and that the execution reaches s in the final time step. $\beta_i(s)$ represents the probability (density) of $o_{i+1\ldots n}$ being generated by λ given that the execution has reached s in the previous time step.

The adaptation is similar to that of the Viterbi algorithm. α is calculated as follows:

$$\alpha_0(s_b) = 1 \tag{24}$$

$$\alpha_i(s) = \begin{cases} \sum \alpha_{i-1}(s')T(s',s)O(o_i,s) & \text{if } s \in E \\ \sum \alpha_i(s')T(s',s) & \text{if } s \in N \end{cases}. \tag{25}$$

For β, the initialization depends on whether we are assuming that the model terminated after the observation sequence. If we are, then we initialize with $\beta_n(s_e) = 1$. If not, $\beta_n(e) = 1$ for all $e \in E$. In either case, the remaining values are calculated using

$$\beta_i(s) = \sum_{s' \in E} \beta_{i+1}(s')T(s,s')O(o_{i+1},s') + \sum_{s' \in N} \beta_i(s')T(s,s'). \tag{26}$$

We can use forward and backward variables to extract higher level information about the pattern and the model. For example, the probability that observations $o_1, o_2, \ldots, o_n$ are the first n observations produced by the model λ is given by $\sum_{e \in E} \alpha_n(e)$.

If $S_e = \{s_e\}$, then the probability that λ emits $o_1, o_2, \ldots, o_n$ and then terminates is given by $\alpha_n(s_e)$.

2.4.3 TRAINING

The purpose of training an AHMM model is to determine the exact parameters defining the model (E, N, S_b, S_e, T, and O). Training of models that represent a single pattern is usually performed using a number of examples of the pattern (training data).

The state structure (E, N, S_b, S_e) of the AHMM is determined first, as are the the possible transitions (transitions for which $T(s, s')$ is allowed to be non-zero). This is either determined a priori using domain knowledge, or determined as some function of the training data (for example, we might choose that the number of emitting states should equal the minimum, average, or maximum number of observations in the gesture examples).

The model is initialized using reasonable initial values. Training of the state transition probability function T and the observation probability distribution function O is then performed using the expectation-maximization (EM) algorithm.

The general outline of the EM algorithm is as follows. We are given the initialized model as well as the training dataset. The training dataset $O^1, O^2, \ldots$ consists of examples of the pattern, where each example O^d is a sequence of observations $o_1^d, o_2^d, \ldots$, with each $o_i^d \in \mathbb{O}$. Each training sample O^d is then evaluated using the forward and backward algorithms. That is the expectation step. The forward and backward variables are then used to revise the transition and observation distribution probabilities in the model. That is the maximization step.

2.4.3.1 Transition Probability Estimation

The basics of the EM algorithm for AHMMs is essentially the same as for a regular HMM, with slight modifications to take into account non-emitting states. To revise the transition probability between a state s and an emitting state e (respectively, a non-emitting state n), we first compute the expectation of a transition between the two states as

$$\begin{aligned} P_T(s,e) &\propto \sum_d \left(\frac{1}{P(O^d)} \times \right. \\ &\qquad \left. \sum_i \alpha_i(s) T(s,e) O(e, o_{i+1}) \beta_{i+1}(e) \right) \qquad (27) \\ P_T(s,n) &\propto \sum_d \frac{1}{P(O^d)} \sum_i \alpha_i(s) T(s,n) \beta_i(n) \qquad (28) \end{aligned}$$

Note that proportionality is sufficient as transition probabilities out of a state must sum to 1.

$P(O^d)$ is estimated using the forward algorithm. We then calculate the expectation that a transition belonging to a particular equivalence class $c \in C_T$ is taken:

$$T_c \propto \frac{1}{d} \sum_d \left(\sum_{(s,s') \in c} P_T(s,s') \right) \qquad (29)$$

The transition probability $T(s, s')$ for all $(s, s') \in c$ is then set to T_c in the maximization phase of the algorithm.

2.4.3.2 Observation Probability Distribution Estimation

Training observation probability distributions is also similar to how it is performed in a regular EM algorithm, as it only applies to emitting states. Each observation distribution is trained from a weighted set of observations, where the weight of each observation is the probability that the observation was produced by a state:

$$P_O(s,o) \propto \sum_d \left(\frac{1}{P(O^d)} \sum_{\{i | o_i^d = o\}} \alpha_i(s) \beta_i(s) \right) \qquad (30)$$

We then calculate the expectation that an observation is produced by a state belonging to a particular equivalence class $c \in C_O$:

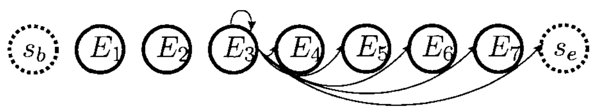

Fig. 2.11. Transitions for a standard left-to-right HMM model with 7 emitting states. Only transitions out of one state (E_3) are displayed for clarity. In general, any state can have a non-zero probability specified for transitions to itself or to any state depicted to its right.

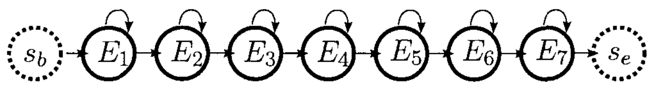

Fig. 2.12. A chain HMM with 7 emitting states. Each state in the chain can transition to itself or the next state. The chain HMM is one of the topologies tested in the experiments.

$$O_c(o) \propto \frac{1}{d}\sum_{d}\left(\sum_{i,s\in c} P_O(s,o)\right) \tag{31}$$

The observation probability distribution for each state $s \in c$ is then adjusted using all of the observations and their weights $O_c(o)$. The details depend on how the observation distribution is modeled. If $\mathbb{O} = \mathbb{R}$ and we model each observation distribution by a univariate normal distribution, then the mean of the distribution might be set to the weighted mean of the observations, and the variance to the weighted variance of the observations.

2.5 Experimental Results

We have implemented a generic C++ library (see Chapter 4) providing data types and algorithms related to AHMM+ECs presented in this chapter. The library is available in the open source AMELiA distribution [40] under the terms of the GNU General Public License. The code for the experiments described below, as well as the datasets used, are also present in the distribution.

We have tested the implementation on several publicly available data sets. Insofar as the results obtained using our implementation replicate previously published results, they support the validity of the implementation. Furthermore, the ability to easily evaluate a number of models using the same implementation in some cases allows us to expand on several previously published results.

The models tested in the experiments are:

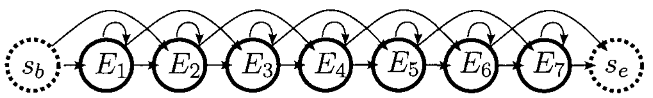

Fig. 2.13. A chain+skip HMM with 7 emitting states. Each state in the chain can transition to itself or the next two states. The chain+skip HMM is one of the topologies tested in the experiments.

- *constant* - the constant parameter model
- *reduced* - the reduced model
- *l-to-r* - a standard left-to-right HMM (see Figure 2.11)
- *chain* - a chain HMM (see Figure 2.12)
- *skip* - a chain+skip HMM (see Figure 2.13)
- *semi* - hidden semi-Markov model with two emitting AHMM states per semi-Markov model state.

Note that the computational complexity of each model with respect to training and inference algorithm is determined by the order of incoming transitions to each state [43]. The *constant*, *reduced*, *chain*, *skip*, and *semi* models all have $\Theta(1)$ incoming transition per state. The standard l-to-r model has $\Theta(|E|)$ incoming transitions per state.

The overall goal of our experiments is to establish that the AHMM provides a framework in which a large number of models can easily be tested, and that the ability to do so is useful. We do not aim to argue for or against the usefulness of any of the models or algorithms tested - each has already been proposed and evaluated in research literature.

2.5.1 Runtime Analysis

To provide a fuller understanding of the various models tested, we begin with their runtime analysis. The results presented in Figure 2.14 show the total time taken to train and test each model on a particular dataset, using various numbers of emitting states. The dataset used will be presented in detail in Section 2.5.4. It's details are not important for this runtime analysis - it was chosen because the experiment involved the greatest variety in numbers of emitting states. The experiment itself involved training the models from training data, and then invoking the Viterbi algorithm on the testing data. The time listed includes combined training and testing runtime.

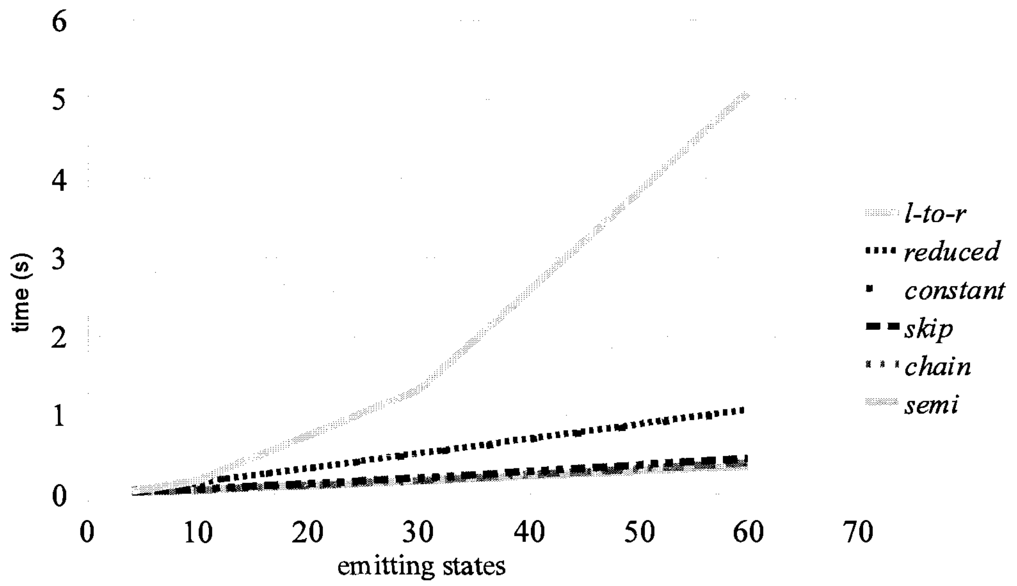

Fig. 2.14. Runtime comparison for various AHMM+EC models tested. The models are listed in the legend on the right in the decreasing order of their overall runtime cost.

The results show that the *l-to-r* model is the most costly, and features a quadratic increase in runtime as the number of emitting states increases. The rest of the models are more efficient, and show a linear increase in runtime. The quadratic and linear increases are as predicted by the complexity of the inference algorithms, which is the dominant factor in both the training and testing.

Among the more efficient models, the *reduced* and *constant* models are more costly, which is caused by the additional non-emitting states used (in general, for each emitting state in these models there are two non-emitting states). The *skip*, *chain*, and *semi* models offer nearly identical runtime, with slight variation due to the slightly fewer transitions in each consecutive model.

TABLE 2.1

Recognition rate (in %) on the Japanese Vowel speaker identification dataset. All tested models (using 5 emitting states) achieve approximately the same accuracy as the results originally presented in [21], which report the recognition rate on a 5-state HMM as 96.2%.

constant	*reduced*	*l-to-r*	*chain*	*skip*	*semi*
97.0	96.2	97.6	97.0	97.6	96.8

While we only present the runtime results for this particular experiment, the runtime performance of the models is similar in all of the experiments. This is because all of the experiments use the same underlying types of algorithms.

2.5.2 JAPANESE VOWELS

The first experimental results we present are on the *Japanese Vowels* dataset [21], available through the UCI Machine Learning Repository [3]. The dataset contains 640 recordings of a Japanese vowel, spoken by nine male speakers. Each recording is a time series of 12 LPC cepstrum coefficients. The classification task is speaker identification.

In our experiment, we replicate the setup from [21], using 270 recordings (30 per speaker) for training, and the remainder for testing. Table 2.1 shows the results obtained. We obtain a 97.6% recognition rate using our 5-state HMM (*l-to-r*), compared to 96.2% originally reported in [21]. While the small variation could be due to slight variations in training or inference strategies, or numerical precision, the relative consistency of reported results supports the validity of our model and implementation.

2.5.3 AUSTRALIAN SIGN LANGUAGE

The Australian sign language *Auslan2* dataset [16] is also available through the UCI Machine Learning Repository [3].

TABLE 2.2

Recognition rate (in %) on the auslan2 dataset. With a minimum observation distribution variance of 0.0001, most models perform worse than those generated by the STACS and V-STACS state-splitting algorithms presented in [49], while the *semi* model outperforms them at 55 emitting states and the *chain* and *skip* models do comparably well at 12 and 55 emitting states, respectively. When the minimum variance is increased to 0.1, all models except for the *chain* model at 55 states perform comparably to the state-splitting algorithms. The results for STACS and V-STACS are taken directly from [49], and use a minimum variance of 0.1.

minimum variance	emitting states	*constant*	*reduced*	*l-to-r*	*chain*	*skip*	*semi*	STACS	V-STACS
0.0001	12	49.1	62.8	64.6	91.9	76.1	85.6		
0.0001	55	66.3	64.9	69.1	64.2	92.3	97.9		
0.1	12	94.7	92.6	90.9	96.1	93.7	93.0	90.9	
0.1	55	95.1	94.7	95.1	64.9	92.0	95.8		95.8

Our experimental setup mimics the one by Siddiqi et al. [49], where a pair of novel state-splitting algorithms are used to learn the topology of the HMM used for recognition. The two algorithms, STACS and V-STACS, achieve a 90.9% and 95.8% (respectively) word recognition rate. The models found by the algorithms have an average number of 12 and 55 (respectively) states per gesture model. The topology of the HMMs, as well as the number of states, is found completely autonomously. While this is a very desirable property of the algorithms, they also come with considerable cost in time complexity.

Table 2.2 shows the results obtained on the 6 experimental models. With the tested models, the topology is set, and the number of emitting states is set explicitly. Note that the results are impacted significantly by the minimum observation distribution variance used in the experiment (this is a parameter often used in the training of observation distributions, mostly to avoid numeric problems stemming from low variances). With a minimum observation distribution variance of 0.0001, most models perform worse than those generated by the STACS and V-STACS state-splitting algorithms presented in [49], while the *semi* model outperforms them at 55 emitting states and the *chain* and *skip* models do comparably well at 12 and 55 emitting states, respectively. When the minimum variance is increased to 0.1, all models except for the *chain* model at 55 states perform comparably to the state-splitting algorithms.

This example shows how a thorough evaluation of a number of models can match the results of a much costlier state-splitting algorithm. In this case, we find that the dataset is adequately modeled by the simple *chain* model for a low number of emitting states, regardless of the minimum variance parameter. We also find that the *skip* model also performs well for a larger number of emitting states. Finally, the more complicated *semi* model works well overall.

In this experiment, we evaluated the algorithms using numbers of emitting states that were found by the STACS and V-STACS algorithm. In the absence of these numbers, we would have had to choose the number of emitting states, perhaps arbitrarily. A more detailed evaluation has shown that the performance of the models interpolates between the numbers of emitting states shown here, so many choices would have also yielded good performance by the *chain*, *skip*, and *semi* models.

This particular dataset is such that a number of models performs well on it - as well as a costly state-splitting algorithm. The thorough evaluation is enabled by the AHMM+EC framework which makes it easy to quickly evaluate a number of models. In other scenarios, the evaluation may not reveal a consistently well-performing model or topology. In that case, autonomous topology-finding algorithms such as STACS or V-STACS may still need to be employed.

2.5.4 Arc Gesture Discrimination

The *tangible object dataset* consists of collected gesture data of a tangible object (ball) moving in a 3D space, as well as synthesized ball data. The collected gesture data consisted of 100 trials of the simple arcing gesture shown in Figure 2.15, with the position of the ball provided by a 6-camera tracking system at 60 frames per second (resulting in $\approx$ 50-80 frames per trial). The 100 trials were

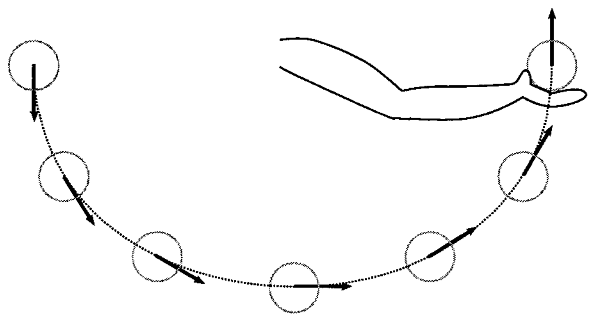
Fig. 2.15. Movement of the example arc gesture.

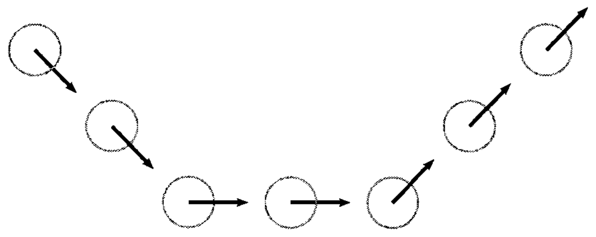
Fig. 2.16. Synthesized gesture similar to the gesture shown in Figure 2.15, also used for experimental results.

divided into 30 training and 70 testing trials. In addition, we synthesized the same number of trials of a gesture similar to the captured gesture, which is shown in Figure 2.16.

In this experiment, we test each the models' ability to discriminate between a gesture it has been trained on, and a similar non-gesture movement. We again turn to the tangible dataset, where we will use the collected gesture as the *training gesture* and the synthesized gesture as the *similar non-gesture*.

Our experiment was as follows. Each model was trained using a variable number of training trials of the training gesture, and then tested on the testing trials of both the trained gesture and the similar non-gesture. The test would attempt to classify the testing trial as either the trained gesture or a non-gesture. The discrimination was accomplished by using a threshold value - if a testing trial resulted in a likelihood (given by the gesture model) higher than the threshold, it was classified as the trained gesture. Otherwise, it was classified as non-gesture movement. The threshold was

TABLE 2.3

Recognition rate (in %) on the arc gesture dataset. The simple chain+skip model performs as well as the *reduced* model, and with the same computational complexity.

emitting states	*constant*	*reduced*	*l-to-r*	*chain*	*skip*	*semi*
4	43.1	43.1	43.1	43.1	43.1	41.5
6	43.8	43.8	43.8	43.8	43.8	42.3
10	45.4	45.4	40.8	45.4	44.8	44.6
30	71.5	87.7	83.8	88.5	85.4	47.7
60	58.5	86.2	87.7	50.0	85.4	64.6

in each case determined automatically from by the minimum likelihood given by the model for all training gesture trials.

This dataset was originally used in our prior publication [43]. The results of that paper indicated that a large number of states was necessary to adequately discriminate between the training gesture and the synthesized gesture. This finding is supported by the results shown in Table 2.3.

However, we previously [43] went on to conclude that the *constant* and *reduced* models are the only models to combine good discrimination results with a low computational complexity. Among the models tested in [43], this was true, but the results in Table 2.3 show that the *skip* model also yields good discrimination results with a low computational complexity. This is another example of how a more holistic model comparison can yield a more optimal model for a given problem domain or data set (as the *skip* model is conceptually simpler than the *reduced* or *constant* models). Again, the ease of testing a number of models is enabled by the AHMM+EC framework.

2.5.5 Video Gesture Classification

The final dataset we examine is the *video gesture dataset,* obtained from video recordings of full body gestures. The data for the gestures was recorded using two video cameras, with each stereo frame converted to a feature vector (using the method described in [36]) for use by the gesture

TABLE 2.4

Recognition rate (in %) on the video gesture dataset. The chain+skip model outperforms all other models, and is as or more computationally efficient than all other models.

constant	*reduced*	*l-to-r*	*chain*	*skip*	*semi*
69.6	71.1	71.1	56.5	72.7	71.1

recognition algorithm. The dataset contains 20 gestures with 24 recorded trials each. The feature vector at each frame has 20 elements, with a typical gesture lasting 20-40 frames. Twelve trials were used for training, and twelve for testing.

The results shown in Table 2.4 indicate that the video gesture dataset is another scenario in which the simpler *skip* model does as well as the *constant* and *reduced* models and at the same computational cost, similarly to the results presented in Section 2.5.4.

2.6 Conclusions

We have presented the augmented hidden Markov model with equivalence classes (AHMM+EC), along with its associated training and inference algorithms. We have also shown how a number of existing state-based probabilistic generative models can be cast as special cases of AHMM+EC. Each of the special cases can be used directly with any of the AHMM+EC algorithms, without the need to adapt the algorithm to the model. We have implemented the models and algorithms into an open source generic C++ library, and presented a number of results using the implementation.

We believe that the AHMM+EC as a unifying framework provides an effective way of comparing various model and equivalence topologies. The experimental results support this claim by presenting several scenarios in which a broader comparison of candidate models provides a more holistic understanding of their comparative performance.

The ease at which different models can be evaluated in the AHMM+EC framework can benefit the research community through a better understanding of the applicability of each model to various problem domains. It can also benefit application developers by easing the selection of the most appropriate model for a particular problem domain.

3. SEMANTIC NETWORK MODEL

3.1 INTRODUCTION

Since gestures and speech are a natural means of human communication, gesture and speech recognition is becoming an increasingly important part of innovative computer applications. Compared to usual methods of human-computer interaction, such as mouse clicks and keystrokes, gesture and speech-based methods provide a much more intuitive and natural interface, contributing to an immersive experience. For example, a user could communicate with a system using learned hand gestures [22], sign language [56, 57], or even gestures expressed using the mouse, as offered by commercially available packages [46, 34], computer games [52] and web browsers [35].

Many speech and gesture recognition applications use hidden Markov models or one of their many variations for the underlying pattern recognition tasks. Such models have been studied in depth and provide the necessary algorithms for efficient training and inference, and can offer adequate accuracy. They are well suited for detecting patterns which are exemplified by relatively consistent sequences of *observations*. They can detect patterns such as a particular movement of a person's body (e.g., a baseball umpire's safe signal), the inscription of a symbol using a pen (e.g., a letter written in cursive), a particular pattern of motion of a stadium crowd (e.g., a crowd "wave"), or a melody (e.g., the Liberty Bell March).

For example, a hierarchical hidden Markov model (HHMM) consisting of a number of submodels (e.g., each modeling a word) can process a sequence of observations (e.g., an appropriately pre-processed audio signal), and segment it so that it is known which submodels most likely correspond to the observations (e.g., which words are present in the audio signal). This is typically done using the Viterbi algorithm, which produces the most likely correspondence between the states of the model and the observations.

However, in some cases the full correspondence between the states and observations is more information than is needed. Applications are often concerned only with high-level information such as knowing when a pattern occurred. While the full correspondence can be used to extract the high-level information, in some cases obtaining the high-level information directly could be done more efficiently.

To this end, we introduce the semantic network model (SNM), which allows us to mark certain states of an underlying probabilistic model. The marked states are termed *semantic states*, with the assumption that visiting such a state carries some meaning important to the application. With an SNM, we can formulate and solve general problems related to semantic states. For example, given a sequence of observations, we can find what semantic states were likely visited, and when they were visited. If the semantic states were used to mark the beginning and end of pattern submodels, that corresponds to finding which patterns occurred in the data, and when they occurred. This gives a more direct way of obtaining the important high-level information, without the need to find the full correspondence between all states and observations.

Obtaining the high-level information more efficiently is especially important in real-time scenarios, such as those involving continuous user interaction via gesture or speech. In such scenarios, we continuously receive new data that can change our perspective on what has happened in the past. For example, what we first interpret as the word "forever" may turn out to more likely correspond to the words "for everyone" as we receive new data. The SNM allows us to pinpoint higher-level events such as recognitions of new words or undoing recognitions of previously hypothesized occurrences, by concisely describing them as changes in the hypothesized sequence of visited semantic states.

For example, the SNM framework can continuously provide the application with high-level event information concerning the semantic states:

- at frame 100: "the word *forever* has been completed at frame 100"

- at frame 101: "the completion of the word *forever* has been revised to frame 101"
- at frame 200: "undo completion of the word *forever*; the word *for* has been completed at frame 30; the word *everyone* has been completed at frame 198".

An application using speech and gesture recognition can then be implemented as an event driven system that responds appropriately to new visits of semantic states, revised times of visitation, and undoing of previously assumed visits to semantic states.

In addition to gesture segmentation and event-driven application frameworks, SNMs can also be used to facilitate on-line learning, as well as finding multiple likely explanations of the data.

This chapter is organized as follows. In Section 3.1.1, we will briefly discuss existing work related to the SNM. Section 3.2 introduces the SNM, which couples an AHMM with a set of semantic states. Algorithms related to extracting optimal sequences of semantic states are presented in Sections 3.2.1 and 3.2.2. Section 3.3 presents a few practical applications of the SNM. We end with experimental results in Section 3.4 and conclusions in Section 3.5.

3.1.1 Related Work

The semantic network model was presented in a preliminary form in [38]. In this paper, we define the SNM using the augmented hidden Markov model (AHMM) (see Chapter 2), which allows us to state the SNM concisely as a method of marking semantic states in an arbitrary AHMM.

This permits us to formulate problems specifically related to semantic states. The algorithms we present as solutions to these problems are modifications of the commonly used Viterbi algorithm [12], and in particular its adaptation to AHMMs.

For hierarchically constructed AHMMs, semantic states can be used to mark the boundaries of submodels. Combined with the algorithms we present in this paper, this yields efficient segmentation of observations across the submodels. Similar functionality is offered by traditional hierarchical hidden Markov models [11], but the extraction of the segmentation is not as efficient. This is important when we need to revise the segmentation continuously as new observations are received.

Because the SNM is presented in terms of the AHMM, it is directly applicable to a number of probabilistic models representable by AHMMs. This includes hidden Markov models (HMMs) [37], terminating HMMs [2], hierarchical HMMs (HHMMs) [11], semi-Markov models [32], reduced-parameter models [43], and Markov models of order larger than 1 [48].

3.2 Semantic Network Model

The semantic network model permits states of an AHMM to be marked as *semantic states*. An SNM is therefore defined by an AHMM and a set of semantic states. We use the name *semantic* with the assumption that passing through a semantic state has some higher-level meaning, although this does not necessarily need to apply. Marking the states allows us to formulate and solve general problems that focus specifically on detecting when semantic states are visited.

For example, in many gesture recognition applications, the main goal is to determine which gestures have been executed. Given an AHMM which models a number of gestures, we can use the Viterbi algorithm and its traceback mechanism to solve this problem - by tracing back through the path of most likely states, and knowing which gestures the states correspond to, we know what gestures were executed. But if we only care about which gestures the traceback visits, we could use semantic states to mark the beginning and ending states of each gesture, and then find only the semantic states in the most likely state sequence. Then, the most likely sequence of semantic states corresponds directly to the gestures most likely completed in the observation sequence.

In general, with the semantic states marked, we can pose questions regarding sequences of visited semantic states. In this paper, given an AHMM model with marked semantic states and a sequence of observations produced by the model, we present algorithms answering the following two questions:

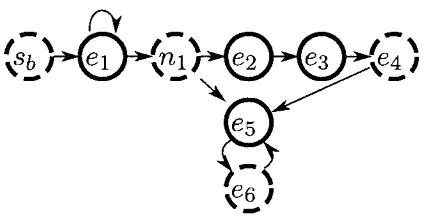

Fig. 3.1. A sample semantic network model. The semantic states are indicated by dashed circles.

1. What is the semantic state subsequence of the most likely sequence of states?

2. What is the most likely sequence of semantic states and their times of visitation?

These two problems are discussed in more detail in the next two subsections. The main difference between the two problems is that the first problem considers only the most likely non-semantic path between two visitations of semantic states. The second takes into account all non-semantic paths between two visitations of semantic states. While we present solutions to both problems, only the solution to the first problem is efficient.

3.2.1 Semantic subsequence of optimal state sequence

The Viterbi algorithm is commonly used to find the most likely state sequence to have generated a known sequence of observations. In this section, we will present a modification of the Viterbi algorithm that will allow us to efficiently retrieve the subsequence of the most likely state sequence consisting only of semantic states (without needing to extract the entire most likely state sequence first). We term the modified algorithm the *semantic Viterbi* algorithm.

The Viterbi algorithm was originally presented in Section 2.4.1.

We now present the semantic Viterbi algorithm - a modification to the Viterbi algorithm that will allow us to efficiently extract the subsequence of the state sequence $s_1, s_2, \dots, s_m$ consisting of exactly those states marked as semantic states. Intuitively, we will have semantic states serve as shortcuts for the traceback - every state keeps a pointer to the last semantic state encountered in the optimal state sequence ending at that state. See Figure 3.2 for an illustration.

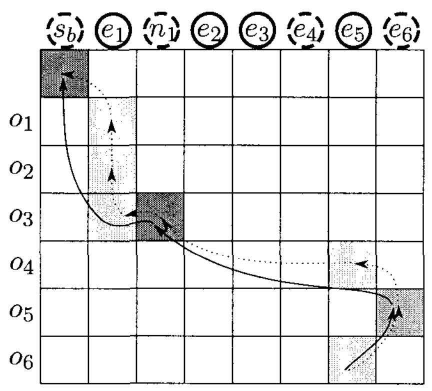

Fig. 3.2. The regular and semantic Viterbi algorithm tracebacks for the SNM shown in Figure 3.1 after 6 observations. The semantic states s_b, n_1, e_4, and e_6 are indicated by dashed circles. The matrix contains the dynamic programming variable δ, for example the cell at the intersection of o_3 and e_1 represents $\delta_3(e_i)$. The arrowed lines represent the corresponding traceback pointers captured by δ^* and δ° values. The regular traceback is shown by the dotted lines. The semantic traceback leads only to semantic states, and is shown by solid lines.

Formally, let $\overset{\circ}{S} \subseteq S$ be the set of semantic states for an AHMM $\lambda = (E, N, S_b, S_e, T, O)$. In addition to the regular traceback pointer δ^*, we also keep track to a semantic traceback pointer δ° as follows:

$$\delta_i^\circ(s) = \begin{cases} \delta_i^*(s) & \text{if } \delta_i^*(s) \in \overset{\circ}{S} \\ \delta^\circ(\delta_i^*(s)) & \text{otherwise} \end{cases} . \tag{1}$$

The semantic traceback pointer of the begin state can be initialized to a null value (denoted $\varnothing$), indicating it has no prior visited semantic state.

If we start the traceback from the final state of the most likely state sequence, we can find the most likely sequence of semantic states much quicker by using δ°. If there are n states in the state sequence, and m of them are semantic states, retrieving the semantic states will take only m traceback steps rather than the n steps that would have been required when using δ^*.

While traversing the traceback can be more efficient for the semantic Viterbi algorithm, the space and time complexity of computing the traceback is the same as that of the regular Viterbi algorithm. The semantic Viterbi needs an additional pointer for each dynamic programming variable value - $\delta_i^\circ(s)$ in addition to $\delta_i^*(s)$ for each $\delta_i(s)$. The time complexity for computing $\delta_i^\circ(s)$ is constant once $\delta_i^*(s)$ is computed. Hence, both the space and time used by the semantic Viterbi are within constant factors of space and time used by the regular Viterbi algorithm.

3.2.2 Optimal semantic state sequence

In this section we discuss the problem of finding the most likely sequence of semantic states (and the times at which they were visited), given a sequence of observations. This problem is different than the one discussed in the previous section because it accounts for different paths between the semantic states, whereas the previous problem only considered most likely paths.

Formally, given an AHMM $\lambda = (E, N, S_b, S_e, T, O)$, a set of semantic states $\mathring{S} \subseteq (E \cup N)$, and a particular sequence of n observations $o_1, o_2, \ldots, o_n$, we seek the most likely sequence $(\mathring{s}_1, \mathring{t}_1), (\mathring{s}_2, \mathring{t}_2), \ldots, (\mathring{s}_m, \mathring{t}_m)$ of semantic states $\mathring{s}_i \in \mathring{S}$ visited at times $\mathring{t}_1, \mathring{t}_2, \ldots, \mathring{t}_m$.

Unlike the previous problem, we do not have an efficient solution for the optimal semantic state sequence. For completeness, we present an inefficient optimal solution.

In expressing the solution, we will use a partial semantic state sequences terminated by a visit to a particular state (either semantic or non-semantic). We will denote this as $(\mathring{s}_1, \mathring{t}_1), (\mathring{s}_2, \mathring{t}_2), \ldots, (\mathring{s}_m, \mathring{t}_m), (s, i)$. Such a sequence denotes that the first semantic state visited was $\mathring{s}_1$, and that it was visited at time $\mathring{t}_1$, followed by $\mathring{s}_2$ at time $\mathring{t}_2$. It is possible that there is a gap between $\mathring{t}_1$ and $\mathring{t}_2$, if there were one or more emitting states visited between $\mathring{s}_1$ and $\mathring{s}_2$. It is also possible that $\mathring{t}_1 = \mathring{t}_2$, if $\mathring{s}_2$ is a non-emitting state and all states visited between $\mathring{s}_1$ and $\mathring{s}_2$ at time $\mathring{t}_1 = \mathring{t}_2$ are non-emitting states. The last semantic state visited (with the possible exception of s) is $\mathring{s}_m$, and it was visited at $\mathring{t}_m$. Following $\mathring{s}_m$, the last state to be visited is s, at time i. Any number of non-semantic states may be visited between $\mathring{s}_m$ and s.

We can now define the following dynamic programming variable (note that the $\varnothing$ symbol represents the null value, rather than the empty set $\emptyset$):

$$\gamma_i(s, \mathring{s}, t) = \max p \left(\begin{array}{l} o_{1:i}, \\ (\mathring{s}_1, \mathring{t}_1), \ldots, (\mathring{s}_m = \mathring{s}, \mathring{t}_m = t), \\ (s, i) \end{array} \middle| \lambda \right), \tag{2}$$

s.t. $\mathring{s} \in \mathring{S} \cup \{\varnothing\}$ and $\mathring{s}_j \in \mathring{S}$ for $j = 1, \ldots, m$.

The variable maximizes the joint probability of the observations $o_{1:i}$ with a partial optimal semantic state sequence, with the constraint that the complete state sequence ends with s (visited at time i), and has $\mathring{s}$ as the prior visited semantic state (visited at time t). If $s \in \mathring{S}$, this is equivalent to the probability of the optimal partial semantic state sequence constrained to have $\mathring{s}$ and s as the last two visited semantic states.

We also define:

$$\zeta_i(s) = \max_{\mathring{s}, t} \gamma_i(s, \mathring{s}, t) \tag{3}$$

$\zeta_i(s)$ represents the probability of the most likely prior visited semantic state $\mathring{s}$ and time of visitation t.

We initialize by setting $\gamma_i(s_b, \varnothing, \varnothing)$ to 1. In general, γ values for a state are calculated from γ values among all states that we can transition from. For $t = i - 1$ and $s \in E$, we have:

$$\gamma_i(s, \mathring{s}, t) = \zeta_{i-1}(\mathring{s}) + \sum_{s' \notin \mathring{S}} \gamma_{i-1}(s', \mathring{s}, t) T(s', s) O(o_i, s) \tag{4}$$

For $t = i$ and $s \in N$, we have:

$$\gamma_i(s, \mathring{s}, t) = \zeta_i(\mathring{s}) + \sum_{s' \notin \mathring{S}} \gamma_i(s', \mathring{s}, t) T(s', s) \tag{5}$$

For the remaining cases, we have:

$$\gamma_i(s, \mathring{s}, t) = \begin{cases} \sum_{s' \notin \mathring{S}} \gamma_{i-1}(s', \mathring{s}, t) T(s', s) O(o_i, s) & \text{if } s \in E \\ \sum_{s' \notin \mathring{S}} \gamma_i(s', \mathring{s}, t) T(s', s) & \text{if } s \in N \end{cases} \tag{6}$$

The traceback can now be computed using:

$$\zeta_i^*(s) = \arg\max_{(\mathring{s}, t)} \gamma_i(s, \mathring{s}, t) \tag{7}$$

As we have mentioned, this solution is very inefficient, as it requires a 4-dimensional array for the dynamic programming variables - one dimension for the time step (i), one for the final state (s), one for the prior semantic state ($\mathring{s}$), and one for the time of visitation of the prior semantic state (t).

3.3 Applications

3.3.1 A Pattern Segmentation SNM

As a concrete example of application of SNMs, we tackle the problem of pattern recognition in an unsegmented observation sequence. For example, the patterns to be recognized may be gestures or words.

Suppose we want to recognize n different patterns, numbered 1 through n. We assume we are given *examples* of each of the patterns as training data. Each example is given as a tuple (L^i, M^i), where $L^i \in \{1, 2, \ldots, n\}$ is the pattern number, and $M^i = M_1^i, M_2^i, \ldots, M_{|M^i|}^i$, with each $M_j^i \in \mathbb{O}$, is the sequence of observations for that example.

We are also given an observation sequence $o_1, o_2, \ldots$, one observation at a time, with the goal of finding continuous parts of the observation sequence that correspond to the patterns. That is, we want to separate parts of the observation sequence that are somehow similar to the example patterns from parts that are not.

The typical way of addressing this problem using probabilistic state-based models such as AHMMs is to cast the probabilistic model as a pattern producing mechanism. To accomplish pattern

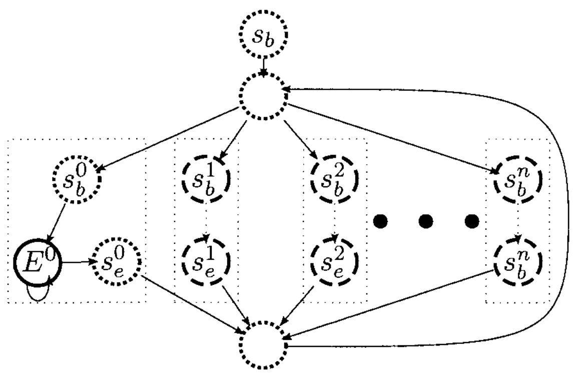

Fig. 3.3. The pattern recognition SNM. Dotted rectangles represent the submodels representing non-pattern observations (superscript ^{0}) and gesture observations. Dotted transitions indicate a path in the transition graph. Begin and end states of each gesture submodel are marked as semantic states, to capture beginnings and ends of gestures.

recognition, we then focus on two tasks. First, we train the probabilistic model so that patterns it is likely to produce correspond to patterns we are trying to recognize. Each pattern will be represented by a individual submodel, which is then trained using only examples of that pattern. The submodels will be combined in a larger model, in which we try to find the most likely sequence of states to have produced the observed observations. The sequence of states will then indicate which patterns were executed, and when.

To solve the problem, we use an SNM composed of n submodels, with each submodel modeling one of the patterns. A particular submodel k will be trained using all examples (L^i, M^i) such that $L^i = k$. The SNM will also contain a submodel which will model non-pattern behavior - observations observed when no pattern is being expressed. Finally, there are two additional states used to glue the submodels together, a begin state for the composite AHMM model, and (optionally) a terminating end state.

Figures 3.3 and 3.4 display the general structure of the two SNM variants (one without a terminating state, and one with it). For $i = 0$ (non-pattern submodel) and $i = 1, \ldots, n$ (pattern submodels), we mark the states s_b^i and s_e^i as semantic states, indicating the initiation and completion

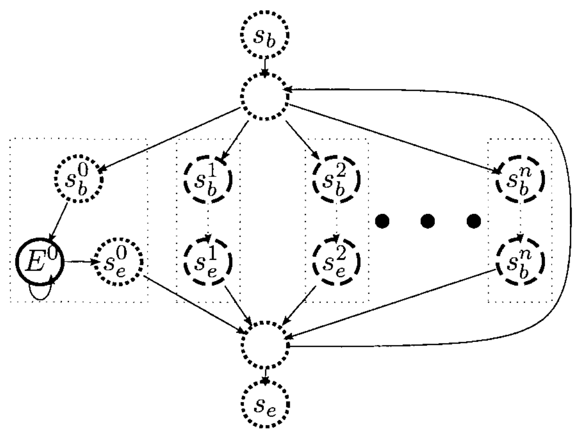

Fig. 3.4. The terminating pattern recognition SNM. Equivalent to the pattern recognition SNM presented in Figure3.3, except for the addition of the end state.

of a pattern sequence. Transitions into each of the pattern entry states s_b^i have equal values, i.e. we consider each pattern to be equally likely.

Note that this closely follows the structure of a hierarchical hidden Markov model, in which each of the gesture models and the non-pattern model would correspond to a single abstract state containing the underlying states.

Using the above SNM, at any point of the observation sequence we can use the semantic Viterbi algorithm (Section 3.2.1) and its semantic traceback to yield the desired segmentation of observations into identified patterns. In the following section, we present the use of the semantic Viterbi algorithm in more detail, and in conjunction with an event-based application framework.

3.3.2 Event-based Application Framework

We can continuously monitor the observation sequence $o_1, o_2, \ldots$ for execution of learned patterns by feeding the observations to the adapted Viterbi algorithm (Section 3.2.1). After each observation is processed and the δ values for that time step computed, we can perform a traceback to extract the semantic states from the most likely state sequence. We offer two strategies for the choice of the starting state for the traceback, corresponding to the two variants of the model presented in Section 3.3.1.

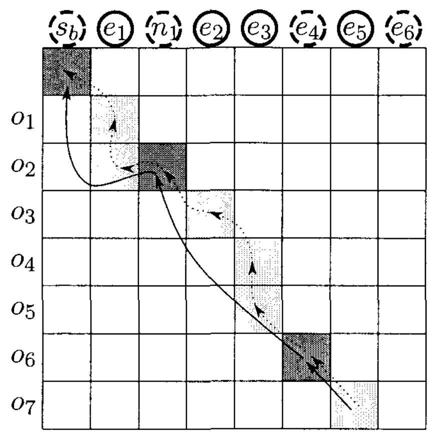

Fig. 3.5. The regular and semantic Viterbi algorithm tracebacks for a 7-state AHMM after 7 observations - one observation in addition to the scenario presented in Figure 3.2. The new observation has changed both the regular and semantic tracebacks.

For the non-terminating SNM illustrated in Figure 3.3, we know that the overall most likely state sequence ends with state s given by $\arg\max_s \delta_i(s)$. We can perform a semantic traceback from s to determine the semantic states in the most likely state sequence.

For the terminating SNM illustrated in Figure 3.4, after each observation we can make the assumption that the observation sequence terminates at that point. With that assumption, any state sequence must terminate with s_e, and therefore we perform the traceback from s_e.

The two strategies yield slightly different behaviors, as will be shown in our experimental results.

We can obtain the semantic traceback after every observation is processed. As new observations come in, it is possible that the most likely state sequence (and therefore the semantic traceback) can change significantly. Changes can occur not only in the most recent gesture, but most likely previous gestures as well. For example, Figure 3.5 shows an updated traceback obtained by processing one additional observation from Figure 3.2.

Note however that if the semantic traceback obtained after time step i and the semantic traceback obtained at time step j share a common element (e.g., the visit to semantic state s at time k), all elements of the two respective semantic tracebacks prior to the common element will also be shared. We can therefore conclude that if the final elements of two semantic tracebacks are equal, the two tracebacks are completely equal (or equivalently, if the semantic tracebacks are different, their final elements must also be different).

Because changes to the traceback from one frame to another occur from the end of the traceback, they can be expressed in terms of pushes (new visits to semantic states appearing in the traceback) and pops (disappearances of visits to semantic states previously present in the traceback). For example, the semantic state sequence with associated visitation times from Figure 3.2 is $(s_b, 0)$, $(n_1, 3)$, $(e_6, 5)$. The updated semantic sequence from Figure 3.5 is $(s_b, 0)$, $(n_1, 2)$, $(e_4, 6)$. The transformation between the former sequence and the latter can be described as a pop of $(e_6, 5)$ and $(n_1, 3)$ from the end of the sequence followed by a push of $(n_1, 2), (e_4, 6)$.

The same sequence of changes could be given to a client application in the form of events:

- undo visit to e_6 at time step 5 (pop $(e_6, 5)$)
- revise time of visit to n_1 from 3 to 2 (pop $(n_1, 3)$, push $(n_1, 2)$)
- report visit to e_4 at time step 6 (push $(e_4, 6)$).

Returning to our pattern segmentation SNM, we can issue similar higher-level events to a client application:

- report beginning of pattern k at time step t
- report end of pattern k at time step t
- revise beginning of pattern k from time step t to t'
- revise end of pattern k from time step t to t'

- undo beginning of pattern k at time step t
- undo end of pattern k at time step t

Note that the revisions in time of visitation are simply consecutive pops and pushes of the same semantic state, but with different times. Because we find it to be a common occurrence in running the algorithms, we treat it specially. Client applications are also likely to treat a revision of time of visitation differently than the combination of undoing and re-reporting.

3.3.3 ON-LINE LEARNING

When we detect an instance of a pattern in the observation sequence (which, if detected correctly, is an example of the pattern much like the training data), it can be used to improve the pattern model.

In the pattern segmentation SNM, we can use the semantic traceback to segment the execution of a single pattern. Within the segment, we can then use the regular traceback to retrieve the most likely complete state sequence that has produced the recently recognized gesture, which we can use to update the submodel for that particular pattern (this is similar to the approach taken by Lee and Xu[22]). For example, for each state in the state sequence that was assumed to produce a particular observation in the observations sequence, we can modify the observation probability function related to the state to incorporate information given by the new observation. This can be done in many ways, for example by re-running whatever training algorithm is used with the new example included, or by using other methods which integrate new data into an existing observation probability model. Transition probabilities can be updated similarly, using the transitions from the most likely state sequence.

3.3.4 Considering multiple likely last states

The concept of a semantic traceback allows us to consider multiple likely high-level explanations of the observation sequence. So far, we have only considered the semantic subsequence of the most likely state sequence, but we can similarly consider semantic subsequences of other state sequences.

By starting the semantic traceback from an arbitrary state s, we obtain the semantic subsequence of the most likely state sequence ending with s. If we compute the traceback from a number of states (not just the most likely ending state), we get a number of semantic subsequences. To find the k most likely semantic subsequences of such state sequences (note that this is not necessarily the same as the k most likely semantic subsequences of all possible state sequences), we can follow the following algorithm:

1. find the most likely ending state s
2. compute the semantic traceback from state s and record it
3. find the next most likely ending state s'
4. if the last visited state of this semantic traceback from state s' is different than the last visited state of any of the recorded tracebacks, record this semantic traceback
5. if the number of recorded semantic tracebacks is less than k, return to step 3.

This algorithm relies on the property presented in Section 3.3.2, stating that if two semantic tracebacks are different, their final elements must also be different. The experimental results section offers some runtime measurements related to examining the final elements of all semantic tracebacks.

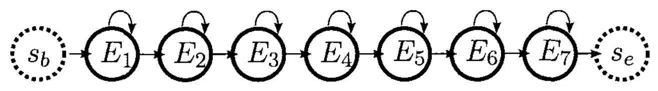

Fig. 3.6. A chain AHMM with 7 emitting states. Each state in the chain can transition to itself or the next state. A chain AHMM with 12 emitting states was used as model for each trained gesture.

3.4 Experimental Results

We have implemented the modified Viterbi algorithm used to find the semantic subsequence of the most likely sequence of states (Section 3.2.1) in the AME Patterns open source library (see Chapter 4), distributed as part of AMELiA [40]. In this section, we present some experimental results obtained from the implementation. The results have been obtained using two datasets: one with gesture data and one with speech data.

The first dataset used is obtained from stereo video data, with feature vectors extracted using the method described in [36]. It contains training data for 14 gestures (8 training samples per gesture), and 7 unsegmented test sequences containing executions of the trained gestures as well as non-gesture data. The lengths of the test sequences vary between 1235 and 1865 frames, with an average length of 1471. Typical executions of gestures vary between roughly 10 and 30 frames in length.

Each gesture was modeled by a chain AHMM with 12 emitting states (see Figure 3.6 for an example), with the submodels combined into an SNM as described in Section 3.3.1, with beginning and ending states of each submodel marked as semantic states.

The second dataset used is a subset of the TIMIT corpus [23]. The dataset is processed identically to what is described in [6]. It contains 200 training sequences and 192 test sequences, each a manually segmented spoken sentence annotated with ground truth data. The data is modeled by 48 phonetic classes, however the 48 classes are mapped down to 39 classes for the purpose of evaluating the correctness of a recognition (that is, some of the 48 classes are considered equivalent for this purpose).

TABLE 3.1

Run times of inference and traceback when running the algorithms on a concatenation of the 7 test sequences, and traversing a complete traceback after each observation. The semantic Viterbi algorithm is overall significantly more efficient.

	Viterbi	semantic Viterbi
inference	4.77	5.50
traceback	4.47	0.10
total	9.24	5.60

Each phoneme class is modeled by a single state (which can transition to itself or terminate the submodel), with the submodels combined and semantically marked similarly to what was done for the gesture dataset.

3.4.1 Runtime

We first compare the runtime of the general AHMM Viterbi algorithm with the modified version introduced in this paper. As the modified version requires extra data for each state for each observation, to hold the semantic traceback pointer, as well as extra computation to set the traceback pointer, it will take more space and more time. For the runtime tests, we only utilize the gesture dataset.

For each test case below, the runtime was measured for two separate phases. The inference phase of the algorithm involves calculation of δ, δ^*, and in the case of the semantic Viterbi, δ° values. In the traceback phase, for the regular Viterbi algorithm we traverse the regular traceback, and for the semantic Viterbi, we traverse the semantic traceback. All reported times are means of 5 separate measurements.

Table 3.1 shows the results of running the regular and semantic Viterbi algorithms on the 7 test sequences, with the sequences given to the algorithms consecutively one after another (as if we were processing a single test sequence that was a concatenation of the 7). A full traceback

TABLE 3.2

Run times of inference and traceback when running the algorithms separately on the 7 test sequences, and traversing a complete traceback after each observation. Because of the shorter average traceback length, the advantage obtained from speeding up the traceback is smaller than the cost of paying for the more expensive inference of the semantic Viterbi algorithm.

	Viterbi	semantic Viterbi
inference	4.76	5.52
traceback	0.20	0.02
total	4.96	5.54

(either regular or semantic) was computed after every observation. During the regular traceback, we also tested each state to see whether it is semantic (this is what we would need to do to obtain the semantic subsequence without the semantic traceback). Because of the combined processing of test sequences, the average length of the full traceback was roughly 5000 elements long. This compares to an average of roughly 140 elements for the semantic traceback.

As the results indicate, the semantic Viterbi algorithm takes approximately 15% more time to compute the inference portion of the algorithm because of the additional δ° computation. However, traversing the tracebacks took approximately 45 times less with the semantic Viterbi.

For this experiment, the total runtime cost of inference + traceback was roughly 65% more costly for the regular Viterbi algorithm.

In the next experiment, we repeat the previous setup, except we process each test sequence individually. This yields an average length of the regular traceback of roughly 750, and an average length of the semantic traceback to roughly 20.

The results are presented in Table 3.2. The inference portion of the algorithm performs almost identically as the previous setup, as would be expected. However, the difference in the traceback is now less significant because the average lengths of tracebacks are much smaller. Consequently, the

TABLE 3.3

Run times of inference and traceback when running the algorithms separately on the 7 test sequences, and traversing a partial traceback from each state after each observation. The semantic Viterbi algorithm is more efficient overall.

	Viterbi	semantic Viterbi
inference	4.76	5.52
traceback	2.15	0.47
total	6.91	5.89

advantage obtained from speeding up the semantic traceback is smaller than the cost of paying for the more expensive inference of the semantic Viterbi algorithm.

For this experiment, the total runtime cost of inference + traceback was roughly 12% more costly for the semantic Viterbi algorithm.

For the final runtime experiment, we now at each time step compute a partial traceback from *each* state. We traverse the traceback only to the last visited semantic state. This kind of computation would be useful, for example, if we wanted to examine other semantic subsequences in addition to the most likely one (for example, we wanted to consider the k most likely distinct semantic subsequences). As presented in Section 3.2, it is sufficient to compare the last visited semantic states of two state sequences to determine whether their semantic subsequences are distinct.

In this case, the semantic Viterbi traceback takes roughly 4.5 times less time than the regular traceback (see Table 3.3). This is sufficient to offset the cost of the more expensive semantic Viterbi inference.

For this experiment, the total runtime cost of inference + traceback was roughly 17% more costly for the regular Viterbi algorithm.

TABLE 3.4

Semantic tracebacks obtained after processing all observations of test sequence 1, using the two different strategies. The tracebacks are identical except for the most likely state strategy reporting an additional gesture beginning as the last semantic state visited.

most likely state strategy		end state strategy	
semantic state	time step	semantic state	time step
g. 1 beginning	1	g. 1 beginning	1
g. 1 end	137	g. 1 end	137
g. 1 beginning	150	g. 1 beginning	150
g. 1 end	191	g. 1 end	191
g. 2 beginning	205	g. 2 beginning	205
g. 2 end	333	g. 2 end	333
g. 2 beginning	444	g. 2 beginning	444
g. 2 end	505	g. 2 end	505
g. 8 beginning	514	g. 8 beginning	514
g. 8 end	934	g. 8 end	934
g. 1 beginning	934	g. 1 beginning	934
g. 1 end	1061	g. 1 end	1061
g. 2 beginning	1136	g. 2 beginning	1136
g. 2 end	1219	g. 2 end	1219
g. 1 beginning	1297	g. 1 beginning	1297
g. 1 end	1365	g. 1 end	1365
g. 1 beginning	1441	g. 1 beginning	1441
g. 1 end	1522	g. 1 end	1522
g. 1 beginning	1535	g. 1 beginning	1535
g. 1 end	1579	g. 1 end	1579
g. 3 beginning	1602		

3.4.2 Segmentation of a Complete Sequence

We next turn our attention to segmentation of a complete test sequence, performed by processing the entire sequence using the semantic Viterbi algorithm, and then computing a single semantic traceback.

We compare the two strategies presented in Section 3.3.2 - using the non-terminating SNM and performing the traceback from the most likely state (the *most likely state* strategy), or using the terminating SNM and performing the traceback from the end state (the *end state* strategy).

We will first present some qualitative results for the gesture dataset. We found that for this dataset, after an entire test sequence was processed the most likely state strategy always resulted in a traceback that ended with a visit to a semantic state marking the beginning of a gesture. This means that the model predicted that the test sequence was inside a gesture after the last observation. However, each test sequence was segmented such that it both began and ended with non-gesture movement. Table 3.4 shows the complete traceback for the first test observation sequence, where the tracebacks are identical except for the most likely state strategy reporting an additional gesture beginning as the last semantic state visited.

On the other hand, the end state strategy always resulted in a traceback that ended with a visit marking the end of a gesture (since the end state cannot be reached from a beginning state of a gesture without going through the end state of the gesture, this must be the case). Therefore, if the scenario is such that the test observation sequence has a known point of termination, we recommend using the terminating SNM and the end state strategy for the final traceback.

We now turn to the TIMIT speech recognition dataset. For this dataset, we present some quantitative performance metrics evaluating the segmentation.

The metrics were obtained by computing the optimal edit distance alignment between the inferred segmentation of a test sequence and the ground truth. An inferred segmentation segment is allowed to be correctly matched to a ground truth segment if they overlap and belong to the same phoneme class. If they overlap but do not belong to the same class, they can be considered a mismatch. False alarms (inferred segments that do not correspond to a ground truth segment) and missed detections (ground truth segments that do not correspond to an inferred segment) correspond to edit-distance insertions and deletions. Edit distance penalties for errors (mismatches, false alarms, and missed detections) are all set to 1 (hence, the optimal alignment minimizes the total number of errors).

TABLE 3.5

Metrics provided by edit-distance alignment of ground truth and the inferred segmentation for the TIMIT dataset. The results were identical for both the end state strategy and the most likely state strategy.

Ground Truth Segments	Inferred Segments	Matches	Mismatches	Misses	False Alarms	Error rate
7198	6370	4240	1752	1206	378	46.3%

Table 3.5 shows the number of matches, mismatches, false alarms and misses in the segmentation of complete sequences of the TIMIT dataset, as well as the error rate (computed as the number of errors divided by the ground truth size). Note that the error rate is comparable to that of the method presented in [6].

3.4.3 Latency and Stability

Since new observations can significantly change the semantic traceback, we also compare the two strategies on the behavior of the semantic traceback throughout the entire observation sequence.

We will first present some qualitative results for the gesture dataset. For clarity, we focus on detection of gesture completions, as indicated by visits of semantic states marking the end of a gesture.

While processing the first test sequence one observation at a time, the most likely state strategy yields the following sequence of events related to visits to end states of gestures:

- at frame 139: report end of gesture 1 at frame 137
- at frame 193: report end of gesture 1 at frame 191
- at frame 335: report end of gesture 2 at frame 333
- at frame 508: report end of gesture 2 at frame 505

- at frame 668: report end of gesture 3 at frame 661
- at frame 857: undo end of gesture 3 at frame 661
- at frame 872: report end of gesture 8 at frame 869
- at frame 917: undo end of gesture 8 at frame 869
- at frame 937: report end of gesture 8 at frame 934
- at frame 1064: report end of gesture 1 at frame 1061
- at frame 1226: report end of gesture 2 at frame 1219
- at frame 1366: report end of gesture 1 at frame 1365
- at frame 1524: report end of gesture 1 at frame 1522
- at frame 1583: report end of gesture 1 at frame 1579

Most recognitions found in the final traceback (see Table 3.4) are reported exactly once. The exception is the end of gesture 8, eventually detected at time step 934. Prior to that detection, there is a detection of end of gesture 3, which is later undone, and an earlier detection of the end of gesture 8, which is also undone.

For recognitions that match the final traceback, the time of detection slightly lags behind the time the end of the gesture is visited. For example, the fact that the first occurrence of gesture 1 ends at frame 137 is detected two frames later, at frame 139. The average latency of detection of gestures 1 and 2 for this test case is 2.9 frames, ranging from 1 frame of latency to 7 frames.

For the end state strategy, the list of events is much longer. For example, this is the list of events related to the second gesture recognition (eventually settling on gesture 1 ending at frame 191):

- at frame 185: report end of gesture 2 at frame 185
- at frame 186: undo end of gesture 2 at frame 185
- at frame 189: report end of gesture 1 at frame 189
- at frame 190: revise end of gesture 1 to frame 190
- at frame 191: revise end of gesture 1 to frame 191
- at frame 192: revise end of gesture 1 to frame 192
- at frame 193: revise end of gesture 1 to frame 193
- at frame 194: revise end of gesture 1 to frame 191

While the final event related to this recognition matches that found by the most likely state strategy, it is preceded by a number of other events. First, the model reports a recognition of gesture 2, only to undo the recognition a frame later, and finally replace it with a recognition of gesture 1. Gesture 1 completion is first detected at frame 189, and then revised five times before it settles on frame 191. Hence, the gesture was first detected 2 frames earlier than the most likely point of completion of the gesture as given after all observations are processed. On average, the end state strategy reported recognitions of gestures 1 and 2 ahead of the most likely point of completion by 2.7 frames. However, all of the recognitions were preceded by recognitions of other gestures which were eventually undone.

For the TIMIT dataset, we again present quantitative performance metrics related to latency and stability. Latency is computed by measuring the lag between the inferred beginning (or end, respectively) of a pattern, and the time that the beginning (or end, respectively) is detected by each of the strategies. Furthermore, it is measured both for the first detection of a beginning or end, and

TABLE 3.6

Latency (in frames) of first and last detections of beginnings and ends of patterns.

Strategy	first beginning detection	last beginning detection	first end detection	last end detection
most likely state	5.97	5.97	3.64	4.71
end state	5.96	5.96	-2.95	4.51

TABLE 3.7

Number of cases when a strategy undid or revised a previously reported beginning or end of a pattern. Percentages are given relative to the total number of reported segments at the end of the sequence.

Strategy	undos of beginnings	revisions of beginnings	undos of ends	revisions of ends
most likely state	5570 (88.8%)	13 (0.2%)	1086 (17.3%)	1949 (31.1%)
end state	5777 (90.7%)	10 (0.2%)	5777 (90.7%)	53598 (841.4%)

the last detection (recall that the system can occasionally revise an estimate of a beginning or end of a pattern).

Table 3.6 presents the latency metrics for both of the recognition strategies. Both strategies yielded a lag of roughly 6 frames before the beginning of a pattern was detected. Since the average length of an inferred pattern was roughly 9 frames, it means beginnings of patterns were typically detected two-thirds of the way into the pattern.

With respect to latency in end detection, the two strategies differ in their behavior. The most likely state strategy shows an average lag of 3.64 frames for the first detection of a pattern end. The position of end of the pattern is then further re-estimated for an average of one additional frame. The end state strategy, on the other hand, detects a pattern end nearly 3 frames ahead of the point of completion.

Stability is measured by looking at cases where a strategy revises a previously detected beginning or end of a pattern, as well as the number of cases when the detection is completely undone. Note that undos can occur either when it stops being likely that a pattern was being executed, or when a different pattern becomes a more likely explanation for the data.

Table 3.7 shows the number of occurrences of each of these cases, as well as a percentage relative to the total number of reported segments at the end of the sequence. Both strategies show a relatively high number of undos of beginnings of patterns. The number of such undos is at roughly 90% of the total number of segments reported, meaning that for almost every reported segment there is a segment whose beginning was reported and later undone. Both strategies also show a very low percentage (0.2%) of revisions of beginnings of patterns, meaning that once the beginning of a pattern is detected, its position is rarely revised.

With ends of patterns, the two strategies exhibit drastically different behavior. The most likely state strategy shows a relatively low number of undos of pattern end detections (17.3%). This means that once a pattern end is detected, the pattern detection is relatively rarely undone. The number of revisions is also comparatively low (31.1%), meaning that the estimate of the position of the end of the pattern doesn't change often. On the other hand, the end state strategy shows a much higher number of undos of pattern end detections (90.7% of the total number of reported segments at the end of the sequence - note that this is the same as the percentage of undos of pattern beginning detections, which is always the case with the end state strategy). This means that for almost every reported segment there is a segment whose end was reported and later undone. The number of revisions of pattern ends is staggeringly high for the end state strategy (over 8 times the number of reported segments at the end of the sequence). This means that once a pattern end is detected, the position of its end tends to be revised many times.

Based on the latency and stability performance of both of the datasets, it is clear that the two strategies offer different tradeoffs between stability and latency. The most likely state strategy offers higher stability at the cost of higher latency. The end state strategy offers lower latency (often even reporting the end of a pattern before it happens), but at the expense of stability.

3.5 Conclusions

We have presented the semantic network model as a method of marking augmented hidden Markov models with semantic states, as well as a modified Viterbi algorithm which allows us to efficiently extract the semantic subsequence of a state traceback. The semantic traceback is useful in problems of segmentation of unsegmented observation sequences, event-based application frameworks, on-line training, and finding a number of distinct likely explanations of an observation sequence.

In our experimental results, we show that the semantic Viterbi algorithm shows reasonable speedup over the regular Viterbi algorithm when dealing with long or multiple semantic tracebacks. In some cases, however, the higher cost of the semantic Viterbi algorithm does not match the benefit of the faster semantic traceback - in such cases, it may be better to extract the semantic traceback by traversing the full traceback provided by the regular Viterbi algorithm.

We also compared two strategies for event-based application frameworks, which differ in their model and choice of state from which to perform the traceback. The most likely state strategy was shown to yield stable results on our test data (low number of undos / revisions of the semantic sequence), but suffered from some latency. On the other hand, the end state strategy was much more unstable (high number of undos / revisions), but tended to detect gesture completions much earlier.

4. AME PATTERNS LIBRARY

4.1 INTRODUCTION

The AME Patterns library [39] is a C++ library for modeling, recognition, and synthesis of sequential patterns. It can be used for applications such as gesture recognition from video or motion capture data, speech recognition, as well as synthesis of gesture and speech patterns. The library is released under the GNU General Public License as a part of AMELiA (the Arts, Media and Engineering Library Assortment), an open source library collection.

The core of the library deals with *pattern models* - probabilistic graph-based models such as the hidden Markov model. The library allows the pattern models to be created either manually or from training data. Once built, pattern models can be used with *inference* algorithms to achieve pattern classification, segmentation, or on-line recognition, or they can be used with *generation* algorithms to achieve pattern synthesis.

The library uses generic programming [10] to express the algorithms in a generic fashion. The algorithms can then easily be applied to any modality (e.g., speech, gesture, or purely numerical data) by providing models of some basic concepts used within the library - such as the type of *observations* that make up a gesture (e.g., a body gesture can be made up of individual observations of body poses, and a mouse gesture can be made up of individual observations of mouse movement directions). The use of generic programming also allows the library to be used with data types that provide different tradeoffs, e.g. between space and time complexity.

As pattern analysis is used in a wide variety of scenarios, such as speech, gesture, text, numerical data, etc., as a research discipline it can greatly benefit from a library that uses generic programming principles. Such principles allow a single library to be effectively and efficiently used in all of these scenarios, and allows ideas previously developed in individual contexts to be applied to other contexts, while still allowing truly domain-specific optimizations to be used when appropriate. Being that much pattern analysis research happens in individual disciplines (e.g., speech and gesture recognition, or speech synthesis), a unifying library facilitates cross-usage of ideas and implementation. The AME Patterns library has been successfully used in settings as varied as speech recognition, full-body gesture recognition using optical motion capture, full-body gesture recognition using a pair of video cameras, gesture recognition using a single camera, and tangible object gesture recognition using a mouse, stylus, or a tracked ball. It has also been used with other related problems such as part-of-speech tagging.

To accommodate different programming styles and constraints, the library also provides multiple application programming interface (API) levels with different tradeoffs between programming complexity, flexibility, portability, and compile times. To achieve the highest level of flexibility, the user can use the pattern models and inference or synthesis algorithms directly, but that requires a level of comfort with generic programming techniques, and increased compile times. For common uses such as classification, on-line recognition, synthesis, or training, the library also provides various *task* class templates which are easier to use and can be somewhat customized via template arguments. We also provide a collection of purely object-oriented C++ classes for a few of the most common usage scenarios and modalities, provided in a pre-compiled library: ofxPatterns. The advantage of ofxPatterns are reduced compile times, and a simpler object-oriented C++ interface. Finally, we have recently started developing an additional library offering a C interface, with the goal of the same functionality as ofxPatterns. The C interface allows the library to be used in environments where C++ is not an option. For example, it can be loaded as a shared library into C#.

In addition to the core functionality related to probabilistic graph-based models, we are also adding support for other related pattern analysis algorithms, such as dynamic programming alignment, edit distance, and detection of approximate repetition real-time detection of approximate repetition of sequences of observations (in any modality).

This chapter presents an overview of the AME Patterns library, with a focus on the functionality related to probabilistic graph-based models. At the same time, we present the advantages and drawbacks of various design decisions afforded by the use of generic C++ programming, as well as the tradeoffs introduced by different API levels.

Although chapter presents some code examples for the purpose of illustrating the tradeoffs, it is not meant to be an in-depth tutorial on how to use the library. The online documentation [39] shows more information on how to use the library.

4.2 Preliminaries

To get the most out of this chapter, the reader should be familiar with the basics of concept-based generic programming. E.g., know that a *concept* is a specification of requirements for types, and that types that satisfy those requirements are said to *model* that concept. We will always denote concepts using *CamelCaseItalics*. Familiarity with C++ is also beneficial, as all code samples presented are in C++.

There is an inconvenient clash of nomenclature between the use of the word *model* in the concept-based programming context and in the pattern analysis context. E.g., "the type `std::vector<int>` is a model of the *Sequence* concept" vs. "this HMM is a model of a the letter *A*". To help resolve the ambiguity, we will always use "pattern model" for the latter type of usage.

When discussing specific patterns (e.g., the letter *A*), we will distinguish between the *pattern* itself and examples of that pattern. A particular instance of the letter *A* (e.g., this one right here) would be referred to as an *example* of the *pattern A*. Depending on the context, we may also use the word *instance* instead of *example*. In other literature, pattern examples are also sometimes referred to as pattern samples, or pattern exemplars.

Fig. 4.1. Namespace aliases used by code samples

```
namespace patterns = ame::patterns;
namespace observations = ame::observations;
```

The code samples presented in the chapter will assume namespace aliases as given in Figure 4.1.

4.3 Related Work

We will first present a few libraries conceptually related to AME Patterns, and then provide an overview of libraries that AME Patterns uses directly. At the end of the section, we will provide a few final remarks on related work.

4.3.1 Related Libraries

There are several existing libraries with support for HMMs and related models and algorithms. To the best of our knowledge, the AME Patterns library is the first library of this kind to be implemented in C++ using generic programming principles. Other existing libraries are implemented in C (e.g., the Hidden Markov Model Toolkit [5], and the General Hidden Markov Model library [26]), Java (e.g., Jahmm [13]), and Matlab (e.g., the HMM Toolbox [29] and the Bayes Net Toolbox [30]). The closest general-purpose library we found implemented in C++ is the Structural Modeling, Inference, and Learning Engine [9]. However, it is implemented using with an object-oriented design, rather than using generic programming principles. It is also focused on inference in Bayesian networks, and has little documented support for models such as the HMM.

We will begin by discussing two libraries related to the AME Patterns library, both implemented in C: The Hidden Markov Model Toolkit (HTK) [5], and the General Hidden Markov Model library (GHMM) [26]. Being implemented in C, neither of the libraries provide the high level of flexibility in the choice of observation types and probability distributions provided by AME

Patterns, and particularly not the facilities to combine different probability distributions to create compound probability distributions. This is due to the polymorphism in C is limited to runtime polymorphism using function pointers, which makes the implementation of runtime polymorphism cumbersome (compared to C++), and the implementation of compile-time polymorphism nearly impossible (one can rely on preprocessor tricks to achieve a form of compile-time polymorphism, but that gets rather difficult to program and maintain). These C libraries do, however, provide a high level of portability and significantly lower compilation times.

The Hidden Markov Model Toolkit is a portable toolkit for building and manipulating hidden Markov models. It's primary use scenario has been speech recognition, however it has also been applied to speech synthesis, character recognition and DNA sequencing. The library supports both discrete observation probability distributions and continuous density mixture Gaussians, and supports both regular and hierarchical HMMs.

In comparison to the AME Patterns library, advantages of HTK are its extensive documentation and examples, as well as sophisticated facilities for speech analysis, HMM training, testing and results analysis. However, while the source code of HTK is available to licensees, the license is not an OSI-approved open source license. In particular, use of the library and any derivative works is limited to the licensee's organization.

The General Hidden Markov Model library is also a C library (with an unsupported C++ API), and offers support for both discrete and continuous observations, mixtures of PDFs, non-homogeneous Markov chains, pair HMMs, and clustering and mixture modelling for HMMs. It also features a graphical editor, Python bindings, and an XML-based file format - none of which are currently offered by the AME Pattern library. As well, the GHMM is released under the LGPL, which is more permissive than the GPL used by the AME Patterns library.

We will next present a brief comparison with the Jahmm library. Through use of Java interfaces, the Jahmm library does in fact provide flexibility in the use of different observation types and distributions. The library itself provides support for normally distributed real observations, as well as higher-dimensional observations governed by a multivariate normal distribution. The library also supports both inference and synthesis based on the models, much like the AME Patterns library.

However, because of the use of the Java language, the genericity is limited to run-time polymorphism, rather than the compile-time polymorphism afforded by generic programming in C++. The benefit of the latter is that the compiler can make compile-time optimizations when possible. As well, running optimized native code is typically faster than running Java code.

4.3.2 Libraries Used

The AME Patterns library makes use of several other libraries to leverage their functionality and make the implementation of AME Patterns more focused.

Most of the libraries used are parts of Boost [54], with the most important being the Boost Graph Library (BGL). The *PatternModel* concept (presented in Section 4.4.1) is a refinement of graph concepts from that library. The AME Patterns library also makes direct use of many BGL data types and algorithms.

Some additional libraries used are:

- Boost.Range (including the RangeEx extension), used to manipulate sequences (e.g., of observations)
- Boost.Serialization, used to serialize pattern models
- Boost.Fusion, used for compound observations and observation distributions
- Boost.Math, used for statistical distributions, constants, and other mathematical functionality
- Boost.Random, used for generation algorithms

- Boost.Smart Pointers, used for implementation of equivalence classes
- Boost.Assignment, used to initialize training examples in sample code
- Boost.Spirit, used to parse datasets
- Eigen [4], used for linear algebra tasks

4.3.3 Other Related Work

There is an interesting architectural similarity between the AME Patterns library and Boost.Spirit. Boost.Spirit offers both a parser library (Spirit.Qi) and a generator library (Spirit.Karma). The *grammars* used by Spirit.Qi and the *formats* used by Spirit.Karma are highly related. This is similar to how the AME Patterns library offers both inference algorithms and generation/synthesis algorithms, with both types of algorithms using the same pattern models (i.e., the same pattern model could be used for both inference and synthesis).

We should also mention that the variety of API levels provided by the library, with different tradeoffs between ease of use and flexibility, has been partly inspired by the keynote speech given by Bjarne Stroustrup at BoostCon 2008 [51].

4.4 Library Overview

The design of the AME Patterns library makes it highly modular and versatile. At its core, the library employs a clear separation between data and algorithms, and uses concepts to allow data types and algorithms to be adapted to many usage scenarios.

4.4.1 Pattern Models

The core of the library deals with *pattern models*. From a theoretical perspective, a pattern model is an augmented hidden Markov model with equivalence classes (AHMM+EC), which was presented in Chapter 2. Therefore, the library can be used with all types of pattern models covered by the AHMM+EC, such as hidden Markov models (HMMs) [37], terminating HMMs [2], hierarchical

HMMs (HHMMs) [11], semi-Markov models [32], reduced-parameter models [43], and Markov models of order larger than 1 [48].

From an implementation perspective, the pattern model is captured by the *PatternModel* concept. This concept is a refinement of a number of concepts offered by the boost graph library. Hence, the pattern model itself is a graph in which vertices correspond to states of the pattern model, and edges to transitions between states. The concept refinements add the following functionality and constraints:

- they allow vertices to be marked as beginning or end states
- they require that emitting states must have observation probability distributions as bundled properties
- they require that edges must have transition probabilities as bundled properties

The benefit of using the *PatternModel* concept is that in addition to the general AHMM *PatternModel* provided by the library, additional *PatternModels* can be implemented offering space or time complexity optimizations for certain types of pattern models or pattern model topologies. In fact, the AHMM *PatternModel* provided in the library is itself a template that offers different specializations depending on whether the pattern model will make use of transition equivalence classes and/or observation distribution equivalence classes. When equivalence classes are used, observation distributions and transition probabilities are stored using `boost::shared_ptr`, with elements of the same equivalence class pointing to the same observation distribution / probability. Thus, the specializations offer different tradeoffs in space and time usage. An example of this is shown in Table 4.1.

TABLE 4.1

Space and time usage related to a pattern model using 10 emitting states, all of which use the same observation distribution. When `sizeof(ObservationDistribution)` is large compared to `sizeof(boost::shared_ptr)`, using equivalence classes can provide savings in memory use. However, time use may suffer due to the indirection penalty caused by use of pointers.

equivalence classes	space usage for observation distributions
yes	`sizeof(ObservationDistribution)` `+ 10sizeof(boost::shared_ptr)`
no	`10sizeof(ObservationDistribution)`

4.4.2 Observations

The *PatternModel* concept requires that emitting states have observation probability distributions as bundled properties. The library provides a number of useful observation distribution models (such as univariate and multivariate normal distributions, and discrete distributions), as well as adapters which can combine a number of component distributions into a more complex distribution (such as combining a number of univariate normal distributions into a multivariate distribution of normally distributed, independent components; or combining a discrete distribution with a normal distribution into a compound observation which has one discretely distributed part and one normally distributed part).

The observation probability distributions must model the *ObservationDistribution* concept. Models of such a concept specify a probability distribution of an underlying *Observation* type. For example, the library provides a `normal` *ObservationDistribution* which represents the normal (Gaussian) distribution over *Observations* of type `double`.

When used with inference or generation scenarios, observations will also need a *ObservationDistributionTrainer* or *ObservationDistributionGenerator*, respectively.

The purpose of an *ObservationDistributionTrainer* is to train an *ObservationDistribution* from either weighted or unweighted samples of the distribution. The purpose of an *ObservationDistributionGenerator* is to generate samples from a distribution.

For example, the AME Patterns library provides an *ObservationDistributionTrainer* for the `normal` *ObservationDistribution* which trains the distribution by computing the weighted or unweighted mean and sample variance of the provided samples. It also provides *ObservationDistributionGenerator* which generates random observations from the distribution.

The *ObservationDistribution*, *ObservationDistributionTrainer* and *ObservationDistributionGenerator* concepts, and the provided models could be considered as a sub-library of AME Patterns, as they don't rely on any other part of the library. They are in itself useful for implementation of e.g., Gaussian classifiers, and other probabilistic tasks that are concerned with single observations (rather than sequences of observations).

4.4.3 Pattern Model Construction

Pattern models in the AME Patterns library can be constructed in two ways - manually or through training. When constructing models manually, one has complete freedom over the topology, transitions probabilities, and observation probability distributions of the pattern model.

The code sample in Figure 4.2 shows how to manually construct a pattern model consisting of two emitting states, and two non-emitting states (a start state and an end state). The emitting states have normal observation probability distributions with means 0.0 and 1.0, respectively, and a standard deviation of 0.1. The transition probabilities are as shown in Figure 4.3.

Training can also be used to construct models. The library provides two training algorithms: expectation-maximization training and Viterbi training. They take an existing model and update transition and observation distribution probabilities according to provided training data. Typically, the model is first initialized manually or using some initialization algorithm that uses the training

Fig. 4.2. Manual pattern model construction

```
typedef patterns::model::ahmm<observations::normal>
    ahmm_type;

ahmm_type ahmm;
ahmm_type::vertex_descriptor a, b, c, d;

a = add_vertex(ahmm);
b = add_vertex(observations::normal(0.0, 0.1), ahmm);
c = add_vertex(observations::normal(1.0, 0.1), ahmm);
d = add_vertex(ahmm);

add_edge(a, b, ahmm);
add_edge(b, b, 0.75, ahmm);
add_edge(b, c, 0.25, ahmm);
add_edge(c, c, 0.5, ahmm);
add_edge(c, d, 0.5, ahmm);

ahmm.make_begin_vertex(a);
ahmm.make_end_vertex(d);
```

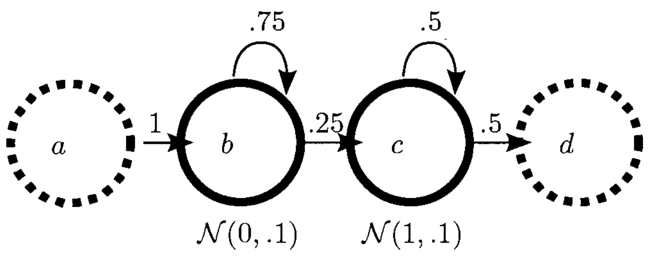

Fig. 4.3. AHMM constructed by the code sample in Figure 4.2.

Fig. 4.4. Pattern model training

```
typedef patterns::model::chain_hmm<
        observations::normal>
    ahmm_type;

ahmm_type ahmm(2);

std::vector<std::vector<double> > examples(2);

using namespace boost::assign;

examples[0] += 0, 0.1, -0.1, 1.1, 0.9, 1.0;
examples[1] += 0.1, -0.1, 0.1, -0.1, 0, 1.0;

typedef patterns::inference::best_match
<
    ahmm_type,
    ame::selectors::vector
>
    inference_type;

patterns::training::naive_alignment(ahmm, examples);
patterns::training::best_match_maximization<
        inference_type>
    (ahmm, examples, 1, 0.0);
```

data. The library provides two initialization algorithms: one which naively aligns the training data to the states in order to initialize state observation probability distributions, and one which allows the alignment to be specified in order to initialize the model.

The code sample in Figure 4.4 shows how a particular *PatternModel* type (`chain_hmm`) is initialized with 2 emitting states. The topology of this pattern model is identical to the one shown in Figure 4.3, however the transition probabilities and observation probability distributions have different initial values than those shown in the figure. The pattern model is then initialized via the naive alignment algorithm, and further refined using the best match (Viterbi) algorithm. The resulting pattern model is identical to the one shown in Figure 4.3, with the exception of the standard deviations being slightly smaller than 0.1 based on the training data.

Note that the code is somewhat involved in its use of types and templates. This is one of the drawbacks of the flexibility provided by the core library API. We will later show how some of the programming complexity can be reduced at the expense of the flexibility in the higher-level APIs (see Section 4.5).

4.4.4 Inference

Once built, pattern models can be used with *inference* algorithms to achieve pattern classification or on-line recognition. The library provides generic implementations of the forward, backward, forward-backward, and Viterbi algorithms. At the core, each inference algorithm processes observations one by one, and provides the probability value for each state for each time step (the exact meaning of the probability value changes depending on the algorithm used).

The code sample in Figure 4.5 shows how the pattern models constructed in Figures 4.2 and 4.4 can be used with best match (Viterbi) inference to find the probability of the most likely state sequence given one of the training samples used in the code sample in Figure 4.4. In addition to this higher level information, the inference algorithms can provide lower level information that

Fig. 4.5. Inference using a pattern model

```
typedef patterns::inference::best_match
<
    ahmm_type,
    ame::selectors::vector
>
    inference_type;

inference_type inference(ahmm);

double best_probability = p_match_sequence(
    inference, examples[0]);
double partial_probability = inference[2][1];
```

is contained in the dynamic programming table used by the algorithm. The code sample shows how to obtain the partially optimal probability of a state sequence ending with state 1 (labeled *b* in Figure 4.3), taking into account only the first two observations of the sequence.

Similar to the training code, inference-related code using the core API can be involved in its use of types and templates. Again, the higher-level presented in Section 4.5 offer an easier to use but less flexible alternative.

4.4.5 GENERATION

Pattern models can also be used with *generation* algorithms to achieve pattern synthesis. The basic algorithm provided by the library simulates the execution of the pattern model by randomly executing transitions, and relying on the *ObservationDistributionGenerator* to generate observations.

The code sample in Figure 4.6 shows how the how the pattern models constructed in Figures 4.4 and 4.2 can be used to stochastically generate observations. The generation algorithm requires that a Boost.Random random number generator be provided, which it then uses to generate

Fig. 4.6. Generation using pattern model

```
generation::generator<ahmm_type> generator(ahmm);

boost::mt19937 engine;
engine.seed(static_cast<unsigned int>(std::time(0)));
boost::variate_generator
<
    boost::mt19937,
    boost::uniform_real<double>
>
    rng(engine, boost::uniform_real<double>());

double result = generator(rng).get();
```

an observation according to the pattern model. Any number of observations can be generated from the model, until it arrives in an ending state.

In the case of generation, some programming complexity arises from always having to provide the random number generator. The higher-level APIs related to generation and synthesis can provide their own number generator, offering a more simplified programming interface (at the expense of the flexibility of being able to control the random number generator).

4.5 Higher-level APIs

While the pattern models and algorithms can be used directly to accomplish a variety of tasks, some common tasks are directly supported through higher-level, simplified interfaces provided by the library. For common uses such as training, classification, on-line recognition, and synthesis, the library provides various *task* class templates which can be somewhat customized via template arguments. We also provide a collection of purely object-oriented C++ classes for a few of the most common usage scenarios and modalities, provided in a pre-compiled library. The library is framed as an addon for openFrameworks [60], an easy-to-use C++ library framework. However, as the library does not depend on openFrameworks, it can be used in any C++ application (only the provided examples use openFrameworks functionality). Finally, a subset of the functionality provided by the pre-compiled C++ library is also available through a C interface.

4.5.1 Task classes

4.5.1.1 Training

One of the most common tasks in pattern recognition scenarios is training pattern models from training data. The `training_task` class template offers easy-to-use functionality for training one or more pattern models. It supports different training strategies (expectation-maximization training and Viterbi training) and different pattern model topologies / special cases.

An example of using `training_task` is given in Figure 4.7.

The example shows how a single pattern model is added to the `training_task`. The `training_task` can hold multiple pattern models. In this case, one pattern model was trained from training data (two examples of the pattern). The pattern model has a chain HMM topology with two emitting states, identical to the topology shown in Figure 4.3. It also uses the same training data as the code sample in Figure 4.4, but employs expectation-maximization training rather than best match (Viterbi) training.

When comparing this to the code sample in Figure 4.4, which uses the training algorithms directly, we see that the `training_task` doesn't require us to specify an `inference_type`, or the termination condition of the training algorithm (it offers sensible defaults). As well, there is no need to initialize the model before it can be refined by training (recall that the code sample in Figure 4.4 manually invoked the naive alignment initialization). Instead, the task will automatically initialize the model from the training data, based on its training strategy. The benefit is increased ease of use, at the expense of the flexibility. However, the programmer can also provide their own

Fig. 4.7. training_task example

```
patterns::training_task
<
    // use a chain_hmm model,
    // with a normal observation distribution
    patterns::model::chain_hmm<observations::normal>,
    // use the EM algorithm for training
    patterns::expectation_maximization_training
>
    em_training_task;

// this will hold our training example
std::vector<std::vector<double> > examples(2);

using namespace boost::assign;

// two examples for pattern 0
examples[0] += 0, 0.1, -0.1, 1.1, 0.9, 1.0;
examples[1] += 0.1, -0.1, 0.1, -0.1, 0, 1.0;

// add a new pattern model with 2 states to the task,
// trained from the examples
em_training_task.add_pattern_with_examples(2, examples);
```

training strategy types (e.g., a strategy that uses different initialization), which can restore some of the flexibility present in the core API.

Another advantage of the `training_task` interface is that once the task is set up, it is easy to add additional pattern models. The code sample in Figure 4.8 shows an example of this, where two pattern models are added to the task to allow later classification.

4.5.1.2 CLASSIFICATION

`classification_task` is used to classify an unknown instance of a pattern as belonging to one of a number of pattern classes. `classification_task` is derived from `training_task`. The `training_task` functionality presented in Section 4.5.1.1 is used to add the pattern models to the `classification_task`. The pattern models correspond to the pattern classes - each pattern model defines a class.

An example of using `classification_task` is given in Figure 4.8. It shows two pattern models being trained from examples, followed by classification of a pattern instance.

Note that the inference code presented in the code sample in Figure 4.5 could be extended to accomplish classification. This would be done by training a number of pattern models, computing the likelihood values of the pattern instance to be classified using each of the models in turn, and then choosing the most likely pattern model as the result of the classification. When comparing this to the code sample in Figure 4.8, one can see that the `classification_task` accomplishes all this much more directly. This is again a benefit of ease of use that comes at the expense of flexibility that is available when using the core API directly.

4.5.1.3 ON-LINE RECOGNITION

The on-line recognition task is for scenarios where observations are being collected in real-time, with the goal of being able to detect executed instances of a pattern as they are being executed (or immediately after they are completed).

Fig. 4.8. classification_task example

```
patterns::classification_task
<
    // use a chain_hmm model,
    // with a normal observation distribution
    patterns::model::chain_hmm<observations::normal>,
    // use the EM algorithm for training
    patterns::expectation_maximization_training,
    // use the forward algorithm for inference
    patterns::forward_inference
>
    task;

// this will hold our training example
std::vector<std::vector<double> > examples(2);

using namespace boost::assign;

// two examples for pattern 0
examples[0] += 0, 0.1, -0.1, 0.2, 0.2, 0.2, 1.1;
examples[1] += 0.1, 1.2;

// add a new pattern model with 2 states,
// trained from the examples
task.add_pattern_with_examples(2, examples);

examples.clear();
examples.resize(2);

// examples for pattern 1
examples[0] += -0.4, 1.1;
examples[1] += -0.3, 1.2;

// add a new pattern model with 2 states,
// trained from the examples
task.add_pattern_with_examples(2, examples);

// this is our pattern instance to classify
std::vector<double> pattern;
pattern += -0.2, 1.1;

// task.classify(pattern) should return 1
int classification_result = task.classify(pattern);
```

The `semantic_recognition_task` solves this problem by employing the semantic network model, presented in Chapter 3. It associates semantic states with beginnings and ends of patterns, which means it can estimate when pattern execution has begun and/or ended. After each observation is given to the task, it will provide updates on any newly detected pattern commencement or completion, and any revisions on previously reported detections. It is possible that after seeing further observations, the task will report that a previously detected pattern completion was erroneous, or that it happened at a time different than what was previously reported.

Unlike the `classification_task` which can have pattern models added dynamically at any time, the `semantic_recognition_task` requires that all pattern models be trained beforehand in a separate `training_task`.

The code sample in Figure 4.9 shows how on-line recognition can be used on the same models trained in the code sample in Figure 4.8. It first uses the `task` trained in the code sample in Figure 4.8 to construct the `semantic_recognition_task`. It then processes observations one by one, with the `semantic_recognition_task` returning events that have most likely occurred (events such as beginnings or ends of a gesture).

Internally, the `semantic_recognition_task` uses the semantic Viterbi algorithm provided as one of the inference algorithms in the library. The algorithm provides the semantic traceback after each observation is processed, and changes in that traceback are translated into events. The semantic Viterbi algorithm could be used directly, but the programmer would then have to process the tracebacks manually. Again, the `semantic_recognition_task` provides an increased ease of use at a cost of flexibility.

Fig. 4.9. `semantic_recognition_task` example

```
typedef patterns::semantic_recognition_task<
        observations::normal>
    recognition_task_type;
recognition_task_type recognition_task(task, 0.01,
    ame::selectors::circular_buffer(20));

std::vector<double> pattern;
pattern += -0.35,-0.35,1.15,10.0,10.0,0.1,1.2,0.95,1.2;

std::vector<double>::iterator it=pattern.begin();

std::vector<patterns::semantic_event> result;

result = recognition_task.match(*it++);
// result.size() == 1u : one event occured
//   (beginning of pattern 2)
// result[0].type == patterns::semantic_event::detected
// result[0].pattern == 2u
// result[0].beginning == 0u
// result[0].end == -1u (no end detected yet)

result = recognition_task.match(*it++);
// result.size() == 1u : revising start of pattern 2
// result[0].type == patterns::semantic_event::detected
// result[0].pattern == 2u
// result[0].beginning == 1u
// result[0].end == -1u (no end detected yet)

result = recognition_task.match(*it++);
// result.size() == 0u : no events

result = recognition_task.match(*it++);
// result.size() == 1u : detecting end of pattern 2
// result[0].type == patterns::semantic_event::detected
// result[0].pattern == 2u
// result[0].beginning == 1u
// result[0].end == 2u

// detects beginning of pattern 1...
recognition_task.match(*it++);
recognition_task.match(*it++);
// detects end of pattern 1...
recognition_task.match(*it++);
```

4.5.1.4 SYNTHESIS

The goal of synthesis is to synthesize instances of a pattern. The synthesis task can be used for this purpose - after it is trained with models of one or more patterns, it can be used to either synthesize an entire instance of a pattern, or to generate one observation at a time.

The code sample in Figure 4.10 shows sample usage of `synthesis_task`. A pattern model is first trained from example data. Because the model is a chain model with two emitting states, and it was trained from examples that are all composed from two observations, it will always produce synthesized instances of two observations. The code sample shows two synthesizations of complete pattern instances, and then individual generation of observations after the task has been reset.

Compared to the code sample in Figure 4.6 which uses the generation algorithm directly, `synthesis_task` provides a more straightforward interface and alleviates the need to provide a random number generator. The drawback is again a loss of flexibility.

4.5.2 OFXPATTERNS

We also provide a collection of purely object-oriented C++ classes for a few of the most common usage scenarios, provided in a pre-compiled library: ofxPatterns. ofxPatterns is framed as an addon for openFrameworks [60], an easy-to-use C++ library framework. However, as ofxPatterns actually does not depend on openFrameworks, it can be used in any C++ application (only the provided examples use openFrameworks functionality).

The interface of the classes provided by the ofxPatterns library is comparable to that provided by the individual task class templates presented, so we will not present it. The benefit of using ofxPatterns compared to the core API is that due to being precompiled, ofxPatterns provides much smaller compilation times. ofxPatterns also uses an object-oriented C++ interface without templates, which can be simpler to use. However, while the task class templates can be somewhat specialized through the use of templates (allowing, e.g., a choice of training strategy and obser-

Fig. 4.10. synthesis_task example

```
patterns::synthesis_task
<
    patterns::model::chain_hmm<observations::normal>,
    patterns::expectation_maximization_training
>
    task;

std::vector<std::vector<double> > examples(2);

using namespace boost::assign;

examples.front() += 0, 1.1;
examples.back() += 0.1, 1.2;

task.add_pattern_with_examples(2, examples);

std::vector<double> synthesized;

task.synthesize(synthesized);
// synthesized.size() == 2u
// e.g. synthesized == 0.03, 1.19

task.synthesize(synthesized);
// synthesized.size() == 2u
// e.g. synthesized == 0.18, 1.02

task.reset();
for(int i=0; i<2; i++)
    task.generate();
```

vation probability distribution), the classes provided by ofxPatterns do not offer the same level of flexibility. They are fixed to use pre-determined training strategies and observation probability distributions, which are only applicable to a limited number of scenarios.

4.5.3 CPATTERNS

Finally, we have recently started developing a library offering a C interface, with the goal of the same functionality as ofxPatterns (the cPatterns library wraps the ofxPatterns classes with a C interface, but in its current stage only a few of ofxPatterns classes have been wrapped). The C interface allows the library to be used in environments where C++ is not an option. For example, it can be loaded as a shared library into C#.

In this case, the gain over ofxPatterns is portability. The drawback is that the convenient syntax of the object-oriented C++ interface is lost, and replaced by a C interface.

4.6 CONCLUSIONS

In this chapter, we have presented an overview of the AME Patterns library, its different API levels, and provided a comparison of different tradeoffs introduced by each API level.

Compared to other related libraries, the core AME Patterns library offers a high level of flexibility, which is achieved through the following:

- the use of the AHMM+EC pattern model allows the library to be used with hidden Markov models (HMMs) [37], terminating HMMs [2], hierarchical HMMs (HHMMs) [11], semi-Markov models [32], reduced-parameter models [43], and Markov models of order larger than 1 [48].

- the use of the *PatternModel* concept allows optimized data types to be used for certain pattern models or pattern model topologies.

- the use of the *ObservationProbabilityDistribution* and related concepts allows the library to be easily and efficiently used with a number of observation types / modalities (e.g., speech or body gesture)

While the flexibility through use of generic C++ programming comes at an increased cost in the ease of use, compile times, and portability (compared to C libraries), these costs are alleviated thought the use of multiple API levels. For example, the task-based API offers a slightly lower level of programming complexity at the expense of somewhat decreased flexibility, as it is only appropriate for the specific tasks provided. Furthermore, the ofxPatterns pre-compiled library offers an additional increase in the ease of use (especially to programmers more comfortable with an object-oriented C++ programming style) and decrease in compile times, at a further cost to flexibility (it is limited to specific scenarios involving pre-determined modalities). Finally, the cPatterns library offers a C-based interface, which provides an increase in portability, at the expense of an often more convenient C++ syntax.

Keeping in mind that pattern analysis spans many application areas, and that the library's flexibility enables the transfer of ideas between individual application areas, as well as providing a single implementation that can be used in all of the application areas, we conclude that the use of generic concept-based programing is highly beneficial for pattern modeling, recognition, and synthesis. Furthermore, the drawbacks accompanying the use of generic concept-based programming can be alleviated through the use of multiple API levels. Thus, the libraries presented here can be useful to both those that need a lot of flexibility (e.g., those working in multiple domains, or developing new algorithms), as well as those working in well-defined scenarios that are addressed by the library.

5. REAL-TIME AUTOMATIC KINEMATIC MODEL BUILDING FOR OPTICAL MOTION CAPTURE USING A MARKOV RANDOM FIELD

5.1 INTRODUCTION

Marker-based optical motion capture is a reliable and mature technique to capture body kinematics such as joint angles and joint locations. In a typical motion capture system, markers are placed on the body of the subject, and their trajectories recorded. The body movement is then inferred from the trajectories of the markers. Using marker-based optical motion capture system to provide frame-by-frame information of the body position and configuration in space has proved useful for many applications, such as rehabilitation, kinesiology, sports training, animation, and interactive arts.

However, using optical motion capture is usually expensive, cumbersome, and time consuming. The placement of markers on the user is intrusive, and usually requires time and care to be placed properly. Furthermore, to reliably track markers through body movement, the system requires a model of the marker set that describes the marker properties and their placement on a particular user. This model will help track and identify markers during a capture. Usually, in the *model preparation phase* of motion capture, a system operator manually enters the marker set model to the motion capture system by providing the name of each marker and specifying which pairs of markers are connected, i.e. have a relatively stable distance between them. The left panel of Figure 5.1 shows such a model that has been manually entered into the EVa Real-Time Software (EVaRT), the motion capture software for optical motion capture from Motion Analysis Corporation.

Additionally, after the model preparation phase, a *subject calibration phase* is still needed before each capture session in which the movement of the subject is calibrated against the marker set model. A subject calibration phase includes the collection of a training sequence in which the subject carefully moves through the range of motion and without occluding the markers, and a manual off-line processing step in which the operator has to clean the captured data before the model can be applied to the subject. This process can take anywhere from a few minutes to few tens of minutes.

Moreover, motion-capture data obtained in real-time is often flawed due to marker occlusion and noise. Marker misidentification and tracking errors are commonplace, and require the data to be cleaned after the capture, usually referred to as the *cleaning phase* of motion capture. Note that for real-time environments using motion capture, e.g. movement-driven human computer interaction, faulty real-time tracking cannot be addressed by cleaning the data after the capture, but motion capture systems sometimes offer some agency to the operator to correct the tracking in real-time.

In addition to these obstacles to a seamless optical motion capture, a quality optical motion capture system is often costly, with the most expensive part of the system being a sophisticated motion capture software capable of tracking the markers reliably during a capture. Recently, partly due to advances in low-cost high speed cameras, the cost of optical motion capture has dropped dramatically. The OptiTrack Foundation Package is a low-cost ($5000) real-time motion capture system released by NaturalPoint in August 2007, including 6 USB infrared cameras and Arena motion capture software. This system is capable of full body motion capture of a single subject. Meanwhile, NaturalPoint also provides an inexpensive software package Point Cloud Tracker ($300) which produces unlabelled marker data at 100 Hz. Using such low-cost motion capture is a practical solution to affordable motion capture.

To make the motion capture process more streamlined and low-cost, the research community has made improvements both in automatizing parts of the marker-based capture process, and on improving video-based markerless motion capture methods. In this chapter, we contribute to the automatization of marker-based motion capture through a comprehensive approach which reduces the need for operator intervention in motion capture, as well as reduces the time spent on recording and preparing motion capture data. First, we modify the model preparation phase to require only minimal information needed to provide consistent labeling of the markers / body parts of interest to the motion capture scenario. Second, we completely automate the subject preparation phase by algorithmic deduction of the kinematic model of the subject in the motion capture space, and automatic matching to the prepared model. Finally, we improve the robustness of real-time tracking, thereby reducing the need for post-capture data cleaning or real-time operator intervention. The proposed approach can be integrated to the low-cost motion capture system (e.g., the OptiTrack system) for reliable streamlined low-cost motion capture.

The proposed method is based on Markov random fields, and allows the model building, marker tracking, joint position inference, and body tracking to be achieved simultaneously, automatically, and in real-time. For example, the right panel of Figure 5.1 shows an automatically built model using the proposed algorithm and the same movement sequence as the manually built shown in the left panel of the same figure. The deduced kinematic model is then matched to the prepared model, allowing persistent labels to be attached to the markers or body parts. As the general probabilistic framework we propose is easily extensible, it may be applied to markerless motion capture systems in the future.

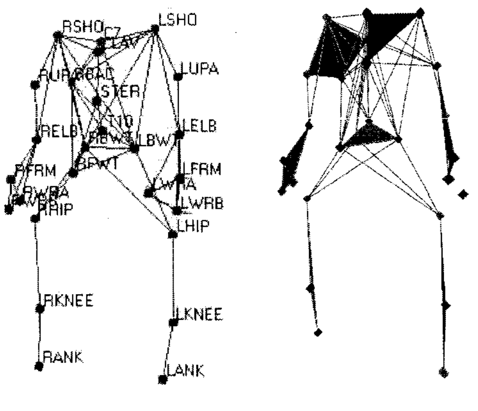

Fig. 5.1. Left: a manually entered model, with the markers named according to body placement. Pairs of markers expected to remain at a relatively constant distance have been manually connected. Right: automatically built model using the proposed algorithm. The model was built using the same movement sequence as the manually built model in the left panel. Spheres represent markers, lines represent inferred rigid links, and solid shapes represent inferred rigid bodies.

This chapter is organized as follows. In Section 5.1.1, we discuss some of the related work, then give a theoretical presentation of our kinematic model building framework in Section 5.2. An overview of the method used to match the automatically built kinematic model to a persistent model is presented in Section 5.3. Finally, the results of the implementation on some real-world and synthetic data are presented in Section 5.4, and conclusions in Section 5.5.

5.1.1 Previous Work

The progression from collected raw point data to a high-level understanding of the movement is divided into different stages [27]. *Tracking* refers to making correspondences between observed points across frames. The final goal is to obtain the paths of markers throughout the trial. *Labeling* corresponds to assigning labels to each of the observed paths. In a system that uses pre-existing models of marker configurations, the goal is to transfer the available marker labels (such as "Left Shoulder", or "Right Knee") from the model to the observed data. Otherwise, the goal would be to build a model (*modeling*) from the observed data, and maintain it as the subject leaves and re-enters the space. Finally, *pose estimation* produces a high-level representation of the motion data, such as the joint angles of the skeletal structure.

There are a number of methods which produce a model of the body, but they require tracked feature / marker data to be provided. Recently, Yan and Pollefeys [59] presented a method which computes the kinematic chain of an articulated body from 2D trajectories of feature points. Silaghi et al. [50] also give a method which accomplishes automatic model building when given tracked motion capture data as input. In addition to building a model of the skeleton, they present the motion in terms of the skeleton. In some methods, tracking is incorporated in the model building, such as the approach taken by Ringer and Lasenby [45], but the produced model is not used to improve the tracking. However, using an a priori model to help with the tracking process has been considered, for example by Zakotnik et al. [61] and Karaulova et al. [17]. Finally, the isolated problem of marker tracking or feature tracking has been studied extensively in both 2D and 3D domains. A good review of some methods is provided by Veenman et al. [55]. Shafique and Shah [47] also give a graph based multi-frame approach.

Recently, Hornung et al. [15], in conjunction with a very nice explanation of the problem and goals which are very much in line with ours, proposed a method which ties all of the above subproblems together and gives promising results. Rajko and Qian [42] also presented a heuristic approach that unifies marker tracking and some model building. Our proposed method differs in that it brings together all of the elements of the problem into a unifying probabilistic framework. This not only provides a more compact solution, but also allows probabilistic inference of the underlying (hidden) ground truth data which can give more accurate results.

5.2 Kinematic Model Building using Dynamic Markov Random Field

5.2.1 Assumptions and Notation

We assume to have available a motion capture system capable of providing 3D observations of markers attached to the objects of interest. The system provides instantaneous observation information at a constant frame rate, and for simplicity we assume that this information is provided at times $t = 1, 2, \ldots$ which we also refer to as frames. In our notation, we will consistently use (t) in the superscript to indicate the time. Our framework captures the behavior of the body under observation using a series of *models* that specify this behavior at different levels of abstraction. At the highest level, we model the body as a collection of *rigid objects* that are connected by *joints*. At the lower level, we model individual *markers*, their corresponding *observations* provided by the motion capture system, as well as pairs of markers that appear to be connected by a *rigid link*.

The body models can be specified in a little more detail as follows:

- The *observation model* defines the relationship between a marker's true position and its observation provided by the motion capture system.

- The *single marker model* defines the behavior of a single marker and assumes the motion of the marker is relatively smooth, i.e. its velocity does not change much from frame to frame.
- The *rigid link model* specifies that that the distance between markers that lie on the same rigid object should remain relatively constant.
- The *rigid object model* specifies the behavior of all markers on a rigid object in terms of the behavior of the object and the relative placement of the markers on the object.

The observations of the markers are the only observable element of the body - the existence of all other higher level models is inferred by monitoring the behavior of these observations. As the higher level models are constructed, we can use them to provide a more accurate state of the body, as well as perform more accurate marker tracking. Correspondingly, the set of models in the system is not static. It may expand as the system becomes aware of higher level models as they are inferred, or new markers/bodies enter the scene, or shrink as the models cease to be accurate or leave the scene.

To combine the various models, we will make use of a Markov random field (MRF), in which the various models will exhibit themselves through the nodes and edges of the MRF. In this section, we introduce the state and observation variables to be used in the MRF for kinematic model building. Some of the variables will be used as the states of nodes in the MRF and some of them as shared states between nodes.

We first define the state of a marker m at time t to be $\mathbf{x}_m^{(t)} = (x_m^{(t)}, \dot{x}_m^{(t)}, \sigma_m^{(t)}, o_m^{(t)})$, by its true position $x_m^{(t)}$, velocity $\dot{x}_m^{(t)}$, uncertainty in the movement characterized by $\sigma_m^{(t)}$, and a binary occlusion state $o_m^{(t)}$. When $o_m^{(t)} = 1$, it means marker m is occluded at time t and none of the observed points corresponds to marker m; when $o_m^{(t)} = 0$, marker m is visible to the motion capture system at time t and one of the observed 3D point is a noisy measurement of its 3D position. The velocity of the marker m at time t is approximated in the sense that $\dot{x}_m^{(t)} = x_m^{(t)} - x_m^{(t-1)}$. Since objects of interest can leave and enter the motion capture volume, the set of markers observed by the system at each time point is not necessarily always the same. We therefore define a marker index set M that encompasses all markers that are a part of the motion capture system at any time of the capture trial. The entire marker set is then defined as $\{\mathbf{x}_m | m \in M\}$. We can then define *the set of markers* in the system at time t to be $\mathbf{X}^{(t)} = \{\mathbf{x}_m^{(t)} |$ marker m is in the system at time $t, m \in M\}$, and the index set of these markers $M^{(t)}$.

Two markers may form a rigid link with a consistent length between the two markers. To represent such rigid links, we introduce the *set of rigid links* $\mathbf{L}^{(t)} = \{\mathbf{l}_{m,n}^{(t)}\}_{m \neq n \in M^{(t)}}$ at time t, where $\mathbf{l}_{m,n}^{(t)} = (\bar{d}_{m,n}^{(t)}, \sigma_{l,m,n}^{(t)})$ consists of respectively the expected value $\bar{d}_{m,n}^{(t)}$ and the uncertainty $\sigma_{l,m,n}^{(t)}$ of the length of the rigid link between markers m and n at time t. The parameters of the rigid link models can be considered to be shared states of the marker nodes. A rigid link is considered to be valid if and only if the uncertainty is less than a pre-chosen value.

Having defined rigid links, a rigid body is simply given by three or more non-collinear markers that have mutual rigid links. In our approach, the *set of rigid objects* at time t is denoted by $\mathbf{O}^{(t)} = \{\mathbf{o}_u^{(t)}\}$. We define a rigid object index set O that (similarly to M) contains indices of each rigid object that is present in the system at any time, so that $\mathbf{O} = \{\mathbf{o}_u | u \in O\}$ is the set of all rigid objects. We will always use subscripts $u \in O$ and $v \in O$ to indicate a specific rigid object. Each rigid object has its member markers and their 3D positions in an object-centered local coordinate system. Hence a rigid object is modeled as $\mathbf{o}_u^{(t)} = (o_u^{(t)}, \omega_u^{(t)}, \dot{o}_u^{(t)}, \dot{\omega}_u^{(t)}, M_u^{(t)}, X_u^{(t)})$ by its position $o_u^{(t)}$ (i.e., the center of the object-centered frame in the world frame), orientation $\omega_u^{(t)}$ (given as an angle vector indicating the orientation of the local frame in the world frame), translational velocity $\dot{o}_u^{(t)}$, rotational velocity $\dot{\omega}_u^{(t)}$, the member marker indices $M_u^{(t)} \subset M$, and $X_u^{(t)} = \{x_{u,m}^{(t)} | m \in M_u^{(t)}\}$ the 3D positions of the member markers of the object in the local object-centered frame. Similarly to the velocity in the marker model,

$$\dot{o}_u^{(t)} = o_u^{(t)} - o_u^{(t-1)} \tag{1}$$

$$\dot{\omega}_u^{(t)} = \omega_u^{(t)} - \omega_u^{(t-1)} \tag{2}$$

We denote *the set of complete observation vectors* at time *t* by $\mathbf{Y}^{(t)} = \{\mathbf{y}_1^{(t)}, \mathbf{y}_2^{(t)}, \cdots, \mathbf{y}_{|Y^{(t)}|}^{(t)}\}$. Each complete observation vector $\mathbf{y}_i^{(t)} = (y_i^{(t)}, \mathbf{z}_i^{(t)})$ consists of $y_i^{(t)}$ a three dimensional point observed in the motion capture reference coordinate system by the motion capture system, and a binary point-to-marker mapping vector $\mathbf{z}_i^{(t)} = (z_{i,1}^{(t)}, \cdots, z_{i,|\mathbf{X}^{(t)}|}^{(t)})$ where $z_{i,m}^{(t)} \in \{0, 1\}$. When an observed point $y_i^{(t)}$ is the observation of marker m at time t, $z_{i,m}^{(t)} = 1$. Otherwise, $z_{i,m}^{(t)} = 0$. In our approach, the complete observation vectors are only partially observed. Only a cloud of 3D points are obtained by the motion capture system. However, the point-to-marker mapping vectors are unknown and need to be solved. To be concise, we denote $Y^{(t)} = \{y_i^{(t)}\}$ to be the observed 3D point cloud at time t, and $\mathbf{Z}^{(t)} = \{\mathbf{z}_i^{(t)}\}$ the unknown point-to-marker mapping vectors associated with all the observed points.

Please note that the observations are noisy, i.e. the true position of the markers is hidden. Also, markers can be occluded (not producing an observation), and we sometimes observe "ghost markers" (observations that do not correspond to a marker).

Due to occluded markers or ghost markers, it is possible that $|\mathbf{Y}^{(t)}| \neq |\mathbf{X}^{(t)}|$.

5.2.2 OVERVIEW

In our approach, the goal is to first keep track of the markers given the current observed 3D point positions $\{y_i^{(t)}\}$ and the previous estimates of the marker, rigid link and object models, namely, $\mathbf{X}^{(t-1)}$, $\mathbf{L}^{(t-1)}$, and $\mathbf{O}^{(t-1)}$, and then update the model parameters. To tackle this challenge, we cast the marker tracking and model estimation problem into a dynamic Markov random field framework [58, 18]. First, prediction of model parameters at the current time instant can be obtained using previous estimates at the previous time. Within one time slice, to utilize all the available constraints to inform the marker tracking and parameter updating, a local Markov random field is adopted to describe the interdependency of the models and observations. In this local MRF, the joint distribution of the observation nodes and the model parameters are given by a Gibbs distribution [14] with potential functions defined to address all the constraints in marker tracking. With the MRF properly set up, theoretically the marker tracking and parameter updating can be solved in an expectation-maximization framework. Essentially, in the E-step, the probability distribution of the point-to-marker mapping vectors $\mathbf{z}_i^{(t)}$'s can be estimated using the current parameter estimates; in the M-step, $\mathbf{z}_i^{(t)}$'s are used in turn to update the parameter estimates. After a few such iterations, both marker tracking and parameter updating can be achieved. However, in practice, the EM-based tracking algorithm is computationally expensive and intractable. Therefore, in stead of following traditional EM algorithm for the inference of $\mathbf{z}_i^{(t)}$'s and the update of the model parameters, we introduce a fast approach to tackle the two problems in an interlaced manner. In our approach, we also handle ghost markers and new markers.

5.2.3 Dynamic Markov Random Field

In the previous section, we have introduced the marker states $\mathbf{X}^{(t)}$, the rigid link models $L^{(t)}$, and the rigid body states $\mathbf{O}^{(t)}$. At each time t, the state of the marker tracking and kinematic model building system is given by $\mathbf{S}^{(t)} = \{\mathbf{X}^{(t)}, \mathbf{L}^{(t)}, \mathbf{O}^{(t)}\}$. Meanwhile, we have also presented the observation vectors $\mathbf{Y}^{(t)}$, which are actually partly observed. This is because that only the 3D point cloud $\{y_i\}$ is observed while the binary point-to-marker mapping vectors $\{\mathbf{z}_i^{(t)}\}$ are unknown and need to be solved. The goal of the system is to track $\mathbf{S}^{(t)}$ from previous estimates $\mathbf{S}^{(t-1)}$ and the currently observed 3D point cloud $\{y_i\}$. To properly track $\mathbf{S}^{(t)}$, two sets of constraints need to be exploited. The first set of constraints resides on the fact the over adjacent time instants, the system states follows a first-order Markov chain in the sense that the system dynamics is represented by a state transition equation, or equivalently a conditional probability density function $p(\mathbf{S}^{(t)}|\mathbf{S}^{(t-1)})$. The second set of constraints capture the interdependency of the states and the observation vector, which is readily represented by an MRF. An MRF [7, 24] is an undirected graph over a set of nodes with potential functions defined over cliques, which are groups of mutually neighboring nodes. Prior knowledge and constraints over the states of the nodes can be described through proper formation of the potential functions. The joint probability of the nodes in an MRF follows Gibbs distribution [14]. MRF has found many important applications in computer vision [7, 24], including low-level vision problems, such as texture analysis and synthesis, image segmentation, and restoration [14], and high level vision problems, e.g. object recognition, and pose estimation [24]. Our approach to marker tracking and kinematic model building is based on a *dynamic Markov random field* (DMRF) framework, named after the fact that we treat $\mathbf{S}^{(t)}$ to be the states of a dynamic system, and that for each time slice, the elements of the state vector and observation vector form a local MRF.

A DMRF is in spirit similar to a dynamic Bayesian network (DBN) [31]. The main difference between DMRF and DBN is the way conditional dependency is represented within each time slice. For DBN, a Bayesian net is used to represent the local conditional dependency of the state variables in DBN. While in DMRF, an MRF is used to represent such dependencies, which can be flexible and capture rich correlations among state variables. Our goal will be to construct the most likely estimate of the system to have produced the observations at each frame. The similar DMRF framework has been successfully used for modeling the dynamic labeling problem, e.g. image segmentation on video sequences [18], and tracking of articulated motion [58], where both temporal dynamics and local spatial constraints are elegantly represented in a DMRF framework.

5.2.4 Constructing the Dynamic Markov Random Field for Kinematic Model Building

In this section, we construct a DMRF representing the joint distribution of $\mathbf{S}^{(t)}$ and $Y^{(t)}$ for all t from the initial time instant to the current time instant. Let $\mathbf{S}^{(1:t)} = (\mathbf{S}^{(1)}, \mathbf{S}^{(2)}, \ldots, \mathbf{S}^{(t)})$ be the state sequence from the start to the current time instant, and $Y^{(1:t)} \doteq (Y^{(1)}, Y^{(2)}, \ldots, Y^{(t)})$ the observed 3D point cloud. According to the chain rule and Markovian property of the system,

$$\begin{aligned} p(Y^{(1:t)}, \mathbf{S}^{(1:t)}) &= p(Y^{(t)}, \mathbf{S}^{(t)} | \mathbf{S}^{(1:t-1)}, Y^{(1:t-1)}) p(\mathbf{S}^{(1:t-1)}, Y^{(1:t-1)}) && (3) \\ &= p(Y^{(t)}, \mathbf{S}^{(t)} | \mathbf{S}^{(t-1)}) p(\mathbf{S}^{(1:t-1)}, Y^{(1:t-1)}) && (4) \end{aligned}$$

After factorizing (4) into the likelihood function and the system dynamic, we have

$$p(Y^{(t)}, \mathbf{S}^{(t)} | \mathbf{S}^{(t-1)}) = p(Y^{(t)} | \mathbf{S}^{(t)}) p(\mathbf{S}^{(t)} | \mathbf{S}^{(t-1)}) \qquad (5)$$

Hence, to obtain the conditional distribution $p(Y^{(t)}, \mathbf{S}^{(t)} | \mathbf{S}^{(t-1)})$, we need to obtain $p(Y^{(t)} | \mathbf{S}^{(t)})$ and $p(\mathbf{S}^{(t)} | \mathbf{S}^{(t-1)})$. In our approach, $p(Y^{(t)} | \mathbf{S}^{(t)})$ is obtained as the conditional distribution in an MRF to exploit constraints represented by the single point models, rigid link models, and rigid object models. The state transition distribution $p(\mathbf{S}^{(t)} | \mathbf{S}^{(t-1)})$ is simply given by a set of conditional distributions.

5.2.4.1 Local MRF within one time slice

At any time instant, we represent the joint distribution $p(\mathbf{Y}^{(t)}, \mathbf{S}^{(t)})$ using an MRF over three types of nodes, the complete observation nodes $\mathbf{Y}^{(t)}$, the marker nodes $\mathbf{X}^{(t)}$, and the rigid object nodes $\mathbf{O}^{(t)}$. In this MRF, the elements in $\mathbf{Y}^{(t)}$ are fully connected to the elements in $\mathbf{X}^{(t)}$ and $\mathbf{O}^{(t)}$ since the actual point-to-marker mapping is unknown and to be solved. Two elements in $\mathbf{X}^{(t)}$ are connected if there is a rigid link between them. A rigid object $\mathbf{o}_u^{(t)}$ is only connected to its member markers in $\mathbf{X}^{(t)}$. Elements in $\mathbf{O}_{(t)}$ do not connect to each other. Similarly, elements in $\mathbf{Y}_{(t)}$ are not connected to each other either. Two nodes are neighbors if they are connected. Figure 5.2 shows a diagram of such an MRF within one time slice.

A group of nodes which share mutual neighborhoods form a clique. We will denote the set of all cliques by C, and the set of all two-node cliques for a particular frame as $C^{(t)}$. $c \in C^{(t)}$ if and only if at least one node in c is from frame t, and both nodes in c come from either frame t or frame $t-1$. We will write all potentials as $V_c^{(t)}$, where $c \in C^{(t)}$. We define potential functions for the following types of cliques, the marker-observation clique $(\mathbf{x}_m, \mathbf{y}_i)$, the marker-marker-point clique (a.k.a rigid-link-observation clique), and the marker-object-observation clique. Figure 5.3 shows these three cliques.

In this section, to be concise, we remove the subscript (t) since all variables are for the current time instant. The potential functions for the marker-observation clique $(\mathbf{x}_m, \mathbf{y}_i)$ is defined as the following. Let $\mathbf{y}_i = (y_i, z_{i,m})$ be a complete observation. The associated potential is given by

$$V(\mathbf{x}_m, \mathbf{y}_i) = -\log\left(\mathcal{N}(y_i; x_m, \sigma_m)\right)^{z_{i,m}} - \log p_o^{1-z_{i,m}} \quad (6)$$

$$= -z_{i,m} \log \mathcal{N}(y_i; x_m, \sigma_m) - (1 - z_{i,m}) \log p_o \quad (7)$$

where p_o is the *a priori* probability that a marker is occluded.

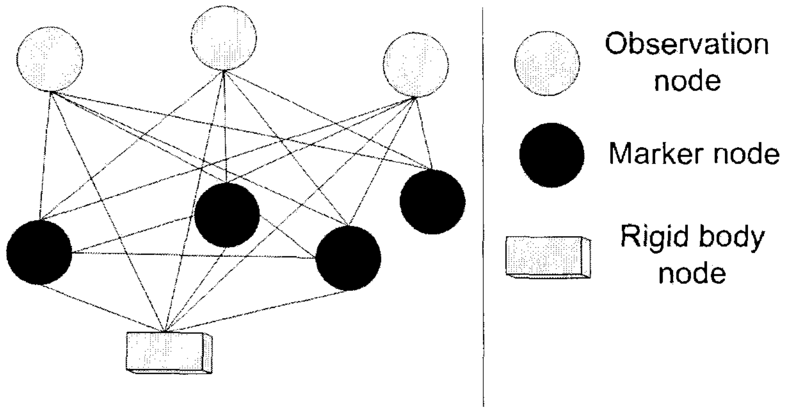

Fig. 5.2. A diagram of an example MRF in one time slice.

If two markers lie on the same rigid object (e.g., rigid body part), we say that the two markers are connected by a rigid link, and employ the *rigid link* model which gives constraints on the distance between two markers. The rigidity of the link dictates that the distance between such a pair of markers should stay relatively constant.

$$\begin{aligned} V(\mathbf{x}_m, \mathbf{x}_n, \mathbf{y}_i) &= -\log\left(\mathcal{N}(\alpha_{m,n}; \bar{d}_{m,n}, \sigma_{l,m,n})\right)^{z_{i,m}} - \log\left(\mathcal{N}(\beta_{m,n}; \bar{d}_{m,n}, \sigma_{l,m,n})\right)^{z_{i,n}} \\ &\quad -\log\left(\mathcal{N}(\gamma_{m,n}; \bar{d}_{m,n}, \sigma_{l,m,n})\right)^{(1-z_{i,m})(1-z_{i,n})} \qquad (8) \\ &= -z_{i,m}\log\mathcal{N}(\alpha_{m,n}; \bar{d}_{m,n}, \sigma_{l,m,n}) - z_{i,n}\log\mathcal{N}(\beta_{m,n}; \bar{d}_{l,m,n}, \sigma_{m,n}) \\ &\quad -(1-z_{i,m})(1-z_{i,n})\log\mathcal{N}(\gamma_{m,n}; \bar{d}_{l,m,n}, \sigma_{m,n}) \qquad (9) \end{aligned}$$

where $\alpha_{m,n} = |y_i - x_n|$ is the measured length of the link between markers m and n at time t when $z_{i,m} = 1$, $\bar{d}_{m,n}$ the expected length of the link, and $\sigma_{l,m,n}$ its standard deviation. Similarly, $\beta_{m,n} = |y_i - x_m|$ is the measured length of the link between markers m and n at time t when $z_{i,n} = 1$. $\gamma_{m,n} = |x_m - x_n|$ is the measured length of the link between the two markers when both $z_{i,m}$ and $z_{i,n}$ are zero. Please note that in general, parameters such as $\bar{d}_{mn}$ and σ_{mn} should not vary much with time. However, it is possible that markers change their relative position on the body during the capture (for example, a marker placed at the end of a sleeve of a tight shirt will permanently come closer to the elbow if the arms reach overhead and cause the sleeve to ride up the arm). For this reason, we allow the properties of the link to vary slightly with time.

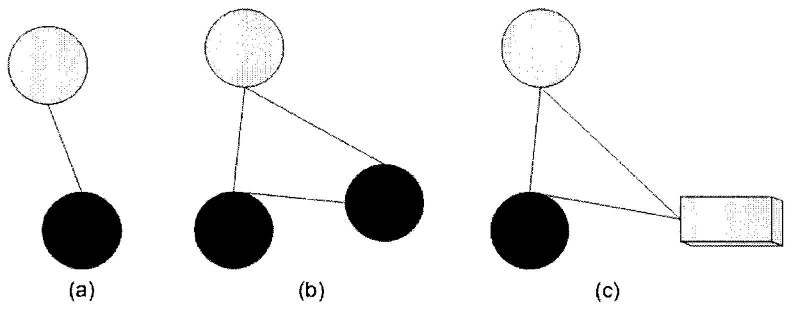

Fig. 5.3. Three cliques with potential functions defined.

The rigid body models also provides additional constraints on the point-to-marker mapping. When marker m is a member of object u, the associated potential is given by

$$V(\mathbf{x}_m, \mathbf{o}_u, \mathbf{y}_i) = -\log\left(\mathcal{N}(y_i; \bar{x}_{u,m}, \sigma_{m,n})\right)^{z_{i,m}} - \log\left(\mathcal{N}(x_m; \bar{x}_{u,m}, \sigma_{m,n})\right)^{(1-z_{i,m})} \quad (10)$$

$$= -z_{i,m}\log\mathcal{N}(y_i; \bar{x}_{u,m}, \sigma_{m,n}) - (1 - z_{i,m})\log\mathcal{N}(x_m; \bar{x}_{u,m}, \sigma_{m,n}) \quad (11)$$

where $\bar{x}_{u,m}$ is the expected location of marker m given the state parameters in $\mathbf{o}_u$ as follows.

$$\bar{x}_{u,m} = \mathbf{R}(\omega_u)x_{u,m} + o_u \quad (12)$$

where $\mathbf{R}(\omega)$ is the rotation matrix from the local object-centered frame to the global frame.

5.2.4.2 System dynamics between two consecutive time slices

State parameters propagate according to the system dynamics from time $t-1$ to time t. For the marker nodes,

$$x_m^{(t)} \sim \mathcal{N}(x_m^{(t)}; x_m^{(t-1)} + \dot{x}_u^{(t)}, \sigma_m^{(t-1)}) \quad (13)$$

For the rigid body nodes,

$$o_u^{(t)} \sim \mathcal{N}(o_u^{(t)}; o_u^{(t-1)} + \dot{o}_u^{(t)}, \sigma_o) \quad (14)$$

$$\omega_u^{(t)} \sim \mathcal{N}(\omega_u^{(t)}; \omega_u^{(t-1)} + \dot{\omega}_u^{(t)}, \sigma_\omega) \quad (15)$$

where $\sigma_m^{(t-1)}$, σ_o, and σ_ω are covariance matrices describing uncertainties in the prediction. The remaining state parameters are assumed to be unchanged from one time instant to the next.

5.2.5 Kinematic Model Building Using DMRF

Using the DMRF described in the previous section, the problem of tracking markers and building kinematic model from unlabeled 3D point observations Y amounts to identifying active marker-observation edges, and updating the parameters in S. Our goal will be to construct the most likely estimate of the system states to have produced the observations at each frame. s.t. $p(Y^{(1:t)}, \mathbf{S}^{(1:t)})$

is maximized. Thus we can divide the problem of maximizing $p(Y^{(1:t)}, \mathbf{S}^{(1:t)})$ by maximizing $p(Y^{(t)}, \mathbf{S}^{(t)}|\mathbf{S}^{(t-1)})$ for each t in a recursive fashion given the previous state estimates. Let the objective function be

$$\mathcal{L} = \log p(Y^{(t)}, \mathbf{S}^{(t)}|\mathbf{S}^{(t-1)}) = \log p(Y^{(t)}|\mathbf{S}^{(t)}) + \log p(\mathbf{S}^{(t)}|\mathbf{S}^{(t-1)}) \tag{16}$$

This maximization problem can be solved in an expectation-maximization (EM) framework. Let $\hat{\mathbf{S}}^{(t)}_{k-1}$ be the estimate of the state parameter after the $k-1^{th}$ EM iteration. Following the standard derivation of the EM algorithm, it can be easily shown that the updated estimate $\hat{\mathbf{S}}^{(t)}_{k}$ found in the following E-M iteration always increase the objective function given by (16)

$$\begin{aligned} \hat{\mathbf{S}}^{(t)}_{k} &= \arg\min_{\hat{\mathbf{S}}^{(t)}} E_{\mathbf{Z}^{(t)}|Y^{(t)},\hat{\mathbf{S}}^{(t)}_{k-1}} \log p(Y^{(t)}, \mathbf{Z}^{(t)}, \mathbf{S}^{(t)}|\mathbf{S}^{(t-1)}) && (17) \\ &= \arg\min_{\hat{\mathbf{S}}^{(t)}} E_{\mathbf{Z}^{(t)}|Y^{(t)},\hat{\mathbf{S}}^{(t)}_{k-1}} \log p(Y^{(t)}, \mathbf{Z}^{(t)}|\mathbf{S}^{(t)}) + \log p(\mathbf{S}^{(t)}|\mathbf{S}^{(t-1)}) && (18) \end{aligned}$$

Thus, in the E-step of the k^{th} iteration, the probabilities of the mapping vectors $\mathbf{Z}^{(t)}$ need be computed as follows.

$$p(z^{(t)}_{i,m} = 1|\hat{\mathbf{S}}^{(t)}_{k-1}, Y^{(t)}) = \frac{p(y^{(t)}_i, z^{(t)}_{i,m} = 1|\hat{\mathbf{S}}^{(t)}_{k-1})}{\sum_m p(y^{(t)}_i, z^{(t)}_{i,m} = 1|\hat{\mathbf{S}}^{(t)}_{k-1})}, \forall i, m \tag{19}$$

Given the MRF modeling the dependency of the observation point cloud and the state parameters, eqn. (19) can be evaluated. The conditional distribution of a node given its neighbors follows the Gibbs distribution, namely,

$$\begin{aligned} p(y^{(t)}_i, z^{(t)}_{i,m} = 1|\mathbf{S}^{(t)}) &= \frac{\exp\{-\sum_{c\in C^{(t)}_i} V^{(t)}_c\}}{Z(C^{(t)}_i)} && (20) \\ &\propto \exp\{-\sum_{c\in C^{(t)}_i} V^{(t)}_c\} && (21) \end{aligned}$$

where $Z(C^{(t)}_i) = \sum_m \int_{y^{(t)}_i, z^{(t)}_{i,m}=1} \exp\{-\sum_{c\in C^{(t)}_i} V^{(t)}_c\} dy^{(t)}_i$ is the partition function determined by the values of the neighbor nodes of $\mathbf{y}^{(t)}_i$. $V^{(t)}_c$ are potential functions over the collection of cliques $C_i^{(t)}$, which includes all the cliques involving $\mathbf{y}_i^{(t)}$. The potentials functions are defined in the previous section.

In the M-step, we need to solve for $\hat{\mathbf{S}}_k^{(t)}$ by maximizing (18) using the probabilities of the mapping vectors $\mathbf{Z}^{(t)}$ obtained from the E-step. It follows that

$$E_{\mathbf{Z}^{(t)}|Y^{(t)},\hat{\mathbf{S}}_{k-1}^{(t)}} \log p(Y^{(t)}, \mathbf{Z}^{(t)}|\mathbf{S}^{(t)}) = E_{\mathbf{Z}^{(t)}|Y^{(t)},\hat{\mathbf{S}}_{k-1}^{(t)}} \sum_i \log p(y_i^{(t)}, \mathbf{z}_i^{(t)}|\mathbf{S}^{(t)}) \quad (22)$$

$$= \sum_i E_{\mathbf{z}_i^{(t)}|y_i^{(t)},\hat{\mathbf{S}}_{k-1}^{(t)}} \log p(y_i^{(t)}, \mathbf{z}_i^{(t)}|\mathbf{S}^{(t)}) \quad (23)$$

Although this EM framework is theoretically sound, to actually solve the marker-to-point mapping and kinematic model building using this EM method is practically difficult. The main difficulty resides in the M-step. To solve for $\hat{\mathbf{S}}_k^{(t)}$, one has to take derivatives of the objective function (16) with respect to the state parameters and then solve the collection of equations obtained by setting the derivatives to zero. Due to the existence of the partition function, taking derivatives of the objective function becomes intractable. In our research, we propose an alternative approach to the maximization of the objective function. The proposed approach can be considered to be an approximation to the original EM algorithm.

An outline of the entire algorithm is summarized in Figure 5.4. Note that the first step of the tracking algorithm is to create a new marker node for each observation. This is because that we allow new markers to enter the capture volume anytime during tracking. To handle possible new markers in the observation, we create a set of new marker nodes for the observed 3D points. These new markers nodes will compete against existing marker nodes for the observed 3D points. Since these new markers nodes are just created, neither are they connected to any other marker nodes by rigid links, nor do they belong to any existing rigid body nodes. Hence at time t, the marker node set $\mathbf{X}^{(t)}$ include both existing marker nodes propagated from time $t-1$ and these newly created marker nodes.

```
for each frame $t$ do
  Create a new marker from each observation $y_i^{(t)}$
  Create a marker node $\mathbf{x}_m^{(t)}$ for each marker $\mathbf{x}_m^{(t-1)}$
  Compute the observation to marker mapping $\mathbf{z}_i^{(t)}$
  Erase new markers that are unassigned, and markers that have been occluded too long
  for frames $t, t-1$, and $t-2$ do
    Update MRF parameters from newly observed values
    Update MRF random variables using updated parameters
  end for
  Detect new models (rigid links, rigid objects, joint positions...)
end for
```

Fig. 5.4. The automatic kinematic model building algorithm. It is an approximate adaptation of the expectation-maximization algorithm.

5.2.5.1 Initialization

To initialize the MRF graph, we start with the marker observations provided for frame 1, $Y^{(1)}$. For each observation $y_k^{(1)} \in Y^{(1)}$ we create a marker model $\mathbf{x}_i^{(1)}$ and set $x_i^{(t)} = y_i^{(t)}$, $\dot{x}_i^{(t)} = (0,0,0)$, and $\sigma_i^{(t)} = \text{diag}(1,1,1)$.

5.2.5.2 Marker Tracking

For each incoming frame of observations $Y^{(t)}$, we similarly create new marker models for each observation. We also extend each marker in $X^{(t-1)}$ to $X^{(t)}$ by initializing $x_m^{(t)}$ to the most likely position given the single point model, and computing $\dot{x}_m^{(t)}$ correspondingly.

The first task for a given frame is to assign markers to observations, i.e. find the mapping $\mathbf{z}_i^{(t)}$. We start with $\mathbf{z}_i^{(t)}$ unknown for all the observations, and then determine the marker-to-point mapping for each $\mathbf{x}_m^{(t)}$ in turn. In this step, we also consider the case when a marker is occluded. For this purpose, we augment the observed point cloud $Y^{(t)}$ by forming $\tilde{Y}^{(t)} = Y^{(t)} \cup \phi$, where ϕ represents the *occlusion* observation. If a marker is mapped to ϕ by the algorithm, it means this marker is occluded in this frame. Once an assignment of a marker to an observation (or occlusion) has been determined, we call that marker *assigned*, and until then it we call it *unassigned*. Similarly, a rigid object is considered assigned if and only if at least three of the markers associated with the rigid object have been assigned. A two node clique c of markers is assigned if both of its node markers are assigned.

The marker-to-point mapping is solved using a greedy approach based on confidence, such as the one used in [42]. Essentially, for each marker m and each observation i in the augmented observation $\tilde{Y}^{(t)}$, the algorithm minimize the value of the potentials associated with the marker, with the conjecture that $z_{i,m}^{(t)} = 1$ and $z_{j,m}^{(t)} = 0, \forall j \neq i$. In the case when the occlusion observation is paired up with a marker m, $z_{j,m}^{(t)} = 0$ for $y_j^{(t)} \in Y^{(t)}$, which means none of the actually observed points corresponds to marker m.

Let $C_a^{(t)}$ be the assigned clique set consisting of assigned cliques. Define the sum of potentials related to a marker m at time t as

$$U(\mathbf{x}_m^{(t)}) = \sum_{c \in C^{(t)}, \mathbf{x}_m^{(t)} \in c} V_c^{(t)}. \tag{24}$$

Let $U_i(\mathbf{x}_m^{(t)})$ be the sum of potentials related to $\mathbf{x}_m^{(t)}$ over assigned cliques and with the conjecture that $z_{i,m}^{(t)} = 1$ and $z_{j,m}^{(t)} = 0, \forall j \neq i$, i.e.,

$$U_i(\mathbf{x}_m^{(t)}) = \sum_{c \in C_a^{(t)}, \mathbf{x}_m^{(t)} \in c} V_c^{(t)}, \text{ and } z_{i,m}^{(t)} = 1, z_{j,m}^{(t)} = 0, \forall j \neq i \tag{25}$$

Determining $\mathcal{Y}(X^{(t)})$ then goes as follows. For each possible marker-observation mapping pair $(\mathbf{x}_m^{(t)}, y_i^{(t)})$, we compute $a_{m,i}$ defined as the following.

$$a_{m,i} = \min_{x_m^{(t)}} U_i(\mathbf{x}_m^{(t)}) - \log p(x_m^{(t)} | \mathbf{x}_m^{(t-1)}) \tag{26}$$

The minimization $U_i(\mathbf{x}_m^{(t)})$ is obtained using a gradient decent method over $x_m^{(t)}$. In other words, we find the most likely location of marker m at time t given the potentials with other assigned markers and/or rigid objects, as well as the observation model relative to $y_i^{(t)}$. Then a mapping matrix $A = \{a_{m,i}\}$ can be constructed. Once A is established, an assignment can then be made

made with a degree of confidence and unambiguity. Let a_{m,i^*_m} be the best and a_{m,j^*_m} the second best matches for $\mathbf{x}_m^{(t)}$ according to the values in A, i.e.,

$$i^*_m = \arg\min_i a_{m,i} \tag{27}$$

$$j^*_m = \arg\min_{i \neq i^*_m} a_{m,i} \tag{28}$$

A confident assignment would be one where the potential of the best match is small compared to the difference between the best and second best matches. Hence, for each marker $\mathbf{x}_m^{(t)} \in X^{(t)}$ we define the following confidence factor

$$\xi(\mathbf{x}_m^{(t)}) = (a_{m,j^*_m} - a_{m,i^*_m}) - a_{m,i^*_m} = a_{m,j^*_m} - 2a_{m,i^*_m} \tag{29}$$

which indicates how confidently we can make a correspondence between $\mathbf{x}_m^{(t)}$ and $y_{i^*_m}$ so that the larger the $\xi(\mathbf{x}_m^{(t)})$ is, the more confident we are about this assignment.

Among all the currently unassigned markers, we choose a marker $\mathbf{x}_m^{(t)}$ for which $\xi(\mathbf{x}_m^{(t)})$ is maximal, and set $z_{i^*_m,m} = 1$. Since a marker is being assigned in this step, we update rows in A corresponding to the remaining unassigned markers, and recalculate all the affected error factors. We can then make the the next correspondence for the unassigned marked with the minimal confidence factor. The process is repeated until all markers are assigned. Markers that have been created in the current frame (using an observation) that are marked as occluded (because an older marker was assigned to that same observation) are immediately erased. Markers that appear to have left the system (by being occluded for too long) are also erased.

At this point, for the last 3 frames we update the estimates of various model parameters (lengths of rigid links, local coordinates of markers on rigid objects, etc.), and maximize the potentials dependent on the random variables in the system (positions of markers, positions and orientations of rigid bodies, etc.), which improves the tracking results with minimal latency.

The outline of this step is as follows. We first compute the means of the model parameters for the most recent few frames, and then update the values of the parameters in the model to take the newly observed information into account. This is analogous to the expectation step of the traditional EM algorithm. We then update the random variables for the same recent frames by maximizing the potentials of markers and rigid objects using the new values of the model parameters. This is analogous to the maximization step of the EM algorithm.

In our implementation, we limit this procedure to the last 3 frames because essentially, we are performing smoothing while updating the model parameters at the same time. To take advantage of smoothing, we should wait for the data to be smoothed before using it. However, to keep the system as real-time as possible, we limit the latency to 3 frames (30 milliseconds in our system).

5.2.5.3 Model detection

Finally, we watch for existence of new models in the system, such as rigid links and rigid objects. In most cases we do so by randomly choosing candidates to observe (e.g., a pair of markers that have not been examined before), and inspect their history in the DMRF to detect whether their behavior satisfies a particular model (e.g., their distance remains relatively constant).

For the rigid links, we randomly choose a marker, and then test whether it appears to satisfy the rigid link property with another marker. We examine the distance between the two markers for a period of time (say, last 5 seconds) and if at least one of the markers has been moving and the distance between the markers has remained relatively constant, a rigid link model is introduced between the two markers. We insist that at least one of the markers has been moving because in a completely stationary body, all markers will have stable pairwise distances regardless of whether they truly connected by a rigid link. Requiring that at least one marker be moving before we hypothesize the presence of the rigid link eliminates false positives from this stationary scenario. After a rigid link has been established, it can later be removed if the behavior of the two markers stops following the rigid link model (for example, if the variance of the distance between them grows large).

Rigid objects are introduced after three or more non-collinear markers have been mutually connected by a rigid link model. With at least three non-collinear markers, we can track a consistent local coordinate system (set of axes) that follows the movement of the rigid object. The markers are assigned coordinates in the coordinate system of the rigid object (or more accurately, a distribution of coordinates). According to the rigid object model, those coordinates should remain relatively stable (i.e., follow the distribution) as the body is moving. Since the critical goal is to consistently track the rigid object once it has been detected, rather than to set up the rigid object's coordinate system in any particular way, we arbitrarily choose one of the three markers as the origin of the coordinate system, point the x axis in the direction of another marker, and finally choose the $x-y$ plane so that the final marker lies in it. The past movement of the markers is then examined to determine what distribution of local coordinates the marker position follows (as the coordinate system is fit through the past data by a least squares technique, the local coordinates of each of the three markers will typically vary).

If another marker later develops a rigid link with the markers of the rigid object, it will join the rigid object model. And as with the rigid link model, if the markers of a rigid object are later observed to no longer follow the model, the model is removed.

After a pair of rigid objects has been observed for some time (e.g., a few seconds) where at least one of them has been moving, we can attempt to determine whether a joint exists between them. To determine whether a joint exists, we seek the existence of a point that has a stable coordinate in both of the rigid object's coordinate systems. To find such a point, we use the history of the movement of the two rigid bodies.

Let us label the two rigid bodies as A and B. Given rigid body A, each fixed point in its coordinate system can be tested for stability in the coordinate system of rigid object B. The most stable point is the best candidate for the joint's local position within rigid object A. We can similarly find the best candidate for the joint's local position within rigid object B. If the best candidates are both stable enough (and close enough to corresponding to the same point), we can hypothesize the existence of a joint between the two rigid objects. Its true position can be estimated from its inferred location in each object's coordinate system, and thus tracked from the time of its detection. Note, however, that apart from tracking we do not use the joint in the MRF framework, as we have not yet implemented a corresponding joint model.

5.3 Matching to a Persistent Model

Even after building a complete and accurate kinematic model by observing the movement of the markers, the obtained information can be useless if we don't know what real-world body (or body part) the model (or it's components) correspond to. Real-world kinesiology experiments might like to track the movement of a specific body part, e.g. the right heel. Interactive environments might want to track what the hands are doing. Animators might like to know the exact location of each of the major body segments at each frame. We therefore need a way to match the automatically inferred kinematic model to a persistent, pre-labeled model. We propose that a simple, flexible method based on heuristics is sufficient (and will later show it working on a real-world scenario).

The method we use is as follows. Let the persistent model be composed of a set of *elements* and a set of *properties*. An element is something that corresponds to elements of an automatically inferred kinematic model (e.g., a marker, or a rigid object). In addition to its type, each element has a persistent label (e.g., "left wrist", or "right upper arm"). In a particular motion capture session, once the kinematic model has been constructed, the goal is to determine the correspondence between the elements of the persistent model and the constructed model (and thereby transfer the labels to the constructed model).

The correspondence is accomplished using the properties of the persistent model. A property is some verifiable relationship of a number of specific elements. For example, a property could state that "the left heel marker tends to be below the left knee marker", or that "the rigid link distance between the right knee marker and the right hip marker is longer than the rigid link distance between the right hip marker and the left hip marker".

The properties then allow us to test whether elements of the constructed model satisfy the relationships that the elements of the persistent model are supposed to have. If we are given two markers (using the properties we listed above), we could decide which one is more likely to be the left heel than the left knee by examining which one tends to be lower to the ground. Given a pair of rigid links that shares a marker, we can decide that the right knee is more likely the marker further away from the shared marker. In other words, given a hypothesis of a concrete correspondence between the elements of the constructed model and the elements of the persistent model, we can use the properties to evaluate the hypothesis. The best method for finding the best match (hypothesis) will depend on the exact properties used.

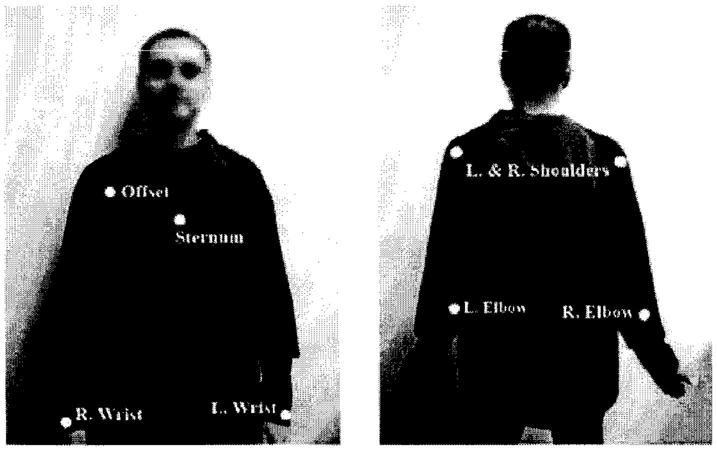

Fig. 5.5. Placement of the markers in the marker set used for the example persistent model.

We will now provide an example of a persistent model involving a simple marker set, and describe the elements and properties involved, and the strategy to find the match. The persistent model involves the upper portion of the torso, and the two arms. The markers are placed on the sternum, on each shoulder, each elbow, and each wrist. There is also an offset marker placed on the upper torso, to help break the sagital symmetry of the marker set.

The elements of the marker set are shown in figure follows:

element type	label	description
marker	Sternum	marker placed on the sternum
marker	Offset	offset marker
marker	L. Shoulder	marker placed on the left shoulder
marker	L. Elbow	marker placed on the left elbow
marker	L. Wrist	marker placed on the left wrist
marker	L. Shoulder	marker placed on the left shoulder
marker	L. Elbow	marker placed on the left elbow
marker	L. Wrist	marker placed on the left wrist
rigid object	torso	rigid object composed of the upper torso markers

TABLE 5.1

Elements of the example persistent model.

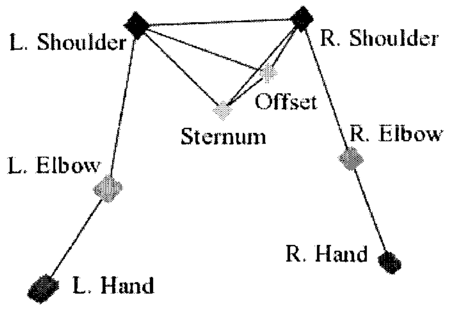

Fig. 5.6. Rigid link properties of the example persistent model (with markers viewed from behind). Displayed connections indicate the exact rigid links that must be present to for a successful match with the example persistent model.

We will now describe the properties of the persistent model. The first property is related to the existence of particular rigid links, as illustrated in Figure 5.6. As the figure indicates, there are 10 rigid links expected to be found in any kinematic model matching the persistent model, and they must match the structure shown (e.g., the L. Wrist marker must have only one rigid link and that is with the L. Elbow marker). The second property has to do with differentiating the Offset marker from the Sternum. The property states that the sum of rigid link distances of the Sternum marker is larger than the sum of rigid link distances of the Offset marker (note that both of these markers are about the same distance from the L. Shoulder marker, but the Sternum is further away from the R. Shoulder marker). The third property has to do with differentiating the left side from the right. It states that the rigid link distance between the Offset marker and the R. Shoulder marker is smaller than the rigid link distance between the Offset marker and the L. Shoulder marker. Finally, we have a property that states that the upper torso object is composed of the Shoulder, Sternum and Offset markers.

To match an automatically constructed kinematic model to the example persistent model, we go through the following steps. First, the constructed model must contain 8 markers. Next, in the connectivity graph induced by the markers and the rigid links, there must be two markers of degree 1, two markers of degree 2, two markers of degree 4, and two markers of degree 3. If this is satisfied, and the connectivity graph matches the one given in Figure 5.6, we can complete the match. We apply the second property to the two markers of degree 3 to decide which one of them is the Sternum and which is the Offset. We then apply the third property to the Offset and the two markers of degree 4 to decide which of the degree 4 markers is the L. Shoulder and which is the R. Shoulder. The rest of the marker element matches follow easily, as does the rigid object element match.

5.4 Experimental Results

We will first present the results of running the algorithm on two motion capture trials to demonstrate the general model-building capabilities of the algorithm. Figure 5.1 (right) shows the model built automatically from a 20 second full-body capture sequence, in which the subject was asked to move using all of their joints. You will notice that most of the rigid links (connections) that had to be entered manually for the model in Figure 5.1 (left) are automatically found. The one exception is a wrist marker on the left arm, which remained unconnected because it was particularly noisy.

Also, the proposed algorithm successfully partitioned the markers into rigid objects corresponding to the two shoulders, two lower arms, the pelvic area, and the two legs. The few errors concern markers that are placed close to a rigid object but are not a part of it. For example, the right upper arm marker was assigned to the right shoulder, and the left upper arm marker was assigned to the left lower arm. Similarly, even though the markers on the legs were actually placed on both the upper leg and the lower leg, the algorithm assigned them to the same rigid object because the range of motion sequence did not involve much bending at the knees.

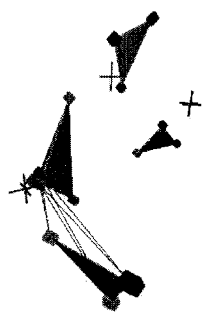

Fig. 5.7. Model built during an arm reaching movement. The solid shapes represent the lower arm, upper arm, right shoulder, and lower back. The 3D crosses represent the automatically estimated positions of joints.

The automatically built model from an arm reaching movement is shown in Figure 5.7. Here we also show the joint locations that were estimated by the system (the elbow, the shoulder, and a joint that was found between the shoulder and the lower back and estimated to be somewhere in the torso).

Second, we use synthetic data to present impact of the automatically built model on the accuracy of marker tracking. We present three synthetic scenarios, each consisting of 1000 frames. In

scenario	*observation*	τ=0	τ=1	τ=2
1 marker	17.57	16.24	13.31	13.18
4 markers - unoccluded	17.03	16.52	14.94	14.96
3 markers - unoccluded	16.95	16.43	14.80	14.81
5 markers - occluded	N/A	19.70	18.61	18.54
3 markers - occluded	N/A	50.51	49.50	48.31

TABLE 5.2

Root Mean Square Error (in mm) of the observation data, as well as the inferred marker position with 0, 1, or 2 frames of latency. The algorithm shows a moderate improvement of tracking accuracy over raw observations, and maintains an acceptable error even when the marker becomes occluded.

the *single marker scenario*, a single marker is following a trajectory given by $[t + \sin(t/100), 0, 0]$, i.e. it is traveling in a steady direction while accelerating and decelerating as given by the sin function. In the *four marker scenario*, four markers are traveling at a constant distance from each other, individually following the same trajectory as the marker in the single point scenario. A marker roughly between the other three becomes occluded between frames 600 and 900. In the *three point scenario*, three co-linear markers are traveling at a constant distance from each other (following the same individual trajectory described above). The middle marker becomes occluded between frames 600 and 900. In all scenarios, the ground truth data was masked by Gaussian noise with a standard deviation of roughly 17mm (very noisy for marker data).

For a marker of interest (in the multiple marker scenarios, it is the marker that becomes occluded), we compare the ground-truth location with the location provided by the algorithm with 0, 1, and 2 frames of latency. Note that each frame of latency involves another pass of the adapted EM algorithm on the position of the marker. The results are given in Table 5.2. We also give a graph of the error for the entire three marker trial in Figure 5.8. The results show the positive impact of EM algorithm passes on tracking results, and indicate very satisfactory tracking results even in the presence of highly noisy and occluded data.

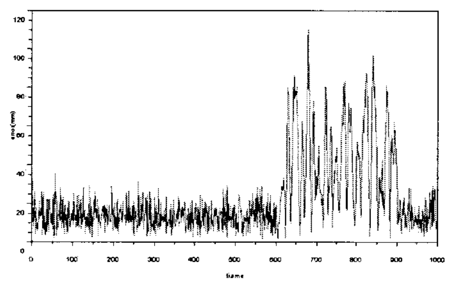

Fig. 5.8. Tracking error in the three marker scenario. Even though the algorithm has barely enough information to infer the position of the occluded marker, it manages to track it successfully.

Finally, we present snapshots of the entire run of the algorithm including the persistent model matching presented in Section 5.3. Figures 5.9-5.12 show the progress of the algorithm from the very beginning until a full match is achieved. Visualization of rigid objects has been disabled to improve clarity of the figures.

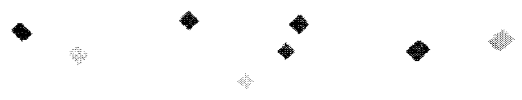

Fig. 5.9. The output of the algorithm at the very beginning of a movement sequence using the marker set presented in Figure 5.5. The subject is viewed from behind and standing with arms stretched to the side. The system has assigned random colors to the markers to help indicate whether they are being tracked consistently from frame to frame.

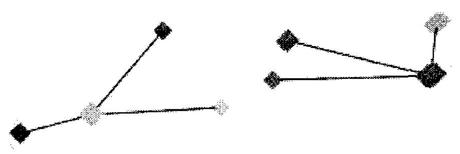

Fig. 5.10. The output of the algorithm after two seconds of arm movement. The algorithm has detected a number of rigid links between moving markers, including a couple that are not expected by the persistent model. For example, a rigid link was detected between the sternum (yellow) and the left elbow.

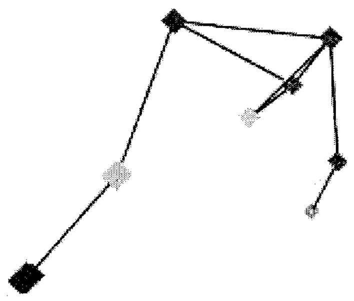

Fig. 5.11. The output of the algorithm after 2.5 seconds of movement. After seeing the additional movement, the algorithm has removed rigid links that proved not to satisfy the rigid link model. We are one link short of matching the full rigid link graph from Figure 5.6.

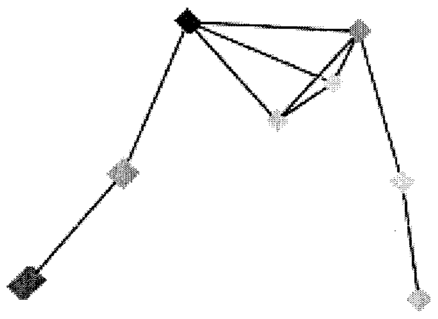

Fig. 5.12. The output of the algorithm after 3 seconds of movement. After the final rigid link has been detected, the first property of the persistent model described in Section 5.3 has been satisfied. The remaining properties are used to label the markers, which is here indicated by a change in color. E.g., elbow markers have been colored turquoise, and wrist markers green. Also, left side markers are displayed slightly darker than the left.

5.5 Conclusions

We have presented a novel dynamic Markov random field framework which unifies the tracking, labeling, and pose estimation steps of motion capture in a optical marker-based system, and demonstrated its ability to reconstruct and track the kinematic structure of a moving body with no a priori model. Our experimental results show that this approach can lead to an accurate model of a human body, joint position estimation, as well as improved marker tracking. This could eliminate requirements for manual model entry, which can potentially make the process of using motion capture systems easier and less tedious.

We have also utilized a straightforward method of matching the automatically constructed model to a pre-labeled, persistent model. We provided a simple example of a persistent model, and a matching algorithm specific to that example. In our future work, we are interested in developing a more general framework which can perform matching using arbitrary properties. We are also interested in the automatic construction of persistent models - i.e., given a few instances of automatically constructed kinematic models all known to correspond to the same persistent model, we would like to infer the elements and the properties of that persistent model automatically.

Another interesting application of this method lies in its extension to markerless tracking. As the MRF-based algorithm is not inherently tied to the 3D observations provided by a marker-based system, it can be augmented to instead work with regular cameras (e.g., with two dimensional point feature locations, or contour-based observations).

Eventually, we hope that this novel method can be used to provide accurate, seamless, and non-intrusive motion capture for human-computer interaction. We are currently working on moving the implementation of the algorithm to a library released under an open source license [40].

6. CONCLUSIONS AND FUTURE WORK

Chapters 2, 3, and 4 presented a cohesive body of work that deals with probabilistic graph-based models for pattern modeling, recognition, and synthesis. The AHMM+EC model presented in Chapter 2 provided a theoretical basis which both unifies a number of individual models typically used in pattern modeling, and maintains a level of simplicity which makes it easy to use when implementing generic algorithms.

This theoretical basis allowed us to apply the concept of semantic states presented through the SNM model in Chapter 3 in a way that is simultaneously applicable to all models that are special cases of the AHMM+EC model. This shows the usefulness of the AHMM+EC in allowing theoretical extensions to be stated once for the general model and be applicable to a number of its special cases.

Using the AHMM+EC as a basis for the implementation of the AME Patterns library showed the practical usefulness of the model, in allowing algorithms to be implemented once for the general model and being immediately applicable to all of its special case submodels.

The future work on probabilistic graph-based models treated in Chapters 2, 3, and 4 will lie mainly in the further development of the AME Patterns library. There are many advances in the fields of pattern modeling, recognition, and synthesis that have occurred in individual disciplines, and implementations of such advances are typically only available in libraries specifically related to the individual disciplines (if at all). Reimplementing such advances into the AME Patterns library will allow them to be useful across fields, and benefit a broader pattern analysis community.

Chapter 5 presented a novel dynamic Markov random field framework which unifies a number of steps of motion capture in an optical marker-based system. This approach allowed us to reconstruct and track the kinematic structure of a moving body with no a priori model, and showed several additional improvements to the performance of existing motion capture system.

Future work in this area lies mostly in adapting the approach to other modalities of motion capture (e.g., video-based techniques), and improving the implementation.

REFERENCES

[1] S.M. Aji and R.J. McEliece. The generalized distributive law. *IEEE Trans. on Information Theory*, 46(2):325–343, March 2000.

[2] Y. Al-Ohali, M. Cheriet, and CY Suen. Introducing termination probabilities to HMM. In *Pattern Recognition, 2002. Proceedings. 16th International Conference on*, volume 3, 2002.

[3] A. Asuncion and D.J. Newman. UCI machine learning repository, 2007.

[4] et al. Benot Jacob, Gal Guennebaud. Eigen: a C++ template library for linear algebra[computer software]. http://eigen.tuxfamily.org/, 2009.

[5] Cambridge University Engineering Department. Hidden Markov Model Toolkit [computer software]. http://htk.eng.cam.ac.uk/, 2009.

[6] S. Chatzis, D. Kosmopoulos, and T. Varvarigou. Robust Sequential Data Modeling Using an Outlier Tolerant Hidden Markov Model. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(9):1657–1669, 2009.

[7] R. Chellappa and A.K. Jain. *Markov Random Fields: Theory and Applications*. Academic Press, 1993.

[8] Andrea Corradini. Dynamic Time Warping for Off-Line Recognition of a Small Gesture Vocabulary. In *Proceedings of the IEEE ICCV RATFG-RTS'01*, page 82, Washington, DC, USA, 2001. IEEE.

[9] Decision Systems Laboratory. Structural Modeling, Inference, and Learning Engine [computer software]. http://genie.sis.pitt.edu/, 2009.

[10] G. Dos Reis and B. Stroustrup. Specifying C++ concepts. *ACM SIGPLAN Notices*, 41(1):295–308, 2006.

[11] Shai Fine, Yoram Singer, and Naftali Tishby. The hierarchical hidden markov model: Analysis and applications. *Machine Learning*, 32(1):41–62, 1998.

[12] GD Forney Jr. The viterbi algorithm. *Proceedings of the IEEE*, 61(3):268–278, 1973.

[13] Jean-Marc Franois. Jahmm: a Java implementation of Hidden Markov Model related algorithms [computer software]. http://www.run.montefiore.ulg.ac.be/ francois/software/jahmm/, 2009.

[14] S. Geman and D. Geman. Stochastic relaxation, gibbs distribution, and the bayesian restoration of images. *IEEE Trans. on Pattern Anal. Machine Intell.*, 6:721–741, 1984.

[15] Alexander Hornung, Sandip Sar-Dessai, and Leif Kobbelt. Self-Calibrating Optical Motion Tracking for Articulated Bodies. *IEEE Virtual Reality Conference*, 2005.

[16] M.W. Kadous. *Temporal classification: Extending the classification paradigm to multivariate time series*. PhD thesis, The University of New South Wales, 2002.

[17] I.A. Karaulova, P.M. Hall, and A.D. Marshall. A Hierarchical Model of Dynamics for Tracking People with a Single Video Camera. In *BMVC*, 2000.

[18] P. Kohli and P.H.S. Torr. Efficiently solving dynamic markov random fields using graph cuts. *Computer Vision, 2005. ICCV 2005. Tenth IEEE International Conference on*, 2:922–929, Oct. 2005.

[19] J. B. Kruskall and M. Liberman. The symmetric time warping algorithm: From continuous to discrete. In *Time Warps, String Edits and Macromolecules*. Addison-Wesley, 1983.

[20] Frank Kschischang, Brendan J. Frey, and Hans-Andrea Loeliger. Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory*, 47, 2001.

[21] M. Kudo, J. Toyama, and M. Shimbo. Multidimensional curve classification using passing-through regions. *Pattern Recognition Letters*, 20(11-13):1103–1111, 1999.

[22] Christopher Lee and Yangsheng Xu. Online, interactive learning of gestures for human/robot interfaces. In *1996 IEEE International Conference on Robotics and Automation*, volume 4, pages 2982–2987, 1996.

[23] K.F. Lee and H.W. Hon. Speaker-independent phone recognition using hidden Markov models. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 37(11):1641–1648, 1989.

[24] Stan Z. Li. *Markov random field modeling in image analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2001.

[25] W. J. Masek and M. S. Paterson. A faster algorithm for computing string edit distances. *Journal of Computer and System Sciences*, 20:18–31, 1980.

[26] Max Planck Institute for Molecular Genetics. General Hidden Markov Model library [computer software]. http://ghmm.sourceforge.net/, 2009.

[27] Thomas B. Moeslund and Erik Granum. A survey of computer vision-based human motion capture. *Comput. Vis. Image Underst.*, 81(3):231–268, 2001.

[28] TK Moon. The expectation-maximization algorithm. *Signal Processing Magazine, IEEE*, 13(6):47–60, 1996.

[29] Kevin Murphy. Hidden Markov Model Toolbox [computer software]. http://people.cs.ubc.ca/ murphyk/Software/HMM/hmm.html, 2005.

[30] Kevin Murphy. Bayes Net Toolbox [computer software]. http://people.cs.ubc.ca/ murphyk/Software/BNT/bnt.html, 2007.

[31] Kevin Patrick Murphy. *Dynamic Bayesian Networks: Representation, Inference and Learning*. PhD thesis, UC Berkeley, Computer Science Division, 2001.

[32] K.P. Murphy. *Dynamic Bayesian Networks: Representation, Inference and Learning*. PhD thesis, UNIVERSITY OF CALIFORNIA, 2002.

[33] P. Natarajan and R. Nevatia. Hierarchical multi-channel hidden semi Markov models. In *Proceedings of the International Joint Conference on Artificial Intelligence, MM Veloso, Ed*, pages 2562–2567, 2007.

[34] TCB Networks. StrokeIt .9.6 [computer software]. http://www.tcbmi.com/strokeit/, 2009.

[35] Opera Software. Opera 8.5 [computer software]. http://www.opera.com/, 2006.

[36] Bo Peng, Gang Qian, and Stjepan Rajko. A View-Invariant Video Based Full Body Gesture Recognition System. *ICPR*, 2008.

[37] LR Rabiner. A tutorial on hidden Markov models and selected applications inspeech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[38] S. Rajko, G. Qian, T. Ingalls, and J. James. Real-time Gesture Recognition with Minimal Training Requirements and On-line Learning. *IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8, 2007.

[39] Stjepan Rajko. AME Patterns Library [computer software]. http://ame4.hc.asu.edu/amelia/patterns/, 2008.

[40] Stjepan Rajko. AMELiA - the Arts, Media and Engineering Library Assortment [computer software]. http://ame4.hc.asu.edu/amelia/, 2008.

[41] Stjepan Rajko and Gang Qian. A Hybrid HMM/DPA Adaptive Gesture Recognition Method. In *ISVC*, pages 227–234, 2005.

[42] Stjepan Rajko and Gang Qian. Autonomous real-time model building for optical motion capture. In *IEEE ICIP*, pages 1284–1287, 2005.

[43] Stjepan Rajko and Gang Qian. HMM Parameter Reduction for Practical Gesture Recognition. *IEEE International Conference on Automatic Face and Gesture Recognition*, 2008.

[44] Stjepan Rajko, Gang Qian, Bo Peng, and Todd Ingalls. The augmented hidden Markov model: a unifying framework. *submitted to IEEE Transactions on Pattern Analysis and Machine Intelligence (under review)*, 2009.

[45] Maurice Ringer and Joan Lasenby. A procedure for automatically estimating model parameters in optical motion capture. *BMVC*, 2002.

[46] Sensiva. Symbol commander [computer software]. http://www.sensiva.com/products/index.html, 2005.

[47] Khurram Shafique and Mubarak Shah. A Non-Iterative Greedy Algorithm for Multi-frame Point Correspondence. In *Int. Conf. Computer Vision*, pages 110–115, Nice, France, 2003.

[48] I. Shahin. Enhancing speaker identification performance under the shouted talking condition using second-order circular hidden Markov models. *Speech Communication*, 48(8):1047–1055, 2006.

[49] S.M. Siddiqi, G.J. Gordon, and A.W. Moore. Fast state discovery for hmm model selection and learning. In *Proc. of the Int. Conf. on Artificial Intelligence and Statistics*, 2007.

[50] Marius-Calin Silaghi, Ralf Plankers, Ronan Boulic, Pascal Fua, and Daniel Thalmann. Local and Global Skeleton Fitting Techniques for Optical Motion Capture. *IFIP CapTech*, 1998.

[51] Bjarne Stroustrup. A c++ libraries wish list [keynote speech]. http://www.boostcon.com/news/2008/Feb/01/bjarne-stroustrup-to-deliver-2008-keynote, 2008.

[52] Lionhead Studios. Black & white 2 [computer software]. http://www.lionhead.com/bw2/, 2005.

[53] P. Taylor. Unifying unit selection and hidden Markov model speech synthesis. In *Ninth International Conference on Spoken Language Processing*. ISCA, 2006.

[54] various authors. Boost C++ Libraries [computer software]. http://www.boost.org/, 2009.

[55] Cor J. Veenman, Marcel J.T. Reinders, and Eric Backer. Resolving Motion Correspondence for Densely Moving Points. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2001.

[56] Christian Vogler and Dimitris Metaxas. ASL recognition based on a coupling between HMMs and 3d motion analysis. In *ICCV '98: Proceedings of the Sixth International Conference on Computer Vision*, page 363, Washington, DC, USA, 1998.

[57] Christian Vogler and Dimitris Metaxas. Handshapes and Movements: Multiple-Channel American Sign Language Recognition. In *Lecture Notes in Computer Science*, volume 2915, pages 247–258, Jan 2004.

[58] Ying Wu, Gang Hua, and Ting Yu. Tracking articulated body by dynamic markov network. *Computer Vision, 2003. Proceedings. Ninth IEEE International Conference on*, 2:1094–1101, 2003.

[59] Jingyu Yan and Marc Pollefeys. Automatic Kinematic Chain Building from Feature Trajectories of Articulated Objects. In *IEEE CVPR*, 2006.

[60] Z. Lieberman et al. openFrameworks [computer software]. http://www.openframeworks.cc/, 2008.

[61] Jure Zakotnik, Tom Matheson, and Volker Durr. A posture optimization algorithm for model-based motion capture of movement sequences. *Journal of Neuroscience Methods*, 2004.