Generating Learning Algorithms: Hidden Markov Models as a Case Study

GENERATING LEARNING ALGORITHMS: HIDDEN MARKOV MODELS AS A CASE STUDY

BY

DANIEL M. SZYMCZAK, B.Eng.

A THESIS

SUBMITTED TO THE DEPARTMENT OF SOFTWARE ENGINEERING

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

Master of Applied Science (2014) McMaster University
(Software Engineering) Hamilton, Ontario, Canada

TITLE:	Generating Learning Algorithms: Hidden Markov Models as a Case Study
AUTHOR:	Daniel M. Szymczak B.Eng., (Software Engineering) McMaster University, Hamilton, Ontario, Canada
SUPERVISOR:	Dr. Jacques Carette
NUMBER OF PAGES:	xi, 99

In loving memory of my grandmother

Abstract

This thesis presents the design and implementation of a source code generator for dealing with Bayesian statistics. The specific focus of this case study is to produce usable source code for handling Hidden Markov Models (HMMs) from a Domain Specific Language (DSL).

Domain specific languages are used to allow domain experts to design their source code from the perspective of the problem domain. The goal of designing in such a way is to increase the development productivity without requiring extensive programming knowledge.

Acknowledgements

I would like to thank my supervisor Dr. Jacques Carette and my family for all of their support, teaching, and guidance.

I would also like to thank the people at xkcd for allowing me to use their comic.

Notation and abbreviations

- AST = Abstract Syntax Tree
- DP = Dynamic Programming
- DSL = Domain Specific Language
- HMM = Hidden Markov Model
- HTML = HyperText Markup Language

Contents

Abstract **iv**

Acknowledgements **v**

Notation and abbreviations **vi**

1 Introduction **1**

2 Requirements **4**

2.1 Code generator requirements 4

2.2 DSL Requirements 5

2.3 Generated source requirements 5

3 Bayesian Statistics **6**

3.1 An Introduction 6

3.2 Bayesian Inference 8

3.3 Introduction to Hidden Markov Models (HMMs) 10

3.4 The Math behind Hidden Markov Models 11

4 Domain Specific Languages **18**

4.1 What are DSLs? 18

4.2 DSLs: Embedded vs. External 19

4.3 Relevance 20

5 Code Generation **22**

5.1 What is code generation? 22

5.2 How does code generation work? 23

5.3 Why use code generation? 23

6 The HMM Generator **25**

6.1 The Components 26

6.2 Some domain specific terms 31

6.3 Constructing a program using the DSL Syntax 34

6.3.1 Expressions 36

6.3.2 Statements 39

6.4 Putting it all together: A simple Hello World program 42

7 HMM Implementation **46**

7.1 The forward algorithm 48

7.2 The backward algorithm 51

7.3 Finding the optimal path using $\gamma_t(i)$ 53

7.4 Re-estimating $A, B,$ and π 55

7.5 Scaling HMMs 57

7.6 HMMs in the HMM Generator 60

8 Testing **66**

8.1 Testing the generator 66

8.2 Testing the generated code 67

8.2.1 Does it work? 67

8.2.2 Does it represent the design? 71

8.3 Known Issues and Special Cases 72

9 Conclusion 74

9.1 Lessons Learned 74

9.2 Next Steps 75

A AST Implementations 77

A.1 ASTDesign 77

A.2 ASTInternal 78

A.3 ASTC 83

B Generated C Code 86

B.1 Header File 86

B.2 C Source File 87

C Large Test Case Results 98

List of Figures

3.1 A look at the difference between frequentist and Bayesian statisticians. 8

6.1 Translating an “Alloc” statement from the internal AST to the C AST 28

6.2 An example function from the printer 29

6.3 The generate function in the design file 29

6.4 Setting up the generator with specific design choices 30

6.5 An example of some helper functions for the printer 30

6.6 A “hello world” implementation in the HMM Generator 43

6.7 Functions for translating declarations and methods to the C AST . . 44

6.8 The “printC” and “printH” functions in the printer 44

6.9 The functions for building the prototypes and methods in C 44

6.10 Printing the “Out” statement 45

6.11 The “HelloWorld” header file 45

6.12 The “HelloWorld” c file 45

7.1 HMM Representation in Haskell 46

7.2 HMM Representation in C 47

7.3 HMM Representation in the HMM Generator 47

7.4 $\alpha_t(i)$ computation in Haskell 49

7.5 $\alpha_t(i)$ computation in C 49

7.6 α-pass in Haskell 50
7.7 α-pass in C 50
7.8 $\beta_t(i)$ computation in Haskell 52
7.9 $\beta_t(i)$ computation in C 52
7.10 $\gamma_t(i)$ computation in Haskell 53
7.11 $\gamma_t(i)$ computation in C 54
7.12 Finding the optimal path in Haskell 54
7.13 Finding the optimal path in C 54
7.14 Training a model in Haskell 56
7.15 Training a model in C 56
7.16 Scaled training code in C 59
7.17 α computation in the HMM Generator 61
7.18 α-pass in the HMM Generator 61
7.19 β computation in the HMM Generator 62
7.20 β-pass in the HMM Generator 63
7.21 A non-scaled γ computation in the HMM Generator 63
7.22 Scaled γ computation in the HMM Generator 64
7.23 Training a model in the HMM Generator 65

Chapter 1

Introduction

There are many software applications that would benefit from the use of learning algorithms, including but not limited to, security, object recognition, search engines, medical diagnostics, fraud detection, and information retrieval. However, creating appropriate models and algorithms for each specific application can be time consuming and costly.

One proposed method for simplifying the production of learning algorithms, while simultaneously allowing them to be customized to their applications, is to create a domain specific language (DSL) and a source code generator. From there, changes in algorithm design decisions can be implemented with trivial effort and working source code will be available promptly.

There are many approaches to learning algorithms including supervised learning, unsupervised learning, semi-supervised learning, transduction (or transduction inference), or reinforcement learning. In fact learning can be broken down into a series of different approaches based on current machine learning algorithms. These approaches include, but are not limited to:

- Artificial neural networks
- Bayesian networks
- Decision tree learning
- Inductive logic programming
- Representation learning
- Sparse dictionary learning

This paper will be looking at Bayesian learning algorithms, specifically focusing on hidden Markov models (HMMs). HMMs are a simple type of dynamic Bayesian network and are very well known for their use in temporal pattern recognition, which has applications in many fields including bioinformatics, cryptanalysis, and speech, handwriting, and gesture recognition.

The overarching goal of this work is to create a DSL and source code generator for handling HMMs in such a way that it will be fairly simple to learn and use. However, as the overall development of the generator followed an approach similar to the agile software development model, this thesis covers the work done over a relatively short period of time and has yet to achieve that final goal. The current version of this work acts as a proof of concept that could be expanded upon into a more fully-featured DSL. The actual production of the generator involved creating an initial version with a barebones DSL and improving upon it in successive iterations in order to refine the language and implement additional features (as per the requirements).

Please Note: The current version of the generator source code is available on the "Projects" page on `http://www.dszymczak.com`

Organization of the thesis

Chapter 2 will outline the requirements for the HMM Generator and for the final generated source code.

Chapter 3 will introduce the reader to Bayesian statistics. This introduction will be followed by a detailed explanation of what HMMs are and how they work. The main types of problems HMMs are used to solve and the math behind the solutions will be discussed at length.

To increase understanding as to why a generative approach was taken, Chapter 4 and Chapter 5 will introduce domain specific languages (DSLs) and code generators. These chapters will focus on what DSLs and code generators are, their advantages and disadvantages, and how they are relevant to the problem at hand.

Chapter 6 will explain the infrastructure of the HMM Generator. There will be an introduction to the various components contained therein, followed by examples of domain-specific terms, and finally an example program which shows how the HMM Generator works.

In Chapter 7 the various HMM implementations across languages will be compared and contrasted. Some design decisions will be introduced, and the HMM implementation file written in the DSL will be broken down and compared with the other language implementations.

Chapter 8 will cover the various stages of testing for the HMM Generator. This will test to ensure that the generator produces output code, that the code compiles and runs, and that the results obtained from running the program are accurate.

Chapter 2

Requirements

In a very general sense there is one main requirement for this project, which is to create a code generator that will produce usable code for handling HMMs.

However, that requirement can be broken down into a series of others. These requirements cover the code generator, the domain specific language, and the generated source code itself.

2.1 Code generator requirements

First off, the code generator should use a domain specific language for defining the desired output source code. The generator should also be able to interpret design choices at a high level and modify the desired source code based on those choices (for example, changing one parameter to create a different implementation for handling the same types of problems). The generator must be able to translate the implementation designed in the DSL into a representation of the given output language internally before outputting any source code. Finally, the generator must be able to actually output the source code into a usable file format.

2.2 DSL Requirements

The end goal of the DSL design is to be able to represent certain aspects of the problem domain in a simple manner such that domain experts will find the terminology familiar. For example, the DSL must include a way to initialize matrices with probability values (as that is a major component in dealing with HMMs) as well as performing computations on matrices. It must also be able to handle summation over a specified range (and other similar mathematical operations) in a straightforward manner. The DSL must also include options for design choices (such as scaling, which will be explained later) that the generator can use to modify the final version of the ouput source code.

2.3 Generated source requirements

The final generated source code has to meet several requirements. First, it must be able to compile without any person performing modifications to it. It must also contain the appropriate algorithms for handling HMMs and they must be able to be run. Finally, the code itself must also be correct in that the results obtained from running the generated code on a hidden Markov model are accurate.

Chapter 3

Bayesian Statistics

Note: Much of the information covered in the following two sections is based on Bolstad (2007).

3.1 An Introduction

Bayesian statistics is named for Thomas Bayes who proved a special case of what is now known as "Bayes' theorem". The general version of the theorem was actually introduced by Pierre-Simon Laplace who came after Bayes. The most common form of Bayes' theorem states the following:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{3.1}$$

What that means depends on the interpretation of probability, as one can look at the Bayesian or frequentist interpretations.

Frequentist Vs. Bayesian Statistics

The simplest explanation of the difference between frequentist and Bayesian statistics is that frequentist statistics looks at $\mathbf{P(D|H)}$ whereas Bayesian statistics look at $\mathbf{P(H|D)}$ where $\mathbf{H}$ is a hypothesis and $\mathbf{D}$ is data. They have different interpretations of what "probability" is.

This means frequentist statistics look at the proportion of outcomes. Essentially the data may come out differently if the experiment is repeated and the hypothesis is either true or false (but which of the two is unknown). The name "frequentist" comes from the expected frequency of observing the data given some hypothesis. A frequentist would essentially think of probability as the frequency with which something would happen, in a lengthy series of trials.

Bayesian statistics, on the other hand, look at probability as the degree of belief in the hypothesis. It takes into account prior data to update the degree of belief in a given hypothesis and as new data emerges the probability can be consistently updated.

An example of the difference (portrayed humorously) can be found in Figure 3.1. In the example we see the frequentist approach of looking at the data: they determine the probability of randomly rolling two dice and having both come up six, then presume the hypothesis correct based on the p-value being less than the significance level. The Bayesian approach while not explicitly stated in the example would start off the same way (by computing the probability of the dice roll coming up with two sixes), but then the degree of belief is updated based on prior knowledge (from the time the sun formed until today it has not gone nova, there is no evidence that the sun was approaching nova) and the Bayesian statistician's degree of belief in the

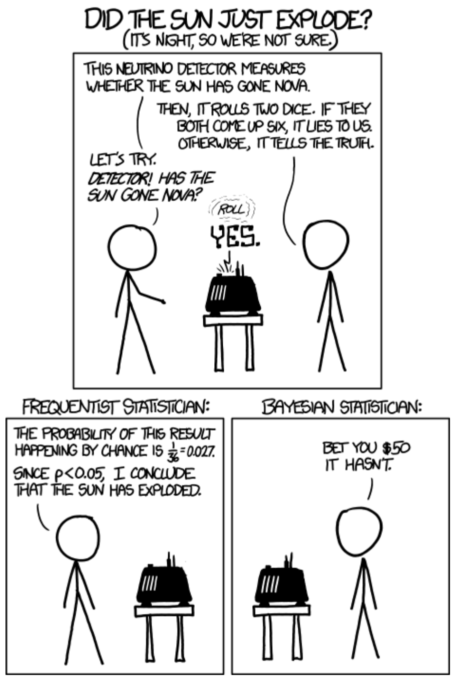
DID THE SUN JUST EXPLODE?
(IT'S NIGHT, SO WE'RE NOT SURE.)
THIS NEUTRINO DETECTOR MEASURES WHETHER THE SUN HAS GONE NOVA.
THEN, IT ROLLS TWO DICE. IF THEY BOTH COME UP SIX, IT LIES TO US. OTHERWISE, IT TELLS THE TRUTH.
LET'S TRY. DETECTOR! HAS THE SUN GONE NOVA?
(ROLL)
YES.
FREQUENTIST STATISTICIAN:
THE PROBABILITY OF THIS RESULT HAPPENING BY CHANCE IS 1/36 = 0.027. SINCE p<0.05, I CONCLUDE THAT THE SUN HAS EXPLODED.
BAYESIAN STATISTICIAN:
BET YOU $50 IT HASN'T.

Figure 3.1: A look at the difference between frequentist and Bayesian statisticians.

hypothesis becomes negligible.

3.2 Bayesian Inference

Bayesian inference is a method of inference using Bayes theorem (Equation 3.1). If we were to call "A" our *hypothesis*, and "B" our *evidence*, then the *prior probability* $P(A)$ is the probability of the hypothesis before observing any evidence. The *posterior probability* $P(A|B)$ is the probability of the hypothesis given the observed evidence. $P(B|A)$ is known as the *likelihood* of observing the evidence given the hypothesis. The *marginal likelihood* $P(B)$ is the likelihood of the evidence without taking into account any other variables. The marginal likelihood remains constant across all the hypotheses that are being considered. Note that for different hypotheses, the only factors that affect the posterior probability are the prior probability and the likelihood since the marginal likelihood does not change across different hypotheses.

Essentially that means that the posterior probability is determined by the initial likeliness of a hypothesis and how compatible that hypothesis is with the observed evidence. This makes sense rationally as it means that a hypothesis should be rejected if the evidence does not match up with it, or if the hypothesis was extremely unlikely in the first place. To clarify, consider the following: A statistician decides to purchase something from an ice cream truck nearby, I have the following four hypotheses

1. H_1: The statistician purchases an ice cream cone
2. H_2: The statistician purchases a popsicle
3. H_3: The statistician purchases a turkey sandwich
4. H_4: The statistician purchases nothing

Now consider the following two scenarios:

1. The statistician enters the room holding an ice cream cone. This evidence supports H_1 and opposes the other three.
2. The statistician enters the room holding a turkey sandwich. While this evidence would seem to support H_3, the prior belief in H_3 (that one can buy a turkey sandwich from an ice cream truck) is extremely small and so the posterior probability remains small.

3.3 Introduction to Hidden Markov Models (HMMs)

In order to understand what a hidden Markov model is, one must first understand what constitutes a Markov model. A Markov model is a stochastic model (it models a process where the state is dependent on previous states in a non-deterministic way) that assumes the Markov property (Baum and Petrie, 1966).

A stochastic process with the Markov property is one wherein the *conditional probability distribution* of future states depends not on the sequence of events leading up to the present state, but on the present state alone.

A hidden Markov model gets its name from the fact that it is a model assumed to be a Markov process with some hidden (unobserved) states. Note that the term “hidden” refers to the actual sequence of states that the model passes through and not on any of the actual parameters of the model. On the other hand, simpler Markov models have states which are directly visible to the observer. The difference then is that an HMM looks at the output (which is state dependent) to infer the sequence of states. This is done through analysis of the probability distributions of each state over the possible outputs.

HMMs can be used for a wide array of applications, however they boil down to one or more of three fundamental problems (Stamp, 2004). The three problems are as follows:

1. Determine the likelihood of an observed sequence of observations given a model.

2. Given a model and an observation sequence, find the most likely state sequence for the underlying model.

3. Find the model that maximizes the probability of a given observation sequence

using that sequence, the number of states in the model, and the number of possible observations in the model. This is commonly called training a model to best fit the observed data.

These problems (and their solutions) will be discussed in more detail in Section 3.4.

Hidden Markov models have many current real-world applications including (but not limited to) cryptanalysis, machine translation, gene prediction, time series analysis, and speech recognition. Speech recognition was actually one of the earliest applications of HMMs (Baker, 1975). The problem of speech recognition can be solved fairly easily using the solution to problem 3 above. By training different models to recognize different words (or phonemes), a segment of speech can be analysed over all the models and the individual words (or phonemes) can be identified. Many speech recognition softwares allow users to train the models based on their own speech patterns. This is a great advantage as it enables the software to account for particular accents or speech mannerisms that may be uncommon in the general population.

3.4 The Math behind Hidden Markov Models

This section will be based heavily on Stamp (2004).

In order to show the mathematics behind HMMs, some terms must first be defined. For a model $\lambda = (A, B, \pi)$ let:

- N = Number of states in the model
- M = Number of observation symbols
- $Q = (q_0, q_1, ..., q_{N-1})$ = Distinct states of the Markov process

- V = Set of possible observations
- π = Initial State Distribution $(1 \times N)$ row stochastic matrix
- $A = \{a_{ij}\}$ = State Transition Probabilities $(N \times N$ row stochastic matrix)
- $B = \{b_j(k)\}$ = Observation Probability Matrix $(N \times M$ row stochastic matrix)
- T = Length of the observation sequence
- $\mathcal{O} = (\mathcal{O}_0, \mathcal{O}_1, ..., \mathcal{O}_{T-1})$ = An observation sequence

Where

$$a_{ij} = P(\text{state } q_j \text{ at } t+1| \text{ state } q_i \text{ at } t)$$

$$b_j(k) = P(\text{observation } k \text{ at } t|\text{state } q_j \text{ at } t)$$

Put more simply, a_{ij} is the probability of transitioning to state q_j from state q_i directly and $b_j(k)$ is the probability of observing k while in state q_j. With that in mind, the three fundamental problems (from Section 3.3) can be solved as follows.

Determine the likelihood of a sequence of observations

Solving this problem requires finding the likelihood of a sequence of observations ($P(\mathcal{O}|\lambda)$). Let $S = (s_0, s_1, ..., s_{T-1})$ be a sequence of states. Now, from the definition of B we have

$$P(\mathcal{O}|S, \lambda) = b_{s_0}(\mathcal{O}_0) b_{s_1}(\mathcal{O}_1) \cdots b_{s_{T-1}}(\mathcal{O}_{T-1}).$$

Also, from the definitions of π and A we see that

$$P(S|\lambda) = \pi_{s_0} \times a_{s_0,s_1} \times a_{s_1,s_2} \times \cdots \times a_{s_{T-2},s_{T-1}}.$$

Now since

$$P(\mathcal{O}, S|\lambda) = \frac{P(\mathcal{O} \cap S \cap \lambda)}{P(\lambda)}$$

and

$$P(\mathcal{O}|S,\lambda)P(S|\lambda) = \frac{P(\mathcal{O} \cap S \cap \lambda)}{P(S \cap \lambda)} \cdot \frac{P(S \cap \lambda)}{P(\lambda)} = \frac{P(\mathcal{O} \cap S \cap \lambda)}{P(\lambda)},$$

therefore

$$P(\mathcal{O}, S|\lambda) = P(\mathcal{O}|S,\lambda)P(S|\lambda).$$

To determine $P(\mathcal{O}|\lambda)$ all possible state sequences must be accounted for, thus the solution is to sum over all possible S like so:

$$\begin{aligned} P(\mathcal{O}|\lambda) &= \sum_S P(\mathcal{O}, S|\lambda) \\ &= \sum_S P(\mathcal{O}|S,\lambda)P(S|\lambda) \\ &= \sum_S \pi_{s_0} b_{s_0}(\mathcal{O}_0) a_{s_0,s_1} b_{s_1} \cdots a_{s_{T-2},s_{T-1}} b_{s_{T-1}}(\mathcal{O}_{T-1}) \end{aligned}$$

Generally computing this directly is infeasible (requiring $\sim\ 2TN^T$ multiplications) (Stamp, 2004). However, there exists an efficient algorithm for computing $P(\mathcal{O}|\lambda)$ which is known as the *α-pass* or *forward algorithm.*

The algorithm requires a definition of

$$\alpha_t(i) = P(\mathcal{O}_0, \mathcal{O}_1, \cdots, \mathcal{O}_t, s_t = q_i|\lambda) \tag{3.2}$$

for $t = 0, 1, \ldots, T-1$ and $i = 0, 1, \ldots, N-1$. The term $\alpha_t(i)$ is known as the probability of a partial observation sequence up to time t with the process in state q_i at time t.

The forward algorithm is as follows:

1. Let $\alpha_0(i) = \pi_i b_i(\mathcal{O}_0)$ for $i = 0, 1, \ldots, N-1$
2. For $i = 0, 1, \ldots, N-1; t = 1, 2, \ldots, T-1$ compute:

$$\alpha_t(i) = \left[\sum_{j=0}^{N-1} \alpha_{t-1}(j) a_{ji}\right] b_i(\mathcal{O}_t)$$

3. Then (from Equation 3.2)

$$P(\mathcal{O}|\lambda) = \sum_{i=0}^{N-1} \alpha_{T-1}(i) \tag{3.3}$$

This will be covered again in Chapter 7.

Find the most likely state sequence

The most likely state sequence can be defined by taking either the HMM approach or the dynamic programming approach. The HMM approach wants to maximize the expected number of correct states, whereas the dynamic programming approach wants to find the highest overall scoring path. These solutions do not necessarily have to be the same. For the purposes of this thesis, the HMM approach will be used.

The HMM approach begins with the definition of the *β-pass* (or *backward algorithm*).

The algorithm requires a definition of

$$\beta_t(i) = P(\mathcal{O}_{t+1}, \mathcal{O}_{t+2}, \cdots, \mathcal{O}_{T-1} | s_t = q_i, \lambda) \tag{3.4}$$

for $t = 0, 1, \ldots, T-1$ and $i = 0, 1, \ldots, N-1$. The backward algorithm is similar to the forward algorithm, except that it measures the partial probability after time t. Computing the $\beta_t(i)$ will be covered in Chapter 7.

One more definition is required, and that is:

$$\gamma_t(i) = P(s_t = q_i | \mathcal{O}, \lambda) \tag{3.5}$$

for $t = 0, 1, \ldots, T-2$ and $i = 0, 1, \ldots, N-1$. Now, since it is known that $\alpha_t(i)$ determines the relevant probability up to time t and $\beta_t(i)$ measures the relevant probability after time t, the most likely state at time t is the state q_i which maximizes

$$\gamma_t(i) = \frac{\alpha_t(i)\beta_t(i)}{P(\mathcal{O}|\lambda)} \tag{3.6}$$

over the index i.

Training the model

To train the model, the dimensions N and M are known (and fixed), but the elements of A, B, and π need to be determined (and they must remain row-stochastic matrices).

The first necessary step is to define "di-gammas"

$$\gamma_t(i,j) = P(s_t = q_i, s_{t+1} = q_j | \mathcal{O}, \lambda) \tag{3.7}$$

for $t = 0, 1, \ldots, T-2$ and $i, j \in \{0, 1, \ldots, N-1\}$. This $\gamma_t(i,j)$ is the probability of transitioning to state q_j at time $t+1$ from state q_i at time t. Di-gammas are related to gamma ($\gamma_t(i)$ is related to $\gamma_t(i,j)$) by:

$$\gamma_t(i) = \sum_{j=0}^{N-1} \gamma_t(i,j).$$

Another way to write the di-gammas (in terms of α, β, A and B) is

$$\gamma_t(i,j) = \frac{\alpha_t(i) a_{ij} b_j(\mathcal{O}_{t+1}) \beta_{t+1}(j)}{P(\mathcal{O}|\lambda)}. \tag{3.8}$$

Finally it is time to re-estimate the value of π, A, and B which is done as follows. For $i = 0, 1, \ldots, N-1$

$$\pi_i = \gamma_0(i). \tag{3.9}$$

For $i = 0, 1, \ldots, N-1;\ j = 0, 1, \ldots, N-1$

$$a_{ij} = \sum_{t=0}^{T-2} \gamma_t(i,j) \Bigg/ \sum_{t=0}^{T-2} \gamma_t(i). \tag{3.10}$$

For $j = 0, 1, \ldots, N-1;\ k = 0, 1, \ldots, M-1$

$$b_j(k) = \sum_{\substack{t \in \{0,1,\ldots,T-2\} \\ \mathcal{O}_t = k}} \gamma_t(j) \Bigg/ \sum_{t=0}^{T-2} \gamma_t(j). \tag{3.11}$$

This is done in many iterations, allowing the model to be trained until $P(\mathcal{O}|\lambda)$ ceases to increase by some predetermined threshhold or the maximum number of iterations is met.

The full algorithm for computing and training an HMM will be described in detail in Chapter 7.

Chapter 4

Domain Specific Languages

The following is an introduction to domain specific languages to provide a baseline for some of the more important concepts behind DSLs. This chapter is based in part on similar work with DSLs from Beyak (2011), Costabile (2012), and Curutan (2013).

4.1 What are DSLs?

Domain specific languages are exactly what their name suggests. They are languages that have been developed to tackle a specific set of problems (known as the problem domain). Domain specific languages are typically very limited in their scope, but are able to describe their respective problem domains in a concise and efficient manner. One example of a very widely known DSL is HTML (commonly used in website design). DSLs are generally not Turing complete, and therefore are not typically used in the same manner as general-purpose languages, however, there are some examples of Turing complete DSLs (such as PostScript).

One of the main challenges for designing a DSL comes in understanding the problem domain. If the problem domain is not sufficiently well understood there will be ambiguity when it comes to translating a DSL program into its desired representation. A well understood domain will have generally standardized terminology, thus allowing those who work in the problem domain to express their ideas in a consistent manner. That consistency allows users of a DSL to work with familiar terminology in defining their desired output.

Consistency and ease of expression are one of the primary goals of DSLs. By simplifying how one would express common tasks in the problem domain with respect to conventional programming languages, it allows for experts in that domain to more effectively implement solutions to complex problems.

DSLs can be implemented through interpretation or code generation methods. Interpretation is when some line(s) of program text (written in the DSL) are parsed at runtime and a result is immediately produced. On the other hand, code generation allows for the program text to be parsed and produces an intermediary output, which can then be processed seperately to provide the desired behaviour (Fowler, 2010). Code generation will be discussed in more detail in Chapter 5.

4.2 DSLs: Embedded vs. External

Embedded (also called "internal") and external (also called "standalone" or "free-standing") DSLs are distinct based on their design and implementation. Each of these DSLs comes with its own specific advantages and disadvantages.

Embedded DSLs get their name from the fact that the DSL is embedded into a general purpose host language (typically a high-level language). The DSL is created through adding specialised constructs to the host language and essentially creating a somewhat distinct language within.

One of the major benefits of an embedded DSL is that it requires relatively little effort to create since the host language's compiler handles the processing. Another benefit is that any useful functionality of the host language can be utilised in the design and implementation of the DSL. As for disadvantages, the DSL is limited by the host language's syntax and grammar rules. This can limit the overall expressiveness of the language. Another potential disadvantage is that those who will be working with the DSL will need a higher level of familiarity with the host language to use it effectively.

External DSLs are not embedded in a host language. This gives them the ability to have incredibly customizable syntax and grammar rules, leading to a much richer and more expressive language for the problem domain. The language can also be made much more intuitive to those familiar with the problem domain. The main disadvantage of an external DSL is the lack of an existing compiler. Design and implementation of a compiler can be an incredibly complex task depending on the scope of the language.

4.3 Relevance

From Chapter 3 we can see that the problem domain (Bayesian Statistics) is a very rich and complex domain. For this project it was decided that a subsection of that domain would be taken, specifically one focusing on Hidden Markov Models (as seen in Sections 3.3 and 3.4).

The chosen problem domain is specialized enough that a DSL would allow for domain experts to create optimized code for dealing with Hidden Markov Models. There are many design decisions that could affect the final output code in terms of efficiency and scalability (these were touched on in Section 3.4 and will come up again in Chapter 7), thus a DSL allows the users to accomodate for their needs with a trivial amount of time and effort.

For the purposes of the HMM Generator, an embedded DSL was created using Haskell as the host language in order to take advantage its pattern matching capabilities.

Chapter 5

Code Generation

This chapter will expand upon code generation as mentioned in Chapter 4. Specifically it will be referring to source code generation.

5.1 What is code generation?

In the simplest terms, code generation is an automated method of creating source code using a higher-level language (Mur, 2006). Compilers are a good example of code generators as they take higher level programming languages and convert them into machine code. However, source code generators are a little different in that they take a high level abstract language (a DSL for example) or model and generate a new source file from it. For example the HMM Generator outputs source code in the C language.

5.2 How does code generation work?

Code generation goes through a series of stages. Once the high level source program is written, the following steps are performed when the generator is run:

1. Compile the framework and parser
2. Parse the high-level source
3. Process the parsed code
    - Part of this involves running optimization algorithms (if implemented)
4. Translate the processed code into the output language
5. Output the new source in the output language

The final output code must then be compiled separately before being used. Also note that processing of the initial source code can be done in a single pass (in some cases), however, it is usually done in multiple passes to run any optimization algorithm(s) in succession.

5.3 Why use code generation?

With code generation there is no restriction on what language the output file must be in. In the case of embedded DSLs the host language can be used to write the parser and once the program code is written in the DSL, the host language can be forgotten about altogether and the generated source can be in any language. This is especially useful when dealing with languages that do not have the tools to support DSLs or systems that only support those languages. If a system were to only support compiled C, the embedded DSL could be written in any host language as long as the generator was set up to translate the DSL into C code. This allows for flexibility in design and implementation without needing to make any compromises as to the input language.

Code generators also allow for decisions to be made in the high level source that can change the way the output code is generated. This allows optimization to take place in the generation phase and results in more efficient output code. That code will then go through its own compiler optimizations at compile time, which can lead to even more efficient code.

The extra compilation step (Step 1 of Section 5.2) along with the need for compilation of the final generated code can be seen as a disadvantage as it increases the complexity of the finished product, which may lead to longer testing and debugging periods. However, for the scope of the HMM Generator the advantages outweigh the disadvantages.

Chapter 6

The HMM Generator

The generator itself is comprised of a series of components that work in sequence to produce the final output code. These components include:

1. A series of Abstract Syntax Trees (ASTs).
2. A translator for moving from one AST to another.
3. A printer which outputs the final output code.
4. An assortment of helper functions.
5. An implementation file.
6. A design file.

Each of the components will be described in detail in the following sections.

6.1 The Components

The ASTs

Note: for the full ASTs, please see Appendix A.

There are three main abstract syntax trees for the HMM Generator. They are the design (or choice) AST, the internal language AST, and a trimmed-down C language AST.

The design AST (Appendix A.1) is solely made up of possible design choices. The data type "Choices" shows which are available to the user currently. The current version of the generator can support choice of language ("Lang"), whether or not to use scaling ("Scale"), and how many iterations should be performed (at max) while training a model ("MaxIt"). Also note that the "Library" type is not used within the AST itself, but it is used elsewhere in the source files.

The internal AST (Appendix A.2) is the "language" that the HMM Generator implementation files are written in. The implementation files themselves will be explained later on. There are six key components of the AST:

1. The Program Declaration
2. Global Declarations
3. Function Signatures
4. Parameter Declarations
5. Statements
6. Expressions

The program section of the AST is used to declare what the output program will look like. A program must include design choices, a name, and a list of global declarations. Global declarations include any class, macro, or function declarations that will be necessary in the final code. The function signatures are analogous to function prototypes in C; they detail the name, type, and parameters of each function. Parameter declarations are fairly self-explanatory as each parameter only needs a name and a type. Statements control the flow and computations in the output code. They include (but are not limited to) assignments, conditionals, and function calls. There are certain statements that are specific to the problem domain (for example *MatrixBlockInit*), and the comments underneath them explain what they do in more detail. Statements contain expressions that describe what is to be computed. The internal AST also has some domain-specific expressions (for example *InitAndSum*). Please note that variables are considered expressions by the AST as they are used in computations. The AST also contains some functions for simplifying the syntax of the language, which will be explained in the example in Section 6.4.

The C AST (Appendix A.3) is a trimmed-down representation of the C language, written for this generator specifically. Only the given statements and expressions are actually necessary for creating the HMM output code. Please note that any code written in the C AST is automatically generated by the translator. The "Program" section of the AST is analagous to the internal AST program definition, just as the declarations are analagous to a combination of the function signatures and global declarations from the internal AST. Methods are fairly self-explanatory (they require a declaration and a block of statements that will be executed when that method is run) and are analogous to the function declarations from the internal AST. Statements

```
stmt (I.Alloc (I.NDArr n s) t) =
    let name = I.NDArr n s
        m = (I.Malloc t (Length name)) in
    do
        declares n (ptr t)
        stmt (I.Assign name (m))
```

Figure 6.1: Translating an “Alloc” statement from the internal AST to the C AST

and expressions in the C AST fill the same roles as in the internal AST, however, the statements and expressions found in the C AST are C-specific. Therefore there are no specialised statements or expressions for the problem domain in this AST.

The Translator

The translator is an incredibly important component. It converts programs from the internal AST format to the format of the output language's AST (the C AST), complete with converting each of the components therein. This means that it contains functions for converting the more complicated functions into simpler C representations, for an example see Figure 6.1 which shows how a single statement like *Alloc* is converted to the C AST. Note that the translation for this statement calls for another (different) translation for it to complete. Many of the more complex translations require going through several calls before the translation of a single statement will complete.

The Printer

The printer renders the final output code in the proper syntax of the output language (in this case C). Essentially the printer acts similarly to the translator, except instead of moving between ASTs, it takes the C AST translation as input and outputs a C

```
retexpr :: String -> Expr -> Doc
retexpr self e = text "return" <+> expr self e <> semi
```

Figure 6.2: An example function from the printer

```
generate :: [Char] -> [Char] -> [D.Choices] -> IO ()
generate path alg ch =
    if (elem (D.Lang C) ch)
        then do outh <- openFile (path ++ alg ++ ".h") WriteMode
                hPutStrLn outh $ render $ C.header alg ch
                hClose outh
                outh <- openFile (path ++ alg ++ ".c") WriteMode
                hPutStrLn outh $ render $ C.code alg ch
                hClose outh
        else error "Invalid␣parameters."
```

Figure 6.3: The generate function in the design file

source file. Take, for example, the print function for *retexpr* shown in Figure 6.2. When called from the statement printing function it formats a return statement in C's grammar. If the code being generated had the line "Return (Int 0)" in the C AST translation, that would become "return 0;" in the final output code. Note that "semi" is a helper function for producing semicolons.

The Rest

The design file is written by the user to select the design choices that they wish to have implemented in the final version of the output code. It is also the file that houses the "generate" function (see Figure 6.3) which sets up the file names for the output code files and renders them. An example of the function used to generate the output code based on certain design decisions can be seen in Figure 6.4. Note that "D.Lang C" means the output code should be written in C, "D.Scale Scaled" means that the final output code should use the scaled versions of the implementation (which will

```
test2 = generate "" "HMMProg" [D.Lang C,
                              D.Scale Scaled,
                              D.MaxIt 1000]
```

Figure 6.4: Setting up the generator with specific design choices

```
dot = text "."
ques = text "?"
hash = text "#"
coldash = text ":-"
pipe = text "|"
dblslash = text "//"
dbldash = text "--"
backslash = text "\\"
unit = brackets empty
amp = text "&"
```

Figure 6.5: An example of some helper functions for the printer

be explained in depth in Chapter 7), and “D.MaxIt 1000” which sets the maximum number of iterations while training to 1000.

The helper functions are, for the most part, shorthand notation for certain terms used by the printer (see Figure 6.5). There is also a helper function included herein to set the value of the maximum number of iterations when training a model based on choices made in the design AST (Appendix A.1).

The implementation file is written in the internal language of the DSL (ASTInternal from Appendix A.2) and covers the implementation of all of the applicable variants based on the possible design choices from the design AST (Appendix A.1). The implementation file will be covered in Section 7.6. This file is used in conjunction with the design file to produce the final output code.

6.2 Some domain specific terms

This section will introduce some of the terms that were created specifically for the DSL as well as explain their use and relevance to the project. For a full listing of the statements and expressions included in the DSL please refer to Appendix A.2.

The first expression we will look at is the "Sum" expression shown here:

```
Sum Name Expr [(Expr, Expr)]
```

The "Sum" expression takes a string (shown as "Name" in the code) which is the variable to perform the summation over, along with the array being summed over (the first Expr in the definition) and a list of pairs containing a variable (of type Expr) to store the resulting sum in, and any mathematical computation to be included in the summation. For example the code

```
Sum i_ array [(ret_val, some_calc)]
```

would be equivalent to

$$\text{ret\_val} = \sum_{i=0}^{length(array)-1} \text{some\_calc}$$

where "some_calc" is a calculation dependent on 'i'.

Next we will look at is the "ProdSeq" expression shown here:

```
ProdSeq Name Expr [(Expr, Expr)]
```

The "ProdSeq" expression uses the exact same terms as the "Sum" expression. The main difference of course comes to the implementation. This "ProdSeq" is essentially equivalent to Π notation. As an example look at

```
ProdSeq i_ array [(ret_val, some_calc)]
```

would be equivalent to

$$\text{ret\_val} = \prod_{i=0}^{length(array)-1} \text{some\_calc}$$

where “some_calc” is a calculation dependent on ‘i’.

There are a few more expressions listed in Appendix A.2, but they should be fairly self explanatory. Next we will move on to statements.

Consider the following statement:

```
UniformDist Expr Name
```

The “UniformDist” statement takes an array / matrix (shown as “Expr”) and a placeholder string (shown as “Name” in the code) and initializes the array / matrix with uniform probability values. Essentially it produces an uninformative prior (which will be covered in Section 7.4). For example the code

```
UniformDist pi i
```

would be equivalent to

$$\text{pi}_i = 1/N, \quad i \in 0, 1, \ldots, N-1$$

where “N” is the length of the array “pi”. Similarly there is a non-uniform distribution statement as well (see Appendix A.2) for creating a row-stochastic matrix of non-uniform probability values.

What if we wanted to initialize a matrix with different values? Or perhaps only initialize a certain block of a matrix? That is where the "MatrixBlockInit" statement comes into play.

```
MatrixBlockInit Expr [Expr] [(Expr,Expr)] Expr
```

The "MatrixBlockInit" statement takes a matrix (the first "Expr"), a list of indices (shown as "[Expr]" in the code), a list of range pairs (shown as "[(Expr,Expr)]") which are of the form (start,end), and a value expression (the final "Expr") term. The indices and range pairs are used to determine where in the matrix to initialize, then the matrix is initialized in those places with the given value expression. For example the code

```
MatrixBlockInit array [i,j] [(0,4),(0,2)] val
```

would be equivalent to

$$\text{array}_{i,j} = \text{val}, \quad i \in 0, 1, \ldots, 4, \quad j \in 0, 1, 2.$$

Having each of these dedicated terms allows for some abstraction of the implementation details so a domain expert would be able to express their needs in a familiar language. Introducing a sum or sequence of products is now as simple as stating that one is needed, and there is no need to worry about the underlying implementation details. Initialization of matrices has also been abstracted away from the implementation details. This section was meant to be an introduction to some of the DSL terminology. Again, for the full list please see Appendix A.2.

6.3 Constructing a program using the DSL Syntax

Now that the concepts have been introduced, we will look deeper into the syntax of the DSL. To start with, a program has a fairly simple definition:

```
Program = Program [D.Choices] Name [GlobalDecl]
```

The entire program is essentially just a list of design choices, the program name, and a list of all of the global declarations. The global declarations themselves are:

```
GlobalDecl = CDecl Name [ParamDecl]
           | MACDecl Name String
           | Funct FuncSig [Stmt]
```

A macro declaration (MACDecl) is the simplest global declaration. It is used to define any global values or functions that may be called in the generated code (for example, in the final generated C code there will be a global value for the maximum number of HMM training iterations). To define a macro, all that is needed is a name to be referenced (a string) and what the value of that macro should be (another string).

A function declaration requires a function signature and a list of statements. The function signature is as follows:

```
FuncSig = FDecl Type Name [ParamDecl]
```

The function signature requires the return type, the name of the function (a string), and a list of parameter declarations which are of the form:

```
ParamDecl = PDecl Type Name
```

Essentially the "ParamDecl" term is used to declare the name and type of a variable being passed to the function. Parameter declarations are also used in class declarations to specify the members of a class and their types. An example of their use can be seen in the class declaration for HMMs:

```
hmmDecl :: [D.Choices] -> GlobalDecl
hmmDecl _ = CDecl "HMM" [PDecl int n_, PDecl int m_, PDecl (ptr dbl)
     initProbs_,PDecl (ptr dbl) transMatrix_, PDecl (ptr dbl)
    obsMatrix_]
```

This declaration shows that the class named "HMM" has the following parameters: 'n' and 'm' of type int; "initProbs","transMatrix", and "obsMatrix" of type double* (a pointer to a double in C, or conceptually an array/matrix). Also, note that the pointer type (ptr) is not a standalone type in the DSL. The syntax for pointers can be seen under the "Type" datatype in the internal AST (Appendix A.2). Essentially it is a function from Type $\rightarrow$ Type (so given any of the other types, a pointer of that type can be created).

Now when it comes to statements, each statement has its own syntax. The full list of statements can be seen in Appendix A.2, but some of them are not very self-explanatory. Alongside statements are expressions. A statement will always contain at least one expression (though the syntax may not appear to in all cases, see "Block"). Expressions, on the other hand, never contain statements.

6.3.1 Expressions

Variables and values are expressions as seen in this section of the "Expr" datatype definition:

```
| Var Variable
| Str String
| Int Integer
| Dbl Double
| NDArr Array [Size]
```

Breaking these down we have variables, strings, integers, doubles, and N-dimensional matrices. For strings, integers, and doubles the syntax is simply the expression type (Str, Int, or Dbl) followed by the value. For example, the syntax for the integer zero would be "Int 0", whereas if it were a double it would be "Dbl 0", or if it were a string it would be 'Str "0"'.

Variables are almost identical to strings, as "Variable" is a string. However, they are intended to be used where a variable would be found in the code (for example, the variable 'i' would be found in many places in the HMM implementation and would be written as 'Var "i"'). The main difference between variables and strings comes when they are being translated from the internal AST, as the translator will reject strings where a variable is required.

Finally, the N-dimensional matrix syntax uses an "Array" (which is just a placeholder similar to "Name" or "Variable" in that it is a string that represents the name of the N-D matrix), and a list of sizes. The list of sizes is the size of each dimension of the matrix, in order, and the sizes are represented by expressions. The actual syntax for a 2D matrix named "example" with 3 rows and 4 columns would be:

```
NDArr "example" [(Int 3), (Int 4)]
```

Now that we have taken a look at matrices and classes, there needs to be an expression to retrieve values/access members within them. This is where the dereferencing expressions come into play. There are two different dereferencing expressions, one for classes (“Deref”) and one for N-dimensional matrices (“NDArrDeref”). They can be seen here:

```
  Deref Expr Expr
| NDArrDeref Expr [Expr]
```

In each case, the first Expr is the data structure being dereferenced (a class for Deref, or an NDArr for NDArrDeref). For “Deref”, the second Expr is the member of the class being referenced (for example, from the HMM declaration that could be any of ‘n’,‘m’,“initProbs”,“transMatrix”, or “obsMatrix”). However, in the “NDArrDeref” the second term is a list of expressions, where each is an index value.

For example, to obtain the [‘i’,‘j’]th element in the matrix “Example” would look like:

```
NDArrDeref Example [(Var "i"), (Var "j")]
```

There are also functions for each version of dereference which were created to simplify the syntax. They are:

```
(@:) :: Expr -> Expr -> Expr
x @: e = Deref x e
(!:) :: Expr -> [Expr] -> Expr
x !: e = NDArrDeref x e
```

Op Syntax	**Meaning**
Expr :* Expr	Multiply two expressions together
Expr :+ Expr	Add two expressions together
Expr :/ Expr	Divide the first Expr by the second
Expr :- Expr	Subtract the second Expr from the first
Expr :> Expr	Check if the first Expr is greater than the second
Expr :< Expr	Check if the first Expr is less than the second
Expr :== Expr	Check if two expressions are equal
Expr :& Expr	Logical AND
Neg Expr	Negate the expression

Table 6.1: Math operations using expressions

Essentially, these functions provide an infix notation for the dereferencing expressions. The example of NDArrDeref above would now look like:

```
Example !: [(Var "i"),(Var "j")]
```

Next up there are unary and binary mathematical operations which can be performed using expressions. They use prefix and infix notation for unary and binary computations respectively. Table 6.1 shows each operator's syntax, and their meanings. Note that the logical AND operator and the operations that check values all return boolean truth values.

Expressions also cover the values returned from function calls, so there exists:

```
| Call Name [Expr]
```

Where "Name" is the name of the function to be called (a string) and there is a list of arguments to be passed to the function. This expression is only used when calling functions which return values. There is a seperate statement for handling procedure calls (used for calculating and manipulating values in memory, but not directly returning any).

Finally there is the "Malloc" expression which is currently an artifact left over from earlier versions and only used by the "Alloc" statement. The rest of the expressions were covered in Section 6.2, so it is time to take a look at statements.

6.3.2 Statements

To start, we will take a look at defining variables, assigning values to them, outputting them, and returning from functions. Each of those statements can be seen here:

```
| Define Type Expr
| Assign Expr Expr
| Out Expr
| Return Expr
```

These should be fairly self-explanatory. "Define" requires a type and an expression for a "Var" to define a variable. "Assign" takes a "Var" as the first expression and any expression as the second, assigning the expression to the variable. "Out" simply creates an output statement (such as printf in C) out of an expression, and "Return" is used at the end of a function definition to return an expression as the function's value.

The syntax for assignment and definition is fairly unwieldly, so a simplification function was created for each. A simplification for defining and assigning simultaneously was also created. These simplifications can be seen below:

```
(%:) :: Name -> Type -> Stmt
x %: t = Define t (Var x)
(=:) :: Expr -> Expr -> Stmt
x =: e = Assign x e
(=%:) :: Expr -> (Expr, Type) -> Stmt
```

```
x =%: (e,t) = Block [Define t x, Assign x e]
```

The following are examples of the functions' use.

- Defining a variable 'x' of type "int":

  ```
  "x" %: int
  ```

- Assigning a variable 'x' a value of 4:

  ```
  (Var "x") =: (Int 4)
  ```

- Combining the last two statements into one:

  ```
  (Var "x") =%: ((Int 4),int)
  ```

Since the generator is outputting to the C language, there must be some form of memory management. Specifically, arrays/matrices require memory allocation. The generator handles this with the "Alloc" statement, seen here:

```
Alloc Expr Type
```

Alloc requires an N-dimensional matrix as the expression, and the type of that matrix. From there it declares the variable (from the NDArr's name) with a pointer of the given type. Then, by looking at the NDArr's size and type, the generator computes the amount of memory to allocate for the given matrix. The name "Alloc" may be misleading, but it can essentially be thought of as a definition statement specific to "NDArr" terms. The translation of an "Alloc" statement can be seen in Figure 6.1. For a usage example, if a matrix of integers named "A" is being declared of size 2*2, then the "Alloc" statement would look like:

```
Alloc (NDArr "A" [2,2]) int
```

The following control flow statements are also included presently as they are currently necessary. However, at some point in the future they may be abstracted out and replaced with higher level statements:

```
| For Expr Expr Expr [Stmt]
| If Expr [Stmt] [Stmt]
| CondLoop Expr [Stmt]
```

The “If” statement takes a boolean operation as the first expression, followed by a list of statements to be performed if that expression evaluates to true, and finally a list of statements to be performed if the expression evaluates to false. The “Condloop” statement is a loop that requires a conditional expression (essentially an operation that returns a boolean value) and a list of statements to be performed so long as that expression returns a value of true.

The “For” statement takes a loop variable (typically a placeholder variable like ‘i’ or ‘j’) followed by expressions for the start and end values of the loop (for example, start at zero and end at ‘N’), and then finally a list of statements to be performed on each iteration of the loop. However, the “For” loop is now largely deprecated as the following “MatrixComps” statement has been created:

```
MatrixComps [Expr] [(Expr,Expr)] [Stmt]
```

This statement essentially works as a multi-dimensional “For” as it performs the list of statements on each pass through the given list of indices [Expr] and list of ranges [(Expr,Expr)]. This should look very similar to the MatrixBlockInit statement covered in Section 6.2 as it performs very similarly.

Finally is a statement alluded to at the end of Section 6.3.1, the procedure call expression:

```
CallProc Name [Expr]
```

This statement is only used when calling procedures that perform calculations and manipulate memory, but do not return a value themselves. From a C-language perspective, it can be seen as a way to call "void" methods.

6.4 Putting it all together: A simple Hello World program

This section will showcase an example of all the parts of the generator working together to produce a simple "Hello World" program in C. The first step is to set up an implementation file and a design file. For this simple "Hello World" program, the generate code from Figure 6.3 can be used, and the only choice parameter being passed will be "D.Lang C" to ensure that the output language will be C. The implementation file is pictured in Figure 6.6. Note that the implementation file is written using the terminology from the internal AST (Appendix A.2). The program is named "HelloProg" and it consists of one function named "hello" (which has a return type of void in C). The function itself consists of a single "Out" statement which outputs the expression "Hello World!" (which is a string).

With just those two files it is now possible to run the generator by compiling everything and running the generate function. When invoked, generate will call the "header" and "code" functions from the printer. Each of these functions takes the

```
module ImplHi where

import ASTInternal as I
import ASTDesign as D

helloProg :: [D.Choices] -> Program
helloProg ch = Program ch "HelloProg" [hello ch]

helloDecl :: [D.Choices] -> FuncSig
helloDecl _ = FDecl VoidType "hello" []

hello :: [D.Choices] -> GlobalDecl
hello ch = Funct (helloDecl ch) $
     [ Out (I.Str "Hello␣World!")
     ]
```

Figure 6.6: A "hello world" implementation in the HMM Generator

program contained in the implementation file as an input and generates the appropriate .h and .c file respectively.

Each of the "header" and "code" functions call the "toImp" function from the translator on the implementation file. This function translates the program from using the internal AST to the C AST and builds the appropriate library list, declarations, and methods. The declarations and methods are translated by mapping the functions shown in Figure 6.7 over the list of global declarations from the implementation.

Next the "header" and "code" functions call "printH" and "printC" (seen in Figure 6.8), respectively. The former builds and inserts a list of libraries into the header file, followed by the prototypes of the C methods that will be used. To build those prototypes, the "buildD" function (See Figure 6.9) is mapped over the declaration list from the translated implementation. Similarly, the "printC" function includes the necessary header file (which was just created) and then builds all of the methods from the method list in the translated implementation. It builds the methods by mapping the "buildM" function (Figure 6.9) over the list of methods, which in

```
buildD :: I.GlobalDecl -> C.Declaration
buildD (I.MACDecl n v) = C.MACDecl n v
buildD (I.CDecl n pL) = C.SDecl n $ params pL
buildD (I.Funct decl _) = buildFS decl

buildFS :: I.FuncSig -> C.Declaration
buildFS (I.FDecl t n pL) = C.MDecl (typ t) n $ params pL

buildM :: I.GlobalDecl -> C.Method
buildM (I.Funct decl methlist) = C.Meth (buildFS decl) $ C.Block .
   sm2s $ stmts methlist
buildM _ = C.NullMeth
```

Figure 6.7: Functions for translating declarations and methods to the C AST

```
printC :: Program -> Doc
printC (Program name _ _ mds) = hash <> text "include" <+> quotation
    (text name <> text ".h") $$ vcat (map buildM mds)

printH :: Program -> Doc
printH (Program _ libs decls _) = libraries libs $$ vcat (map buildD
    decls)
```

Figure 6.8: The “printC” and “printH” functions in the printer

```
buildD :: Declaration -> Doc
buildD (MACDecl name val) = hash <> text "define" <+> text (allcap
   name) <+> text val
buildD (SDecl name pL) = text "typedef␣struct" <+> lbrace $$ nest 4
   (members pL) $$ rbrace <+> text name <> semi
buildD (MDecl t name pL) = typ t <+> text name <> parens (params pL)
    <> semi
buildD _ = empty

buildM :: Method -> Doc
buildM (Meth (MDecl t name pL) statements) = typ t <+> text name <>
   parens (params pL) <+> lbrace $+$ nest 4 (stmt name statements)
   $$ rbrace
buildM _ = empty
```

Figure 6.9: The functions for building the prototypes and methods in C

```
stmt self (Out n) = text "printf" <> parens (expr self n) <> semi
```

Figure 6.10: Printing the “Out” statement

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<limits.h>
void hello();
```

Figure 6.11: The “HelloWorld” header file

turn calls the appropriate code to convert to writable C code. For the “HelloWorld” implementation that is the function seen in Figure 6.10 where “expr” (in this case) returns “Hello World” in double quotes to denote a string.

The final source files can be seen in Figures 6.11 and 6.12 for the .h and .c files respectively. Note that currently the libraries are predetermined based on the needs of the HMM Generator, but they may be customizable in the future through the design file.

```
#include "HelloProg.h"
void hello() {
    printf("Hello␣World!");
}
```

Figure 6.12: The “HelloWorld” c file

Chapter 7

HMM Implementation

This chapter will look at the various implementations of Hidden Markov Models across languages (Haskell, C, and the HMM Generator) based on the math from Chapter 3. The algorithms for solving the three fundamental problems (see Sections 3.3 & 3.4) will be explained in detail and the method for computing them efficiently will be shown in each language.

First off, one must understand how HMMs are represented in each of the languages. In the Haskell version of the HMM implementation, the HMM datatype can be found in Figure 7.1. The HMM in Haskell allows for values of any type (the set of states/observations can be ints, strings, etc). These types are used to declare lists of

```
>data HMM stateType eventType =
>       HMM { states :: [stateType]
>       , events :: [eventType]
>       , initProbs :: (stateType -> LogFloat)
>       , transMatrix :: (stateType -> stateType -> LogFloat)
>       , obsMatrix :: (stateType -> eventType -> LogFloat)
>       }
```

Figure 7.1: HMM Representation in Haskell

```
typedef struct {
    int m,n;
    double *initProbs;
    double *transMatrix;
    double *obsMatrix;
} HMM;
```

Figure 7.2: HMM Representation in C

```
hmmDecl :: [D.Choices] -> GlobalDecl
hmmDecl _ = CDecl "HMM" [PDecl int n_, PDecl int m_, PDecl (ptr dbl)
     initProbs_,PDecl (ptr dbl) transMatrix_, PDecl (ptr dbl)
   obsMatrix_]
```

Figure 7.3: HMM Representation in the HMM Generator

those types which will be populated by the state and observation values/names.

In the C implementation of the HMM code, HMMs are represented as depicted in Figure 7.2. This struct only allows for an integer representation of states / observations, therefore only the lengths of the state and event lists are necessary in the code. However, some bookkeeping will be necessary in order to translate between numeric representations and meaningful names.

In the HMM Generator the HMM is represented as depicted in Figure 7.3. Note that the terms in the comma-separated list are the members of the class (the class being "HMM"). They correspond exactly to the C language implementation ("n,m" are declared as being of type *int* and the matrices as *double**). In order to fully understand the definitions, refer to Chapter 6.

7.1 The forward algorithm

The algorithm for computing $\alpha_t(i)$ (from Equation 3.2) is as follows:

1. Let $\alpha_0(i) = \pi_i b_i(\mathcal{O}_0)$ for $i = 0, 1, \ldots, N-1$

2. For $i = 0, 1, \ldots, N-1; t = 1, 2, \ldots, T-1$ compute:

$$\alpha_t(i) = \left[\sum_{j=0}^{N-1} \alpha_{t-1}(j) a_{ji}\right] b_i(\mathcal{O}_t)$$

3. Then (from Equation 3.2)

$$P(\mathcal{O}|\lambda) = \sum_{i=0}^{N-1} \alpha_{T-1}(i) \tag{7.1}$$

Note that the forward algorithm can be computed recursively meaning it now requires $\sim N^2T$ multiplications (an improvement over the previous $\sim 2TN^T$ multiplications laid out in Section 3.4).

Comparing implementations of the forward algorithm

Please note that The HMM Generator version of this algorithm will be discussed in Section 7.6.

Figures 7.4 and 7.5 show how the values for $\alpha_t(i)$ are calculated in Haskell and C, respectively. The Haskell implementation uses memoization for efficiency; however, the actual calculations in both the C and Haskell versions are identical to those found at the beginning of Section 7.1.

```
>alpha hmm obs = a_1
> where a_1 t state = (Memo.memo2 Memo.integral Memo.integral a_2) t
    (sIndex hmm state)
>       a_2 t' state'
>        | t' == 1 = (obsMatrix hmm (states hmm !! state') $ obs!t')
>                    *(initProbs hmm $ states hmm !! state')
>        | otherwise = (obsMatrix hmm (states hmm !! state') $ obs!t
   ')
>                    *(sum [(a_1 (t'-1) state2)
>                    *(transMatrix hmm state2 (states hmm !! state')
   ) | state2 <- states hmm])
```

Figure 7.4: $\alpha_t(i)$ computation in Haskell

```
double* alpha_array(HMM *hmm, int* obs, int num_obs){
    int n = hmm->n;
    double* a = (double *)malloc(sizeof(double)*num_obs*n);
    int i,j,t;
    // printf("\nT = %d\n", num_obs);
    for (i = 0; i < n; i++){
        a[i] = (hmm->initProbs[i])*(getOMVal(i,obs[0],hmm));
         //printf("\nAlpha[0][%d] = %f",i,a[i]);
    }
    for (t = 1; t < num_obs; t++){
        for (i = 0; i < n; i++){
            a[t*n+i] = 0;
            for (j = 0; j < n; j++){
                a[t*n+i] += a[(t-1)*n+j]*(getTMVal(j,i,hmm));
            }
            a[t*n+i] *= getOMVal(i,obs[t],hmm);
             //printf("\nAlpha[%d][%d] = %f",t,i,a[t*n+i]);
        }
    }
    return a;
}
```

Figure 7.5: $\alpha_t(i)$ computation in C

```
>alpha_Array :: (Eq eventType, Eq stateType, Show eventType, Show
   stateType) => HMM stateType eventType -> Array Int eventType ->
   LogFloat
>alpha_Array hmm obs = sum [alpha hmm obs b_T state | state <-
   states hmm] --Sum partial probabilities
>    where b_T = snd $ bounds obs
```

Figure 7.6: α-pass in Haskell

```
double alpha_pass(HMM *hmm, double* a_A, int num_obs){
    double ret_val;
    int n = hmm->n;
    int i;
    for (i = 0; i < n; i++){
        // printf("A[T-1][i] = %f\n",a_A[(num_obs - 1)*n+i]);
        ret_val += a_A[(num_obs - 1)*n+i];
    }
    return ret_val;
}
```

Figure 7.7: α-pass in C

From there it is a very simple task to compute the final value of the α-pass by summing over the index ‘i’ as shown in Equation 7.1. The necessary code for computing the final sums can be seen in Figures 7.6 and 7.7 for Haskell and C respectively.

7.2 The backward algorithm

The algorithm for computing $\beta_t(i)$ (from Equation 3.4) is as follows:

1. Let $\beta_{T-1}(i) = 1$ for $i = 0, 1, \ldots, N-1$

2. For $i = 0, 1, \ldots, N-1; t = T-2, T-3, \ldots, 0$ compute:

$$\beta_t(i) = \sum_{j=0}^{N-1} a_{ij} b_j(\mathcal{O}_{t+1}) \beta_{t+1}(j)$$

Please note that the solution to finding the optimal state sequence requires the gamma values which will be discussed in Section 7.3.

Comparing implementations of the backward algorithm

Please note that The HMM Generator version of this algorithm will be discussed in Section 7.6.

Figures 7.8 and 7.9 show how the values for $\beta_t(i)$ are calculated in Haskell and C respectively. Again, the Haskell implementation uses memoization for efficiency, however the actual calculations in both the C and Haskell versions are identical to those found at the beginning of Section 7.2.

```
>beta hmm obs = b_1
> where b_T = snd $ bounds obs
>       b_1 t state = (Memo.memo2 Memo.integral Memo.integral b_2) t
    (sIndex hmm state)
>       b_2 t' state'
>           | t' == b_T = 1
>           | otherwise = sum [(transMatrix hmm (states hmm !! state
   ') state2)
>                       *(obsMatrix hmm state2 $ obs!(t'+1))
>                       *(b_1 (t'+1) state2)| state2 <- states hmm]
```

Figure 7.8: $\beta_t(i)$ computation in Haskell

```
double* beta_array(HMM *hmm, int* obs, int num_obs){
    int n = hmm->n;
    double* b = (double *)malloc(sizeof(double)*num_obs*n);
    int i,j,t;
    // printf("\nT = %d\n", num_obs);

    for (i = 0; i < n; i++){
        b[(num_obs-1)*n+i] = 1;
        //printf("\nBeta[%d][%d] = %f",(num_obs-1),i,b[(num_obs-1)*n
            +i]);
    }
    for (t = num_obs-2; t >= 0; t--){
        for (i = 0; i < n; i++){
            b[t*n+i] = 0;
            for (j = 0; j < n; j++){
                b[t*n+i] += b[(t+1)*n+j]*(getTMVal(i,j,hmm))*
                    getOMVal(j,obs[t+1],hmm);
            }
            //printf("\nBeta[%d][%d] = %f",t,i,b[t*n+i]);
        }
    }
    return b;
}
```

Figure 7.9: $\beta_t(i)$ computation in C

```
>gamma hmm obs = g_1
>   where g_1 t' state' = ((alpha hmm obs t' state')*(beta hmm obs t
    ' state'))
>                           / (beta_Array hmm obs)
```

Figure 7.10: $\gamma_t(i)$ computation in Haskell

7.3 Finding the optimal path using $\gamma_t(i)$

Finding the optimal path involves understanding how to compute $\gamma_t(i)$ first; recall Equation 3.6. Calculating the $\gamma_t(i)$ values requires the $\alpha_t(i)$and $\beta_t(i)$ values being multiplied together and then dividing that value by $P(\mathcal{O}|\lambda)$ which is known to be equal to the sum of the α_{T-1} values (see Section 7.1). Then all that is left is to find the state q_i that maximizes $\gamma_t(i)$ for each time period t.

Comparing implementations of the $\gamma_t(i)$ calculations and pathfinding

Figures 7.10 and 7.11 show the implementations of the $\gamma_t(i)$ calculations in Haskell and C respectively.

At this point, finding the optimal path becomes trivial as shown in the Haskell (Figure 7.12) and C (Figure 7.13) implementations.

```
double* gamma_array(HMM *hmm, int* obs, int num_obs, double* a_A,
   double* b_A){
    int n = hmm->n;
    double* g = (double *)malloc(sizeof(double)*num_obs*n);
    int i,t;
    //Remember, arrays here are of the form A[t][i] so do arithmetic
        properly
    for (i = 0; i < n; i++){
        for (t = 0; t < num_obs; t++){
            g[t*n+i] = a_A[t*n+i]*b_A[t*n+i] / alpha_pass(hmm, a_A,
               num_obs);
            //printf("\nGamma[%d][%d] = %f",t,i,g[t*n+i]);
        }
    }
    return g;
}
```

Figure 7.11: $\gamma_t(i)$ computation in C

```
>pathFindHMM hmm obs = [Memo.integral findmax t | t <- [1..b_T]]
>   where b_T = snd $ bounds obs
>         findmax t = argmax (\i -> gamma hmm obs t i) (states hmm)
```

Figure 7.12: Finding the optimal path in Haskell

```
state* pathFindHMM(HMM *hmm, int* obs, int num_obs,double* g_A){
    int n = hmm->n;
    int i,t;
    double max;
    state* ret_array = (state *)malloc(sizeof(state)*num_obs);
    for (t = 0;t < num_obs; t++){
        max = 0;
        for (i = 0; i < n; i++){
            if (g_A[t*n+i] > max) {
                max = g_A[t*n+i];
                ret_array[t] = i;
            }
        }
    }
    return ret_array;
}
```

Figure 7.13: Finding the optimal path in C

7.4 Re-estimating $A, B,$ and π

Computing the di-gammas at this point is trivial thanks to Equation 3.8. Re-estimation of $A, B,$ and π is an iterative process. First the model must be initialized with a best guess for $A, B,$ and π. If there is not enough information to reasonably guess, then choose random values such that $\pi_i \approx 1/N, a_{ij} \approx 1/N,$ and $b_j(k) \approx 1/M$. It is very important that A and B are randomized, as uniform values can result in a local maximum from which the model will not improve. π can be uniform (often called an "uninformative prior" determined by the principle of indifference) or randomized around $1/N$, with the latter often preferred. From there, training of $A, B,$ and π is accomplished using the following algorithm:

1. Initialize the model $\lambda = (A, B, \pi)$
2. Compute the $\alpha_t(i), \beta_t(i), \gamma_t(i, j),$ and $\gamma_t(i)$
3. Re-estimate the model($\lambda = (A, B, \pi)$) using Equations 3.9, 3.10, and 3.11
4. If $P(\mathcal{O}|\lambda)$ increases, repeat from step 2.

Comparing implementations of HMM training

The actual computations for the re-estimation of $A, B,$ and π are trivial at this point. As such their implementations will be largely ignored in this section, as will the implementations of the initialization functions.

In Figure 7.14 training is fairly straightforward. First the iteration counter is checked and then (if the remaining number of iterations is greater than 0) training is called on an updated model named "itr" with one less iteration. The model parameters are re-estimated using the functions "newInitProbs", "newTransMatrix", and

```
>train hmm obs count
>    | count == 0    = hmm
>    | otherwise     = train itr obs (count-1)
>        where itr = trainItr hmm obs
>
>trainItr hmm obs =
>   HMM { states = states hmm
>   , events = events hmm
>   , initProbs = newInitProbs
>   , transMatrix = newTransMatrix
>   , obsMatrix = newObsMatrix
>   }
```

Figure 7.14: Training a model in Haskell

```
void re_est(HMM *hmm, int* obs, int num_obs){
    int iters = 0;
    int i,t;
    int n = hmm->n;
    double oldProb = INT_MIN;
    double newProb = 0;
    double a_P;
    double *a_A,*b_A,*g_A;
    double* dg_A = (double *)malloc(sizeof(double)*num_obs*n*n);
    while (iters < MAX_ITERS && newProb > oldProb){
        oldProb = newProb;
        a_A = alpha_array(hmm, obs, num_obs);
        b_A = beta_array(hmm, obs, num_obs);
        a_P = alpha_pass(hmm, a_A, num_obs);
        dg_A = digamma_array(hmm, obs, num_obs, a_A, b_A, a_P);
        g_A = gamma_array(hmm, obs, num_obs, a_A, b_A);
        est_pi(hmm, g_A);
        est_A(hmm, num_obs, g_A, dg_A);
        est_B(hmm, obs, num_obs, g_A);
        newProb = alpha_pass(hmm, a_A, num_obs);
        iters++;
    }
}
```

Figure 7.15: Training a model in C

"newObsMatrix" called from inside the "trainItr" function. Each of those re-estimates one of π, A, and B respectively.

Figure 7.15 shows how a model would be trained in C. The function loops until either the max number of iterations is reached, or the new $P(\mathcal{O}|\lambda)$ is no longer increasing. The lines from "a_A" to "g_A" are used for bookkeeping, in order to retrieve the necessary values for the re-estimation calculations and then "est_pi", "est_A", and "est_B" compute the re-estimated values of π, A, and B respectively.

7.5 Scaling HMMs

There is one fairly major problem with the computation methods explained thus far, namely that the computations involve calculating products of probabilities. This would not be an issue if the calculations were being performed with infinite precision, but that is not the case. It is a given that $\alpha_t(i)$ tends to 0 as T increases. This will inevitably result in underflow.

There is, however, a solution that will avoid underflow and that solution is to scale the numbers being used in the calculations. In order to use scaled values of $\alpha_t(i)$ and $\beta_t(i)$, the re-estimation formulae must remain valid. This was covered in great detail by Stamp (2004) and it was shown that the re-estimation formulae do, in fact, remain valid with a scaling factor (c_t) equal to

$$c_t = \frac{1}{\sum_{j=0}^{N-1} \alpha_t(j)} \tag{7.2}$$

.

The most interesting result of using this scaling factor is that it results in a few simplifications that result in never having to calculate $P(\mathcal{O}|\lambda)$ again. To clarify, let $\hat{\alpha}_t(i)$ be the scaled $\alpha_t(i)$. Since

$$\sum_{j=0}^{N-1} \hat{\alpha}_{T-1}(j) = 1$$

and the scaling factor is defined such that $\hat{\alpha}_t(i)$ is equal to $\alpha_t(i)$ multiplied by $(c_0 c_1 \ldots c_t)$ then,

$$\sum_{j=0}^{N-1} \hat{\alpha}_{T-1}(j) = (c_0 c_1 \ldots c_t) \sum_{j=0}^{N-1} \alpha_{T-1}(j).$$

Note the last sum is equal to $P(\mathcal{O}|\lambda)$ from Equation 7.1. Substituting the previous two equations into each other gives:

$$P(\mathcal{O}|\lambda) = \frac{1}{\prod_{t=0}^{T-1} c_t}$$

or more importantly:

$$\log[P(\mathcal{O}|\lambda)] = -\sum_{t=0}^{T-1} c_t \tag{7.3}$$

The total sum of the scaling factors can now be used in place of $P(\mathcal{O}|\lambda)$ as a means of determining whether or not it is increasing after each training iteration, thus avoiding the problem of products of probabilities causing underflow.

All of this was used in C to implement a version of the HMM which uses scale factors, however the code is nearly identical to that shown in the previous sections (with the exception of the trivial step of adding a scaling factor where necessary).

```
void re_estimate(HMM *hmm, int* obs, int num_obs){
    int iters = 0;
    int i,t;
    int n = hmm->n;
    double logProb = 0;
    double oldLogProb = INT_MIN;
    double *a_A,*b_A,*g_A;
    double c[num_obs];
    double* dg_A = (double *)malloc(sizeof(double)*num_obs*n*n);
    while (iters < MAX_ITERS && logProb > oldLogProb){
        oldLogProb = logProb;
        a_A = scaled_a_A(hmm, obs, num_obs, c);
        b_A = scaled_b_A(hmm, obs, num_obs, c);
        g_A = compute_scaled_gamma(hmm, obs, num_obs, a_A, b_A, dg_A
           );
        est_pi(hmm, g_A);
        est_A(hmm, num_obs, g_A, dg_A);
        est_B(hmm, obs, num_obs, g_A);
        logProb = 0;
        for (i = 0; i < num_obs; i++) {
            logProb += log(c[i]);
        }
        logProb = -logProb;
        iters++;
        printf("\nLogProb␣=␣%f", logProb);
    }
}
```

Figure 7.16: Scaled training code in C

Figure 7.16 shows the modified code; note the scaling factor *'c'* and the use of *"logProb"*. The Haskell implementation, however, did not require scaling as there was a pre-existing library called "LogFloat" which automatically handled converting to the log domain and working with the values there, thus eliminating the need for an explicit scaling factor.

7.6 HMMs in the HMM Generator

Note: The scaling mentioned in the code was introduced in Section 7.5.

This section will show the HMM Generator representation of the HMM implementation. For the final generated C code, please see Appendix B.

The forward algorithm

Refer to Figure 7.17 for the code which computes the $\alpha_t(i)$ values. The first few lines of code (the "let" block) is merely included for readability's sake. The statements prior to the "if" declaration are the necessary declarations of variables for each version, and the "if" statement handles the design choices that are made when running the generator (as mentioned in Chapter 6). If the chosen implementation is to be scaled, then the $\alpha_t(i)$ values are computed alongside the c_t and the new scaled values are returned. Otherwise, the scaling is thrown away and a much simpler version of the code can be seen.

In Figure 7.18 it is clear that the α-pass code written in the HMM Generator looks very similar to the original Haskell, with the exception of the necessary variable declarations. It also outlines the simplicity of computing a sum using the DSL, as it only requires one command.

The backward algorithm

Figure 7.19 shows the HMM Generator code for computing the $\beta_t(i)$ values. It looks very similar to the code for computing $\alpha_t(i)$ (Figure 7.17). The "let" block is again included for readability's sake and the statements prior to the "if" declaration are

```
alphaArray ch = useCounter_ijt $ \i_ j_ t_ ->
    let i = v i_
        j = v j_
        t = v t_
        scaled = elem (D.Scale Scaled) ch
        hmmIPi = hmm @: (initProbs !: [i])
        callGetOM = Call getOMVal [i, (obs !: [zero]), hmm]
        c0 = scaling !: [zero]
        ct = scaling !: [t]
        sum = Sum j_ n [((a_A !: [t,i]),((a_A !: [(t :- (Int 1)),j])
            :* (Call getTMVal [j,i,hmm])))] in
    Funct (alphaArrayDecl ch) $
    [ n =%: ((hmm @: n),int),
      Alloc a_A dbl
      ] ++
      (if scaled then [
        c0 =: ((Dbl 1.0) :/ (InitAndSum i_ n [(c0,(a_A !: [i]))] [((
            a_A !: [i]),(hmmIPi :* callGetOM))])),
        compute $ ProdSeq i_ n [((a_A !: [i]),(c0))],
        For t (Int 1) num_obs $
          [ ct =: ((Dbl 1.0) :/ (InitAndSum i_ n [(ct, (a_A !: [t,i
              ]))] [((a_A !: [t,i]),(sum :* (Call getOMVal [i, (obs
              !: [t]), hmm])))])),
            compute $ ProdSeq i_ n [((a_A !: [t,i]) , ct)]
          ]
        ]
      else [
        Init (a_A !: [i]) i n (hmmIPi :* callGetOM),
        For t (Int 1) num_obs $
        [ Init (a_A !: [t,i]) i n (sum :* (Call getOMVal [i, (obs !:
            [t]), hmm])) ]
        ] ) ++ [
      Return a_A
    ]
```

Figure 7.17: α computation in the HMM Generator

```
alphaPass ch = useCounter_i $ \i_ ->
    let i = v i_ in
    Funct (alphaPDecl ch) $
    [ ret_val_ %: dbl,
      n =%: ((hmm @: n),int),
      Return $ Sum i_ n [(ret_val, a_A !: [num_obs :- (Int 1),i])]
    ]
```

Figure 7.18: α-pass in the HMM Generator

```
betaArray ch = useCounter_ijt $ \i_ j_ t_ ->
    let i = v i_
        j = v j_
        t = v t_
        scaled = elem (D.Scale Scaled) ch
        callGetOM = Call getOMVal [j, (obs !: [t :+ (Int 1)]), hmm]
        bNum_1 = b_A !: [(num_obs :- (Int 1)),i]
        cNum_1 = scaling !: [num_obs :- (Int 1)]
        ct = scaling !: [t]
        sum = Sum j_ n [((b_A !: [t,i]),((b_A !: [(t :+ (Int 1)),j])
            :* ((Call getTMVal [i,j,hmm]) :* (callGetOM) )))] in
    Funct (betaArrayDecl ch) $
    [ n =%: ((hmm @: n),int),
      Alloc b_A dbl] ++
      (if scaled then [
        Init bNum_1 i n cNum_1,
        MatrixBlockInit b_A [t,i] [((num_obs :- (Int 2)),zero),(zero
            ,n)] (ct :* sum)
       ]
       else [
        Init bNum_1 i n (Int 1),
        MatrixComps [t,i] [(num_obs :- (Int 2),zero),(zero,n)] [
            compute sum]
       ]) ++ [
      Return b_A
    ]
```

Figure 7.19: β computation in the HMM Generator

the necessary declarations of variables for each version, where the "if" statement handles the design choices as previously mentioned. If the chosen implementation is to be scaled, then the $\beta_t(i)$ values are computed and scaled using the c_t values determined from the α-pass. Otherwise, a very similar block of code is used with the key difference being that it lacks scaling.

The β-pass implementation seen in Figure 7.20 should look very familiar as it is almost identical (barring the "let" block) as the α-pass implementation.

```
betaPass ch = useCounter_i $ \i_ ->
    let i = v i_
        beta_var = (b_A !: [i]) :* (callGetOM :* hmmIPi)
        hmmIPi = hmm @: (initProbs !: [i])
        callGetOM = Call getOMVal [i, (obs !: [zero]), hmm] in
    Funct (betaPDecl ch) $
    [ ret_val_ %: dbl,
      Return $ Sum i_ (hmm @: n) [(ret_val, beta_var)]
    ]
```

Figure 7.20: β-pass in the HMM Generator

```
not_sgamma ch = useCounter_it $ \i_ t_ ->
    let i = v i_
        t = v t_
        not_sg = ((a_A !: [t,i]) :* (b_A !: [t,i])) :/ ap
        ap = Call alpha_pass [hmm, (a_A), num_obs] in
    Funct (gammaDecl ch) $
    [ n =%: ((hmm @: n),int),
      t_ %: int,
      Alloc g_A dbl,
      For i zero n $
        [ Init (g_A !: [t,i]) t num_obs not_sg ],
      Return g_A
    ]
```

Figure 7.21: A non-scaled γ computation in the HMM Generator

Pathfinding in the HMM Generator

The pathfinding implementation is incredibly straightforward and as such will not be examined in depth. It merely traverses the $\gamma_t(i)$ values for each "t" and finds the state "i" which maximizes that value (as explained in Section 3.4). However, the $\gamma_t(i)$ calculations can be done in one of two ways. The first method of calculating the $\gamma_t(i)$ values is the straightforward application of the math behind them, which looks very similar to the Haskell and C implementations from Section 7.3 and can be seen in Figure 7.21. Note that it is called the non-scaled version as the computation directly relies on $P(\mathcal{O}|\lambda)$ which will never be calculated if using scaling factors.

```
Funct (gammaDecl ch) $
[ n =%: ((hmm @: n),int),
  Alloc g_A dbl,
  temp_ %: dbl,
  For t zero (num_obs :- (Int 1)) $
    [ compute tempSum,
      For i zero n $
      [ compute $ InitAndSum j_ n [((g_A !: [t,i]),dg_A_Val)] [(
          dg_A_Val,digamma_calc)]  ]
    ],
  Return g_A
]
```

Figure 7.22: Scaled γ computation in the HMM Generator

The statements in Figure 7.22 show the process for computing γ while using scaling factors. This $\gamma_t(i)$ computation relies solely on the $\gamma_t(i,j)$ values which are computed by the HMM Generator code in the same statement. A fairly long "let" block was omitted from the figure, as it is only the definitions of the terms (which should be self-explanatory).

Training a model in the HMM Generator

Again the computations for A, B, and π will be omitted as they are straightforward. The training code can be found in Figure 7.23. Note that in this case, the "let" block is used both for readability and for determining whether or not the code should be using scaling factors. Training in the HMM Generator looks very similar to the C implementation as it includes the same variable declarations followed by a conditional loop that re-estimates A, B, and π iteratively until either the max number of iterations is reached or $P(\mathcal{O}|\lambda)$ is no longer increasing.

.

```
let i = v i_
    while_exp = check1 :& check2
    check1 = iters :< (max_iters)
    check2 = logProb :> oldLogProb
    scaled = elem (D.Scale Scaled) ch
    params0 = [hmm, obs, num_obs]
    params = params0 ++ (if scaled then [scaling] else [])
    gparams = params0 ++ [a_A, b_A] ++ (if scaled then [dg_A]
       else [])
    log = (\x -> Call "log" x) in
Funct (re_estDecl ch) $
[ n_ %: int,
  iters =%: (zero,int),
  logProb_ %: dbl,
  oldLogProb_ %: dbl,
  Alloc scaling dbl,
  a_A_ %: (ptr dbl),
  b_A_ %: (ptr dbl),
  g_A_ %: (ptr dbl),
  n =: (hmm @: n),
  logProb =: (int_min :+ (Int 1)),
  oldLogProb =: int_min,
  Alloc dg_A dbl,
  CondLoop while_exp $
    [ oldLogProb =: logProb,
      a_A =: Call alpha_a params,
      b_A =: Call beta_a params,
      g_A =: Call gamma gparams,
      CallProc est_pi [hmm, g_A],
      CallProc est_A [hmm, num_obs, g_A, dg_A],
      CallProc est_B [hmm, obs, num_obs, g_A],
      logProb =: (Neg (Sum i_ scaling [(logProb,(log [scaling !:
          [i]]))])),
      iters =: (iters :+ (Int 1))
    ]
]
```

Figure 7.23: Training a model in the HMM Generator

Chapter 8

Testing

Testing for the HMM Generator is comprised of two major cases.

1. Ensuring the generator works.

2. Ensuring the generated source code works and that it represents the design.

8.1 Testing the generator

Testing the generator involved compilation tests, execution tests, and testing for erroneous inputs. The first two sets of tests should be fairly self-explanatory.

Compilation tests were merely to ensure that the framework for the generator itself compiled without any issue, whereas execution tests were to ensure that the generator (when run) would actually produce some output code.

The final set of tests (for erroneous input) handled attempts to use the DSL in unexpected ways. An example of such would be attempting to dereference an array on any non-array type, which should throw an error. There were many similar tests, each of which confirmed that the generator would not allow unexpected input.

These tests were necessary due to the way the AST represented the terms. All variable types were implemented as expressions, so a series of checks needed to be performed to ensure that the appropriate expression type was being used in each case.

The majority of the testing was performed after each iteration of the generator was completed and revised (this was, of course, because of the agile software development model which was being used in creating the generator).

The last test performed was an execution test used in conjunction with the HMM implementation file in order to ensure that the generator was outputting the correct source code.

8.2 Testing the generated code

Testing the source code can be represented by two types of tests. The first being whether or not the generated source code performs the task it was designed for, and the second is whether or not the source code represents the design that was specified.

All of the tests were run on each variant of the generated code (essentially, on the scaled and non-scaled versions of the HMM implementation).

8.2.1 Does it work?

To test whether or not the code actually worked, it was run through a series of automated test cases which were verified through external means. The output of the automated tests was going to be added to the appendix, however, the re-estimation tests ended up being far too large (spanning hundreds of pages) for that. What follows will be examples of the types of tests used on each of the source variants.

The α-pass

As an example of this test case, consider the following model $\lambda = (\pi, A, B)$ such that:

$$\pi = \begin{bmatrix} 0.6 & 0.4 \end{bmatrix} \quad A = \begin{bmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{bmatrix} \quad B = \begin{bmatrix} 0.1 & 0.4 & 0.5 \\ 0.7 & 0.2 & 0.1 \end{bmatrix}$$

Given the observation sequence $\mathcal{O} = (0, 1, 0, 2)$ find $P(\mathcal{O}|\lambda)$. Through pen and paper calculations, the probability of the given state sequence was found to be 0.0096296.

Testing the α-pass for the scaled version of the HMM implementation is incredibly simple. The use of scaling factors ensures that an α-pass will always result in a value of '1' (by definition of the scaling factor, see Equation 7.2). However, determining the probability of the observation sequence is a little bit more difficult.

To determine whether the $P(\mathcal{O}|\lambda)$ is correct, there are several methods. The first is to take the inverse log of the negative sum of the scaling factors to compute the actual value (this makes scaling redundant). The second is to determine the c_t values and calculate the probability as $\frac{1}{\prod_{t=0}^{T-1} c_t}$, again, essentially ignoring the point of scaling. Using either of these two methods, the result obtained from running this test on the generated source code was $P(\mathcal{O}|\lambda) = 0.009630$ which is the result of C rounding the value.

Testing the α-pass for the non-scaled version was much simpler. As the HMM used in this test was relatively small (in terms of the number of possible states/observations) this ensured that underflow would not be an issue. Thus, it only required calling the α-pass function and checking the value, which was 0.009630, again due to rounding.

To test the α-pass fully, both the non-scaled and scaled versions were subjected to a series of other tests using different models and observation sequences, and the resulting values were confirmed through external means (in most cases, a slight variance was allowed due to rounding, this will be addressed in a later section). Some of the models resulted in underflow for the non-scaled version of the source (as expected), and the rest of the tests passed on each version without issue.

A simple pathfinding case

For this test case, suppose the same model from the α-pass example was used, that is $\lambda = (\pi, A, B)$ such that:

$$\pi = \begin{bmatrix} 0.6 & 0.4 \end{bmatrix} \quad A = \begin{bmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{bmatrix} \quad B = \begin{bmatrix} 0.1 & 0.4 & 0.5 \\ 0.7 & 0.2 & 0.1 \end{bmatrix}$$

Given the same observation sequence as before, $\mathcal{O} = (0, 1, 0, 2)$, find the most likely state sequence.

As mentioned in Chapter 3, this test will use the HMM approach to the "most likely" state sequence. This means it will look for the path which maximizes the expected number of correct states. Pen and paper calculations reveal the path to be $S = 1, 0, 1, 0$.

The generated code was tested, and each version (scaled and non-scaled) returned the path $S = 1, 0, 1, 0$ as their result. The main difference between these two implementations was in the $\gamma_t(i)$ calculations, as the non-scaled version computes $P(\mathcal{O}|\lambda)$ directly using the α-pass and uses that value in the calculations. The scaled version, on the other hand, computes the $\gamma_t(i, j)$ values to determine the value of $\gamma_t(i)$ (see Figures 7.21 and 7.22 for implementation details).

Again, to test the pathfinding functions fully, other models and observation sequences were used and the returned values were confirmed via external means. All of the tests returned the expected paths (except in cases where underflow was the expected result on the non-scaled code).

Training tests

As a very simple example test, consider an unknown model $\lambda = (\pi, A, B)$ with $N = M = 2$. Now given an observation sequence of length 150 where $\mathcal{O} = (0, 0, 0, \dots)$, train the model.

Intuitively, one can see that if this model is trained on the given observation sequence, the B matrix should end up with zero values for the probability of emitting an observation other than zero from any state. After training the model, the resulting B-matrix (from the generated implementation) was:

$$B = \begin{bmatrix} 1.0 & 0.0 \\ 1.0 & 0.0 \end{bmatrix}$$

This confirms the intuition that the only possible observations are $\mathcal{O}_t = 0$.

The majority of the training (or re-estimation) tests involved models that were too large to include a concise description of. However, the testing process remained the same as in the other sections. The models were trained using an observation sequence and on each iteration of re-estimation all of the calculated values were logged, then compared to their expected values (again, confirmed through external means).

8.2.2 Does it represent the design?

Since the generated source code has been shown to perform the task it was designed for, the last question is whether or not it truly represents the specified design. This may seem redundant or unnecessary (i.e. since the code works, who cares?), but it confirms whether or not the generator is able to produce a proper representation of a high-level design in a designated output language.

Testing this can be difficult depending on the scope of the code being generated as the most effective means (thus far) are to have human programmers read through the source and compare it with the design specifications. Each high-level concept in the DSL should be represented by an appropriate algorithm which makes sense from a programming perspective.

In the case of the HMM implementation, the generator's output was compared with a seperate source for handling HMMs in C as well as the design. It was concluded that the generated source represented the design, and by comparing it with a similar source code in the given output language the algorithms were appropriately chosen.

8.3 Known Issues and Special Cases

Refer to Section 8.2.1 for the example we will reference. There is a very obvious problem with using that example. When it comes to training a model, if the observation set ends with an event never before observed (as it does in that case), the re-estimated probability of observing that event will be zero. This is because of equation (3.7) which calculates the digamma values for each of the $t = 0, 1, \ldots, T-2$ observations. The last observation $(\mathcal{O}_{T-1})$ is used in the calculation of $\gamma_{T-2}(i, j)$ but will never be accounted for in the re-estimation of B (Equation 3.11).

However, this special case should not be a problem in practice as the majority of HMM uses deal with substantial amounts of data. Thus it is highly unlikely that this situation would occur unless the HMM were being used in an atypical manner.

Another known issue is that of rounding errors. Specifically, the current implementation uses C's “double” type to represent the probabilities. A simple fix would be to select a data type with higher precision, however, there will always be the possibility of rounding errors as the probability values become smaller. This is purely dependent on the output language's handling of very small numbers.

Finally there is an issue with pathfinding which comes from the math itself. While the HMM approach to pathfinding gives the most number of expected correct states, there may be some illegal state transitions (for example, where the state transition matrix has a zero probability of moving from state i to state j some segment of the returned path may be {. . . , i, j, . . . }). One solution to this would be to implement a method for checking whether or not a state transition is possible prior to finding the expected correct states, and ignoring any states that would be chosen from an illegal transition (instead selecting the next highest scoring valid state). Another solution would be to implement the dynamic programming approach (known as the Viterbi algorithm), which would then return the path with the overall highest probability (which may differ from the HMM approach solution).

Chapter 9

Conclusion

The HMM Generator created for this paper simplifies the production of differing HMM implementations. Each of the versions produced has been tested and shown to be working code as per the requirements.

This project was relatively small in scale, but works as a proof of concept that specialised learning algorithms can be produced fairly simply once a generator framework is in place. As such, increasing this project's scope to use a larger and more robust DSL (determined with the input of domain experts) could provide an excellent framework for creating a vast array of specialised learning algorithm implementations.

9.1 Lessons Learned

The method of using a domain specific language and code generator for the production of learning algorithms relies on a fundamental understanding of domain concepts (in this case, HMMs and their realizations as algorithms). The domain underlying the production of a DSL must be conceptually clear and unambiguous in order to produce an effective DSL.

Another lesson is that the abstraction process from concrete code to a DSL-driven code generator can be done incrementally. The results shown here are actually at a point inside the (as yet) incomplete abstraction process. While the internal DSL is better than the raw C code, it is still far from the high-level conception that a learning algorithm user would prefer. However, there is no doubt that the abstraction process can be continued, eventually leading to such a high-level conception.

A third lesson, which is entirely invisible from the result shown here, is that this incremental process is not monotone: it frequently happened that a step "sideways" was needed to achieve a proper step forward. For example, in the initial implementation of the DSL, mathematical operations were their own specific type of terrm. In order to clean up the syntax, that was changed and they were folded into the expressions of the DSL. This actually convoluted the code moreso, as it allowed for more erroneous inputs in favour of ease of writing. However, this convolution eventually led to the seperation of statements and expressions which removed a large number of possible errors and cleaned up the syntax of the DSL.

9.2 Next Steps

There are many ways to expand this project's scope and continue the development process.

The most obvious next step in the development process would be to fix the currently known issues with the HMM Generator. Modifying the current implementation by adding new terms to the AST (to match some less obvious patterns) would allow for more concise segments of program code and would also reduce the complexity of learning how to use the HMM Generator.

The HMM Generator can also be expanded in a number of ways which were outside the scope of this project, but would be useful in the future, such as:

- Implementing the Viterbi algorithm for pathfinding and including it as a design choice in the design AST.
- Expanding to cover a larger variety of models and systems that rely on Bayesian statistics.
- Adding the ability to output to multiple languages (not solely C).
- Optimization improvements in the final output code, as well as in the transitions between the internal AST and the output language AST(s).
- Allow for custom libraries to be included

Appendix A

AST Implementations

A.1 ASTDesign

```
module ASTDesign where
data Choices = Lang Language
             | Scale Scaling
             | MaxIt Int
    deriving (Eq,Show)

data Language = C
              | Haskell
    deriving (Eq,Show)

data Scaling = Scaled
             | NotScaled
    deriving (Eq,Show)

type Library = String
```

A.2 ASTInternal

```
module ASTInternal where

--AST module (internal)
import ASTDesign as D

data Program = Program [D.Choices] Name [GlobalDecl]
type Name = String

data GlobalDecl =  CDecl Name [ParamDecl]
                 | MACDecl Name String
                 | Funct FuncSig [Stmt]

data FuncSig = FDecl Type Name [ParamDecl]
data ParamDecl = PDecl Type Name

data Expr = Deref Expr Expr
          | NDArrDeref Expr [Expr]
          -- NDArrDeref Array [Indices]
          --Get val at location [i1,i2,i3,...,in]
          | Malloc Type Expr
            --Malloc Type Multiplier
            --(NOTE: Multiplier should not include SizeOf Type)
            --Type should never be a Ptr for it to work
          | Call Name [Expr] --Call named function
          | Length Expr --Only works on Arrays
          | Var Variable
          | Str String
          | Int Integer
          | Dbl Double
          | NDArr Array [Size]
          --NDimensionalArray Name [row_size,column_size,depth_size
             ,4_index_size,...,n-index_size]
          | Expr :* Expr
          | Expr :+ Expr
          | Expr :/ Expr
          | Expr :- Expr
          | Expr :> Expr
          | Expr :< Expr
          | Expr :== Expr
          | Expr :& Expr
          | Neg Expr
          | Unit
          | Sum Name Expr [(Expr, Expr)]
          -- Sum LoopVar Array [(ResultVar, MathFunct)] ?
          -- S2D Result Loop1 Array Loop2 Array MathFunct
          | Sum2D Expr Name Expr Name Expr Expr
          | InitAndSum Name Expr [(Expr,Expr)] [(Expr,Expr)]
          -- InitAndSum LoopVar Array [Sum pairs] [Init Pairs]
          -- First list is (ResultVar,MathFunct) just like in Sum
          -- Second list is inits, i.e. (Array,Value).
          | ProdSeq Name Expr [(Expr, Expr)]
          -- Works the same as sum, except ends up with result =: (
             result :* math)

data Stmt = CallProc Name [Expr]
          | Out Expr
          | Define Type Expr
          | Assign Expr Expr
            -- Define Var Value Type
          | For Expr Expr Expr [Stmt]
            -- For "Variable" start end [Do these things]
            -- ex. For (Var "i") (Int 0) (Int 10)
            --         [(Var "i") =: (Op (Var "i") :+ (Var "i"))]
            --   --> For (i = 0; i < 10; i++) {i = i+i;}
          | Return Expr
          | If Expr [Stmt] [Stmt] --If condition [do] [else do]
          | CondLoop Expr [Stmt]
          | EmptyStmt Expr
          | Block [Stmt]
          | UniformDist Expr Name
          -- UniformDist Array Index
          -- Init uniform values over an array (using a named index)
          -- ex. UniformDist array i
          -- --> For index 0 (Length array) [array !: index  =: 1/(
             Length array)]
          | NUniDist Expr Name Name
          -- NUniDist array outlvar inlvar
          -- Init non-uniform values over a 2D array
          -- ex. NUniDist (NDArr name [num_row,num_col]) i j
          -- --> For i 0 num_row $
          --         [ For j 0 num_col $
          --              [(array !: [i,j]) =: ((1+j)/ ((num_col :* (
             num_col+1)) :/ 2))]
          --         ]
          | Init Expr Expr Expr Expr
          -- Initialize (Array!:SomeIndExpr) Index Num_Elems Value
```

```
          -- Initialize first num_elems of array (using index in "
             SomeIndExpr") with value of "Expr"
          --ex: Init (a!:(i*n)) i n e = For i zero n [(a!:(i*n)) = e
             ]
          | Alloc Expr Type
          | MatrixBlockInit Expr [Expr] [(Expr,Expr)] Expr
          -- MBlockInit matrix [indices] [Ranges (start,end)] Value
          -- Indices are matched with ranges and setup from there
          | MatrixComps [Expr] [(Expr, Expr)] [Stmt]
          -- MatrixComps [indices] [Ranges] [Ops]
          -- Used to perform computations over matrix indices/ranges
data Type = IntType
          | VoidType
          | StrType
          | PtrType Type
          | HMMType
          | DblType
  deriving Eq


type Variable = String
type Array = String
type Size = Expr
-----------------------------------------------------------------
-- Make things prettier
int, dbl, void, str:: Type
int = IntType
dbl = DblType
void = VoidType
```

```
str = StrType

ptr :: Type -> Type
ptr = PtrType

v = Var

(%:) :: Name -> Type -> Stmt
x %: t = Define t $ v x
(=:) :: Expr -> Expr -> Stmt
x =: e = Assign x e
(=%:) :: Expr -> (Expr, Type) -> Stmt
x =%: (e,t) = Block [Define t x, Assign x e]
(@:) :: Expr -> Expr -> Expr
x @: e = Deref x e
(!:) :: Expr -> [Expr] -> Expr
x !: e = NDArrDeref x e

unit = EmptyStmt Unit

compute :: Expr -> Stmt
compute = EmptyStmt
```

A.3 ASTC

```
module ASTC where


--AST module
import ASTDesign as D


------------Program-------------
data Program = Program Name [D.Library] [Declaration] [Method]
    deriving Show
type Name = String


-------------Method-------------
data Method = Meth Declaration Stmt
            | NullMeth
    deriving Show


----------Declaration-----------
data Declaration = SDecl Name [Declaration]
                 | MDecl Type Name [Declaration]
                 | ADecl Type Variable
                 | MACDecl Name String
    deriving Show


type Variable = String
type Array = String
type Size = Expr
-----------Expression-----------
data Expr = Deref Expr Expr
          | ArrDeref Expr Expr
          | Call Name [Expr] --Call function of type
          | Malloc Type Expr --Type should never be a Ptr for it to
             work
          | SizeOf Type
          | Var Variable
          | Str String
          | Int Integer
          | Dbl Double
          | Mult Expr Expr
          | Add Expr Expr
          | Div Expr Expr
          | Sub Expr Expr
          | Greater Expr Expr
          | Less Expr Expr
          | Equal Expr Expr
          | And Expr Expr
          | Neg Expr
          | Arr Array Size
          | NDArr Array [Size]
          | Unit

    deriving Show
-----------Statements-----------
data Stmt = CallProc Name [Expr]
          | Out Expr
          | Define Type Expr
          | Assign Expr Expr
          | For Expr Expr Expr [Stmt] --For "Variable" start end [Do
              these things] ex. For (Var "i") (Int 0) (Int 10) [(Var
              "i") =: (Op (Var "i") :+ (Var "i"))] --> For (i = 0; i
              < 10; i++) {i = i+i;}
          | Return Expr
          | If Expr [Stmt] [Stmt] --If condition [do] [else do]
          | CondLoop Expr [Stmt]
          | Block [Stmt]

    deriving Show

--------------Type--------------
data Type = IntType
          | VoidType
          | StrType
          | PtrType Type
          | HMMType
          | DblType
          | NullType
    deriving Show
```

Appendix B

Generated C Code

Note: This is the scaled version of the C code as produced by the HMM Generator.

B.1 Header File

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<limits.h>
#define MAX_ITERS 1000
typedef struct {
    int n;
    int m;
    double* initProbs;
    double* transMatrix;
    double* obsMatrix;
} HMM;
double getTMVal(int state,int state2,HMM* hmm);
double getOMVal(int state,int event,HMM* hmm);
void setTMVal(double val,int row,int col,HMM* hmm);
void setOMVal(double val,int row,int col,HMM* hmm);
void init_rand(int n,int m,HMM* hmm);
void init_pi(int n,double* n_pi);
void init_a(int n,double* a);
void init_b(int n,int m,double* b);
double* alpha_Array(HMM* hmm,int* obs,int num_obs,double* scaling);
double* beta_Array(HMM* hmm,int* obs,int num_obs,double* scaling);
double alpha_pass(HMM* hmm,double* a_A,int num_obs);
double beta_pass(HMM* hmm,double* b_A,int* obs,int num_obs);
double* gamma_array(HMM* hmm,int* obs,int num_obs,double* a_A,double
   * b_A,double* dg_A);
int* pathFindHMM(HMM* hmm,int* obs,int num_obs,double* g_A);
void est_pi(HMM* hmm,double* g_A);
void est_A(HMM* hmm,int num_obs,double* g_A,double* dg_A);
void est_B(HMM* hmm,int* obs,int num_obs,double* g_A);
void re_estimate(HMM* hmm,int* obs,int num_obs);
```

B.2 C Source File

```
#include "HMMProg.h"
double getTMVal(int state,int state2,HMM* hmm) {
    int n;
    n = hmm->n;
    return hmm->transMatrix[(state*n+state2)];
}
double getOMVal(int state,int event,HMM* hmm) {
    int m;
    m = hmm->m;
    return hmm->obsMatrix[(state*m+event)];
}
void setTMVal(double val,int row,int col,HMM* hmm) {
    int n;
    n = hmm->n;
    hmm->transMatrix[(row*n+col)] = val;
}
void setOMVal(double val,int row,int col,HMM* hmm) {
    int m;
    m = hmm->m;
    hmm->obsMatrix[(row*m+col)] = val;
}
void init_rand(int n,int m,HMM* hmm) {
    double* a;
    double* b;
    double* n_pi;
    n_pi = (double *)malloc(sizeof(double)*n);
    a = (double *)malloc(sizeof(double)*n*n);
    b = (double *)malloc(sizeof(double)*n*m);
    init_pi(n,n_pi);
    init_a(n,a);
    init_b(n,m,b);
    hmm->n = n;
    hmm->m = m;
    hmm->initProbs = n_pi;
    hmm->transMatrix = a;
    hmm->obsMatrix = b;
}
```

```
void init_pi(int n,double* n_pi) {
    int i0;
    for (i0=0; i0<n; i0++) {
        n_pi[i0] = (1.0/n);
    }
}
void init_a(int n,double* a) {
    int i1;
    int j;
    for (i1=0; i1<n; i1++) {
        int j;
        for (j=0; j<n; j++) {
            a[(i1*n+j)] = ((1.0+j)/(n*(n+1.0)/2.0));
        }
    }
}
void init_b(int n,int m,double* b) {
    int i2;
    int j0;
    for (i2=0; i2<n; i2++) {
        int j0;
        for (j0=0; j0<m; j0++) {
            b[(i2*m+j0)] = ((1.0+j0)/(m*(m+1.0)/2.0));
        }
    }
}
double* alpha_Array(HMM* hmm,int* obs,int num_obs,double* scaling) {
    double* a_A;
    int i3;
    int t;
    int n;
    n = hmm->n;
    a_A = (double *)malloc(sizeof(double)*num_obs*n);
    scaling[0] = 0;
    for (i3=0; i3<n; i3++) {
        a_A[i3] = hmm->initProbs[i3]*getOMVal(i3,obs[0],hmm);
        scaling[0] = (scaling[0]+a_A[i3]);
    }
    scaling[0] = (1.0/scaling[0]);
    for (i3=0; i3<n; i3++) {
        a_A[i3] = a_A[i3]*scaling[0];
    }
    for (t=1; t<num_obs; t++) {
        int i3;
        scaling[t] = 0;
        for (i3=0; i3<n; i3++) {
            int j00;
            a_A[(t*n+i3)] = 0;
            for (j00=0; j00<n; j00++) {
                a_A[(t*n+i3)] = (a_A[(t*n+i3)]+a_A[((t-1)*n+j00)]*
                    getTMVal(j00,i3,hmm));
            }
            a_A[(t*n+i3)] = a_A[(t*n+i3)]*getOMVal(i3,obs[t],hmm);
            scaling[t] = (scaling[t]+a_A[(t*n+i3)]);
        }
        scaling[t] = (1.0/scaling[t]);
        for (i3=0; i3<n; i3++) {
            a_A[(t*n+i3)] = a_A[(t*n+i3)]*scaling[t];
        }
    }
    return a_A;
}
double* beta_Array(HMM* hmm,int* obs,int num_obs,double* scaling) {
    double* b_A;
    int i4;
    int t0;
    int n;
    n = hmm->n;
    b_A = (double *)malloc(sizeof(double)*num_obs*n);
    for (i4=0; i4<n; i4++) {
        b_A[((num_obs-1)*n+i4)] = scaling[(num_obs-1)];
    }
    for (t0=(num_obs-2); t0>=0; t0--) {
        int i4;
        for (i4=0; i4<n; i4++) {
            int j000;
            b_A[(t0*n+i4)] = 0;
            for (j000=0; j000<n; j000++) {
                b_A[(t0*n+i4)] = (b_A[(t0*n+i4)]+b_A[((t0+1)*n+j000)
                   ]*getTMVal(i4,j000,hmm)*getOMVal(j000,obs[(t0+1)
                   ],hmm));
            }
            b_A[(t0*n+i4)] = scaling[t0]*b_A[(t0*n+i4)];
        }
    }
    return b_A;
}
```

```
double alpha_pass(HMM* hmm,double* a_A,int num_obs) {
    int i5;
    double ret_val;
    int n;
    n = hmm->n;
    ret_val = 0;
    for (i5=0; i5<n; i5++) {
        ret_val = (ret_val+a_A[((num_obs-1)*n+i5)]);
    }
    return ret_val;
}
double beta_pass(HMM* hmm,double* b_A,int* obs,int num_obs) {
    int i6;
    double ret_val;
    ret_val = 0;
    for (i6=0; i6<hmm->n; i6++) {
        ret_val = (ret_val+b_A[i6]*getOMVal(i6,obs[0],hmm)*hmm->
            initProbs[i6]);
    }
    return ret_val;
}
double* gamma_array(HMM* hmm,int* obs,int num_obs,double* a_A,double
   * b_A,double* dg_A) {
    double* g_A;
    int t00;
    int n;
    n = hmm->n;
    g_A = (double *)malloc(sizeof(double)*num_obs*n);
    double temp;
    for (t00=0; t00<(num_obs-1); t00++) {
        int i7;
        temp = 0;
        for (i7=0; i7<n; i7++) {
            int j0000;
            for (j0000=0; j0000<n; j0000++) {
                temp = (temp+a_A[(t00*n+i7)]*getTMVal(i7,j0000,hmm)*
                   getOMVal(j0000,obs[(t00+1)],hmm)*b_A[((t00+1)*n+
                   j0000)]);
            }
        }
        for (i7=0; i7<n; i7++) {
            int j0000;
            g_A[(t00*n+i7)] = 0;
            for (j0000=0; j0000<n; j0000++) {
                dg_A[(t00*n*n+(i7*n+j0000))] = (a_A[(t00*n+i7)]*
                   getTMVal(i7,j0000,hmm)*getOMVal(j0000,obs[(t00+1)
                   ],hmm)*b_A[((t00+1)*n+j0000)]/temp);
                g_A[(t00*n+i7)] = (g_A[(t00*n+i7)]+dg_A[(t00*n*n+(i7
                   *n+j0000))]);
            }
        }
    }
    return g_A;
}
int* pathFindHMM(HMM* hmm,int* obs,int num_obs,double* g_A) {
    int* ret_array;
    int t000;
    int n;
    n = hmm->n;
    double max;
    ret_array = (int *)malloc(sizeof(int)*num_obs);
    for (t000=0; t000<num_obs; t000++) {
        int i8;
        max = 0;
        for (i8=0; i8<n; i8++) {
            if (g_A[(t000*n+i8)]>max) {
                max = g_A[(t000*n+i8)];
                ret_array[t000] = i8;
            }
        }
    }
    return ret_array;
}
void est_pi(HMM* hmm,double* g_A) {
    int i9;
    int n;
    n = hmm->n;
    for (i9=0; i9<n; i9++) {
        hmm->initProbs[i9] = g_A[i9];
    }
}
void est_A(HMM* hmm,int num_obs,double* g_A,double* dg_A) {
    int i10;
    double num;
    double denom;
    double temp;
    int n;
    n = hmm->n;
    for (i10=0; i10<n; i10++) {
        int j00000;
        for (j00000=0; j00000<n; j00000++) {
            int t0000;
            num = 0;
            denom = 0;
            for (t0000=0; t0000<(num_obs-1); t0000++) {
                num = (num+dg_A[(t0000*n*n+(i10*n+j00000))]);
                denom = (denom+g_A[(t0000*n+i10)]);
            }
            temp = (num/denom);
            setTMVal(temp,i10,j00000,hmm);
        }
    }
}
void est_B(HMM* hmm,int* obs,int num_obs,double* g_A) {
    int i11;
    int n;
    n = hmm->n;
    int m;
    m = hmm->m;
    double num;
    double denom;
    double temp;
    for (i11=0; i11<n; i11++) {
        int j000000;
        for (j000000=0; j000000<m; j000000++) {
            int t00000;
            num = 0;
            denom = 0;
            for (t00000=0; t00000<(num_obs-1); t00000++) {
                if (obs[t00000]==j000000) {
                    num = (num+g_A[(t00000*n+i11)]);
                }
                denom = (denom+g_A[(t00000*n+i11)]);
            }
            temp = (num/denom);
            setOMVal(temp,i11,j000000,hmm);
        }
    }
}
void re_estimate(HMM* hmm,int* obs,int num_obs) {
    double* dg_A;
    double* scaling;
    int n;
    int iters;
    iters = 0;
    double logProb;
    double oldLogProb;
    scaling = (double *)malloc(sizeof(double)*num_obs);
    double* a_A;
    double* b_A;
    double* g_A;
    n = hmm->n;
    logProb = (INT_MIN+1);
    oldLogProb = INT_MIN;
    dg_A = (double *)malloc(sizeof(double)*num_obs*n*n);
    while (iters<MAX_ITERS && logProb>oldLogProb) {
        int i12;
        oldLogProb = logProb;
        a_A = alpha_Array(hmm,obs,num_obs,scaling);
        b_A = beta_Array(hmm,obs,num_obs,scaling);
        g_A = gamma_array(hmm,obs,num_obs,a_A,b_A,dg_A);
        est_pi(hmm,g_A);
        est_A(hmm,num_obs,g_A,dg_A);
        est_B(hmm,obs,num_obs,g_A);
        logProb = 0;
        for (i12=0; i12<num_obs; i12++) {
            logProb = (logProb+log(scaling[i12]));
        }
        logProb = -logProb;
        iters = (iters+1);
    }
}
```

Appendix C

Large Test Case Results

Results from automated test script were too long (260 pages) to be included.

Over 7300 tests were performed throughout re-estimation processes on a large scale example, and the returned values were tested against the expected results.

All tests passed.

Bibliography

Baker, J. (1975). The DRAGON system–An overview. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, **23**(1), 24–29.

Baum, L. E. and Petrie, T. (1966). Statistical Inference for Probabilistic Functions of Finite State Markov Chains. *The Annals of Mathematical Statistics*, **37**(6), 1554–1563.

Beyak, L. (2011). SAGA: A Story Scripting Tool for Video Game Development.

Bolstad, W. (2007). *Introduction to Bayesian Statistics*. Wiley, second edition.

Costabile, J. (2012). GOOL: A Generic OO Language.

Curutan, B. (2013). CPCG: A Cross-Paradigm Code Generator.

Fowler, M. (2010). *Domain Specific Languages*. Addison-Wesley Professional, 1st edition.

Mur, R. A. (2006.). Automatic Inductive Programming. `http://www.evannai.inf.uc3m.es/et/icml06/aiptutorial.htm`. [ICML 2006 Tutorial.].

Stamp, M. (2004). A Revealing Introduction to Hidden Markov Models.