Keyhole State Space Construction with Applications to User Modeling

by

Peter John Gorniak

B.Sc., University of British Columbia, 1998

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

We accept this thesis as conforming
to the required standard

The University of British Columbia

June 2000

Abstract

Most models that explain observations in time depend on a structured state space as a basis for their modeling. We present methods to derive such a state space and its dynamics automatically from the observations, without any knowledge of their meaning or source. First, we build an explicit state space from an observation history in an off-line fashion (OFESI) by starting with the space induced by the observations and splitting a state when new substates significantly improve the information content of the state's action distribution. We form these new substates by grouping fixed length histories leading up to the state. We apply our algorithm to the user modeling case and show that we can automatically build a meaningful stochastic dynamic model of application use. Second, we discuss prediction of the next observation and show that an approach based on on-line implicit state identification (ONISI) from observed history outperforms other prediction methods. Again, we are interested in user modeling, where we can predict future user actions better than another current algorithm. Both algorithms work without knowledge or modification of the application in use. Third, we apply our explicit state identification algorithm to the problem of state identification for Hidden Markov Models (SIHMM). Taking into account both observation and transition probabilities we learn structures for Hidden Markov Models in a few iterations.

Contents

List of Tables

List of Figures

Acknowledgements

I would like to thank my supervisor, David Poole, for his ideas, comments and guidance. Craig Boutilier and Holger Hoos also provided valuable input. Jesse Hoey enlightened me during long discussions. Cydney listened and read with interest and great patience.

Peter John Gorniak

The University of British Columbia
June 2000

To Cydney

Chapter 1

Introduction

1.1 The General Problem

We commonly observe a signal over time and draw inferences about the process that produces it. For example, we listen to noises and understand speech, we attend a dance performance and argue about its meaning, or we watch my mother use a computer and finally understand what she doesn't understand. One of the goals of Artificial Intelligence is to mimic these abilities: to have a computer understand speech, criticize dance and help our mothers write Christmas letters. A number of techniques exist to model temporal processes computationally. For example, Hidden Markov Models attempt to explain human speech, Markov Decision Processes map state information and observations into actions, and a Bayesian Belief Network may model my mother's confusion. Largely, the basic formulations of these methods assume that most things are known about the model ahead of time. We usually need a good estimate of the structure of the model, be it the nodes of a Bayesian network or the states and connectivities of a Markov model.

In the work presented here, we concern ourselves with inferring the state

structure and dynamics for stochastic models from a sequence of observations. This problem is akin to a child learning speech, a dance critic learning to interpret a performance, and my learning that my mother works with a vastly different model of a computer than I do. In most of our discussion here we will focus on the last problem, that of my mother using a computer. This problem exhibits several features that sparked our interest:

- With the expanding use of computers and the internet, intelligent interfaces and user modeling are busy areas of research, but few approaches attempt to learn the model's structure and dynamics.
- Coarse observation data can be collected automatically without modifying the application in use.
- The system comprising the computer and its user is a hybrid system, meaning that on the one hand modeling is made harder through the spontaneity of human input, but on the other hand human behaviour is filtered and channeled through the computer's interface, and thus easier to observer and model.

We next discuss the user modeling problem in detail.

1.2 The User Modeling Problem

My mother, being very interested in how her son spends his time, decided to learn about search algorithms. To help in her quest, I provided her with a small Java application that demonstrates such algorithms. Watching and helping her use it, I observed a number of shortcomings in the application's interface:

- When she was unclear about how to specify search parameters, the help system showed her information about the colours being used in the interface.
- I had to remind her a number of times about how to reset a search and switch search algorithms.
- The interface did not make it clear when a search was reset, so she pressed the reset button more often than needed.

All these frustrations my mother faced during her venture into graph searching are common shortcomings of direct manipulation interfaces, be it due to design flaws or need for adaptivity. It would be nice if the interface could either be analyzed and redesigned in terms of how my mother uses it, or if it could recognize my mother's intentions and confusion, in short, be a more intelligent interface. I know what she wants to do because I have a good idea of her capabilities, goals and preferences from having observed her using a computer before, and from knowing about the task at hand and how the interface represents it. For an intelligent system to draw the same conclusions, it also needs to observe my mother using an application interface and infer a model of the task and how she means to solve it. For the interface to be intelligent, it should then ask this model for my mother's current context and adapt its display and functionality to help her. It may also relay the model to the application designer, helping him or her identify weaknesses in the interface design.

Specifically, an intelligent interface should

- Ask its model for my mother's current context and adjust its help system to display information about running a search instead of explaining interface colours.

- Identify the recurring pattern of resetting the search and switching to another search algorithm, then adapt the interface to make this process obvious and faster.

- Show the application designer that users tend to reset the search when it is already reset, and inform him or her about this interface design flaw.

User models of varying shapes and sizes have been built before. Artificial Intelligence supplies a vast set of tools to build such models. For example, Horvitz, Breese, Heckerman, Hovel and Rommelse (1998) and Albrecht, Zukerman, Nicholson and Bud (1997) use Bayesian networks to infer a user's intentions from their behaviour, while we previously employed neural networks to capture user's preferences (Gorniak 1998). Coming back to the search application, it is certainly possible to select one of these approaches, fit a model structure to the application (probably through user studies,) modify the application by adding the model implementation, and possibly write separate code to collect data to estimate the model's parameters. Only then can an intelligent interface be created on top of the user model. All this constitutes a fair amount of work in addition to application and user interface design, but it doesn't stop there. Tomorrow, my mother will be bored of search algorithms and move on to constraint satisfaction problems. The whole process of building a model for an intelligent interface will have to be repeated for the constraint satisfaction application. While she is busy with that, I will pour over the behavioural model of her using the search application, modify the search application based on the information I have gained, and thus invalidate the old model through application maintenance. The model now needs to be refitted and reestimated, adding a large overhead to the maintenance cost.

In this work, we address these problems by investigating how much knowledge can be extracted from a user's interaction with an application. We assume no prior information about the application's purpose or structure and require no modifications to the application (somewhat in the spirit of (Lieberman 1998), who gives a survey of techniques to combine intelligent agents with existing interfaces.) We hypothesize that enough knowledge can be extracted to yield a detailed model of the application and its user. This model can then serve both as a knowledge source or starting point for other algorithms as well as provide a common basis for various methods. Most importantly, we set out to construct this model without modifying the application and without running explicit user trials. In fact, the system we present works as a wrapper to the Java runtime environment. In principle, the system can model the use of any Java application without customization of either the application or our system. We cannot hope that our restricted modeling approach will by itself provide a suitable model for all needs, but we believe that it can always provide a good starting point for further modeling efforts.

1.3 Comparison with other Approaches

Our approach differs greatly from informed modeling strategies such as (Horvitz et al. 1998). In informed modeling strategies, knowledge about the application's purpose and structure are incorporated into the model. For example, the model may contain a number of goals that a user may pursue. Actions the user takes can be interpreted according to their purpose and the goals to which they lead. As a result of incorporating application specific knowledge, Horvitz et al. (1998) can guess from the user's actions that he or she is likely trying to define a custom chart in Excel, whereas we can only say that user and application are currently in a state

which we represent through a set of action sequences leading to it and a probability distribution over actions and transitions occurring in it. In addition, we will likely be unable to model the application and the user to the level of detail needed for a sophisticated intelligent interface. An informed model likely lets one take into account the features needed to provide a sufficiently detailed model. However, we are only aware of hand-crafted informed models, where the hand-crafting is a work-intensive process. We cannot provide a solution that completely automates that modeling process to any level of detail, but we demonstrate that much of the structure of application use can indeed be modeled automatically. Our goal is to provide a good initial analysis of user and application and by doing so allow the designer of an intelligent interface to begin the modeling process with a reasonably detailed initial model rather than from scratch.

Other application independent user modeling strategies exist. Some of them build no application model at all, and thus do not provide any automatic analysis of the application. They perform worse in cases where such application analysis boosts performance, such as future action prediction (Davison and Hirsh 1998). Others stop early on in their analysis and subsequently rely on application specific knowledge (Encarnacao 1997). We are not aware of other work that attempts to identify the current state of application and user without any knowledge or modification of the application. Within the model building area, our research is related to the next action prediction work for web precaching by Zukerman, Albrecht and Nicholson (1999). They work with combinations of simple Markov models based on request counts, and do not identify the system's state in more detail. Thus, their model is similar to the base model we introduce in chapter 2. Our goal is also akin to some work in the areas of data mining and specifically clustering. For example,

Cadez, Heckerman, Meek, Smyth and White (2000) cluster users based on their web page request patterns for visualization purposes, but they do not build a detailed stochastic dynamic model like ours.

1.4 Assumptions and Choice of Model

We make two primary assumptions. First, We assume that we do not know the application's structure or purpose. Second, we assume that we can and have observed the user using an application. Specifically, we assume that we can observe all visible elements of the user interface, as well as user actions pertaining to these elements (button clicks, menu activations, etc.)

As for a choice of modeling approach, we require that our model be human readable to allow application designers to examine the structure of their applications and use this structural information as a basis for intelligent interfaces. This excludes opaque solutions like neural networks. We also require that our model naturally capture the temporal nature of the process being observed, because we want to reveal and take advantage of the dynamics exhibited in application use. This excludes approaches that tend towards the static, like Bayesian networks. Finally, to stay as general as our assumptions dictate, we must be able to infer structure and parameter settings of our chosen model from observing the application's interface alone, without any knowledge as to the purpose of the events we observe. This restriction limits the modeling possibilities, but it makes model estimation easily portable to new applications, and, in fact, allows for an implementation that does not modify the application to be modeled. The restriction also precludes most informed modeling strategies, like plan recognizers that work from possible goals, or models that guess the user's mental state from the semantics of the actions the user

carries out.

Therefore, we elect to construct dynamic, stochastic Markov models of application use. Our models consist of **states** that produce **observations**. The models also contain probability distributions that govern the **transitions** amongst states and the production of observations. At a given point in time a model is in exactly one state, and the model's future (in terms of observation and transition probabilities) depends only on that state. The Markov assumption captures this fact by stating that the future of the model is independent of its past given its current state. Using one of our models implies this assumption through the nature of the model, and we may thus ask whether the model is correct in that respect by looking for times when the future of the model can be predicted from the past better than just from its current state. These models fulfill our criteria of being human readable because their structure provides an intuitive overview over the dynamic development of the system. The models' states can easily be correlated with landmarks along a user's path through the application, and the transition and observation probabilities stochastically reveal patterns of user behaviour. With our limiting assumptions, we can still easily infer a base model and use the Markov assumption for refinement. Most of this work discusses how to introduce good states into our models based on the observed interactions of the user with the application interface. A state can loosely be seen as capturing two things: the complete configuration of the application and the user's situation, as far as they can be estimated from the observations. The **observations** that are the basis for our model fall into two categories: **interface configurations**, which comprise information about all visible elements of the user interface at a point in time, and **actions**, which identify the user's interactions with the interface elements. We will present various methods for inferring

and representing states based upon observations, each one good for a slightly different problem. Due to our assumptions and our choice of model we refer to our approach as a Keyhole State Space Construction approach. "Keyhole" refers to the fact that we perceive the user through very limited channels and thus have to work from only a small fraction of all the things that could be observed. "State Space Construction" refers to the fact that we perform structure learning for our choice of dynamic, stochastic Markov model.

Under our assumptions, our observations induce a base model structure: the base model states correspond directly to observed interface states, whereas action frequencies induce transition probabilities. All our model refinement strategies improve this base model by analysing the interaction history. We check whether states fulfill the Markov assumption, that is, whether history cannot be used to predict what happens in a state. If the Markov assumption holds, and we can use the base model's state and probabilities as a model of interaction. If the Markov assumption does not hold, we refine the model by attaching one or more history segments to a state, supplying new k-Markov states. These new state will be more Markovian and thus have greater predictive power than the original state in the base model. We pursue this approach explicitly to analyse and model applications and user behaviour in chapter 3 and implicitly to predict the next action in the current situation in chapter 4.

1.5 Notation

Our models are closely related to Hidden Markov Models (HMMs), so we take our notation from the one adopted for HMMs by (Rabiner 1989). However, note that in chapters 3 and 4 our goals and methods differ from those commonly associated with

HMMs (e.g. time sequence classification.) In chapter 5 we show how the algorithms derived in the earlier chapters can be applied to HMMs. In particular, the standard HMM formulation in (Rabiner 1989) assumes that the number of states and some structural information for the HMM are given a priori. In chapter 5 we apply our explicit state identification method to determine the number of states needed and to estimate the HMM's structure.

A model contains N states denoted by $\{S_1, S_2 \ldots S_N\}$. The state the model is in at time t is denoted by q_t. State i carries with it a probability distribution over all states $A = \{a_{ij}\}$ where a_{ij} is the probability of the model transitioning from state i to state j. Note that these transition probabilities are independent of the actions observed in state i. We treat observed actions as the observation symbols for HMMs and call the M possible actions $V = \{v_1, v_2 \ldots v_M\}$. Each state also carries a probability distribution over these actions, $B = \{b_j(k)\}$, where $b_j(k)$ is the probability of observing action v_k at a time when the model is in state q_j. The action actually observed at time t is called O_t. We assume that we have observed T timesteps all in all. In addition to these HMM elements, we associate each state with one interface configuration denoted $C(S_i)$ for state S_i and index the L observed interface configurations as $\{C_i\}_{i=1}^{L}$. Finally, we denote by c_t the interface configuration observed at time t. Multiple states may have the same interface configuration, so that the observed interface configurations form a projection of the state space. According to our assumptions, we have access to a sequence of T observation pairs $H = \{(c_t, O_t)\}_{t=1}^{T}$. We call its members $h_1 \ldots h_t$. Throughout, we will assume that we deal with only one observation sequence, but all algorithms presented easily transfer to multiple observation sequences.

1.6 State Space Inference Algorithms

We present three variations of a state space inference algorithm, designed with different goals:

OFESI, Off-Line Explicit State Identification, explicitly identifies states of user and application. OFESI refines model states by checking whether they fulfill the Markov assumption. If a state does not (up to a threshold), OFESI replaces it with new substates that are more Markovian. Applying OFESI results in a detailed model of how one or many people are using the application. Such a model can help application designers analyse their applications and augment them with intelligent extensions. For example, it is easy to learn about user behaviour and find unexpected consequences of a design decision. Or, in building an intelligent help system, the designer can have the help system query our model for its current state, as well as our prediction for the user's future actions and then tailor the help to that scenario. We first introduced OFESI in (Gorniak and Poole 2000b).

ONISI, On-Line Implicit State Identification, predicts the future actions of application users. It observes the current application interface state and assigns probabilities to all possible actions within that state. It arrives at these probabilities by estimating how much observed history supports and action in the current context. Its estimates can be seen as an approximate check of the Markov assumption for the current context. In contrast to OFESI, ONISI is fast enough to make prediction on-line, but the probabilities it assigns to actions are only valid in the current context. ONISI algorithm performs better than other current next action prediction algorithms. We first introduced

ONISI in (Gorniak and Poole 2000a).

SIHMM, State Identification for Hidden Markov Models, addresses the problem of finding appropriate structures for HMMs. In a nutshell, SIHMM is a version of OFESI's state splitting algorithm adapted to HMMs. The basic Hidden Markov Model algorithms require the number of states in the model to be specified ahead of time. Also, HMMs often converge to local maxima if no information about the state transition and observation probabilities is included (Rabiner 1989). We address both problems by building our models up from single state models and using OFESI's Markov assumption check for model states to introduce more Markovian states in appropriate places. SIHMM can retrieve the model structure from observations produced by known Hidden Markov Models.

ONISI can make the necessary predictions to know that my mother wants to switch search algorithms at a certain point in time. An intelligent help system for the search application could easily reset the problem for her after a search is completed and display the choice of search algorithms. OFESI, on the other hand, supplies a model that indicates that she tends to press the reset button when the search is already reset, pointing to an interface design flaw. Once a designer has annotated and extended this model to better capture the user's state, the help system could query it for the current context, and deduce that when my mother is running automated searches she does not want to be informed about the use of colours in the interface, but more likely about the parameters for the searches. Throughout this work, we will present results from a search application modeled through ONISI and OFESI without any modifications to the application and show that these algorithms provide a solid basis for modeling application use.

1.7 Organization

We first describe the Java program observing the target applications as well as the applications used to evaluate our algorithms in chapter 2. Chapters 4, 3, 5 respectively describe OFESI, ONISI and SIHMM in detail and present an evaluation for each algorithm. Chapter 6 summarizes and points to future work.

Chapter 2

Implementation and Experimental Setup

2.1 Java Implementation

Java's reflective capabilities and dynamic loading strategy make the language a prime candidate for an application independent approach (JDK 1998). It allows not only inspection of a structure of known visual components, but it can also inspect unknown components for state information. Java and JavaBeans introduced standard naming conventions for object methods. For example, *isVisible()* returns the visibility status of visual components, whereas *getEnabled()* returns whether they are currently useable. Components derived from standard Abstract Window Toolkit components inherit these methods automatically, and other components should define them. Java's Reflection mechanism, on the other hand, allows one to check whether a given object includes one of these state-revealing methods, and lets one call this method without knowing the object's class. Finally, Java's dynamic loading of classes rids the developer of needing to link with or even know about

classes that will be present at runtime. Using these tools, one can establish the user interface state of an application built using Java at runtime by dynamically linking into its code, examining the methods available in its objects and calling the methods relevant to the interface state. This process requires no modification of the targeted application at all.

The system used for the experiments presented below runs as a wrapper to a Java application. Before it starts the application, it hooks itself into the application's event queue and thus sees all event activity within the Java Abstract Window Toolkit and components derived from it. It intercepts each such event that it considers an action (such as a button being pressed or a window closed) and records the observed configuration of the application's interface before and after the event occurs. In this way, this system records a history consisting of actions and interface configurations as a person uses an unmodified application.

2.2 Modeled Applications and Data Collected

The applications[1] under consideration here are educational AI applications. They were written to help undergraduate university students learn concepts in Artificial Intelligence. One application familiarizes the student with search problems and algorithms, the second deals with constraint satisfaction problems and the third demonstrates backpropagation neural network learning. In each, the student has the option to either load an example problem or to create his or her own problem by drawing a graph. He or she can then switch to a problem solution mode and step through the various algorithms at different levels of detail.

We collected data from the students of a third year undergraduate Artificial

[1]The applications can be found at *http://www.cs.ubc.ca/labs/lci/CIspace/*.

Intelligence course. The students used the applications to solve homework problems. When running one of the applications, a student would be asked whether he or she agreed to let us record the session. If he or she agreed, the application would continue to run normally, but our system would write the interface configuration and action trace to a file. For their convenience, the students also had the choice of using an applet version of the applications. We could not record the applet. Most of our data stems from the search application, with which we recorded about 2200 actions of 9 users. The other applications yielded less data (1100 actions for the neural net and 200 actions for the constraint satisfaction application.) Most of the assignment questions referred to supplied example problems, so the students tended to explore the problem creation facilities of the applications less than their solving functionality. The following discussion and results focus mainly on the application for search algorithms as shown in Figure 2.1.

The screenshot shows the search application in problem solution mode, and one can see buttons to step through the current problem with a algorithm, to show the result of using an algorithm on the current problem, and to switch to problem creation mode. Currently, the interface is in its Stepping state, that is, the user has pressed the Step button at least once. Without knowing the semantics of the application, we can identify this as a unique interface configuration because the Problem Solution button is disabled, and the Show Result button is active, amongst other features. We will see this interface configuration represented through a state in the base model, and later analyse it with the goal of producing states of better predictive power.

In all our model building efforts, we treated all data collected from one application as if coming from one user. We usually only recorded one or two sessions

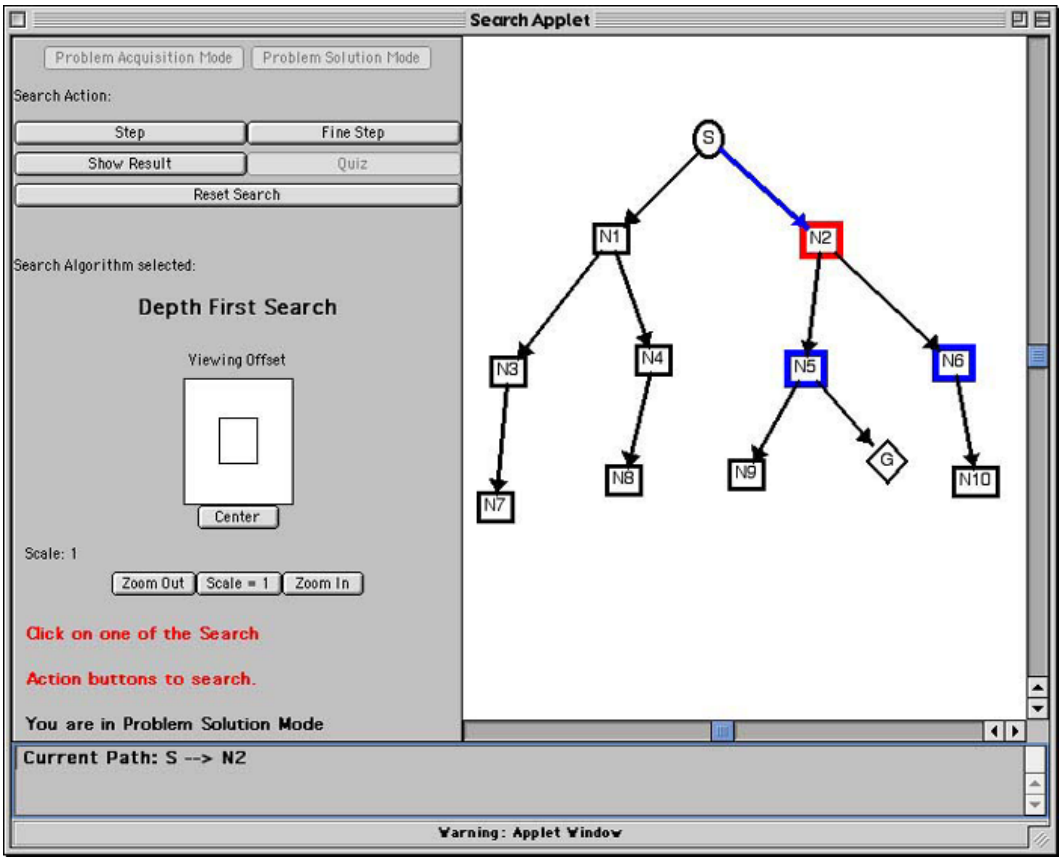

Figure 2.1: The Modeled Search Application Interface

per user, so there was not enough data to make statistically significant claims on a per-user basis. We believe pooling the data in this way is valid, especially because students used the problem creation facilities of the application very little. The problem solution mode of the applications is simple enough that users do not differ greatly in how they use it, and there is no significant learning that occurs on the user's side. In addition, our algorithms are relatively insensitive to mixing data in this way: ONISI will simply ignore data that doesn't match the current context, whereas OFESI will only split off states that represent features exhibited in a significant portion of that data (according to its parameter settings,) so a model from many users will simply tend to only represent those features that are common amongst users.

We cannot generally answer the question of how much data is needed to make our models significant. ONISI will recognize behavioural patterns if they only occurred once before, but because it works with exact matches it takes some time to establish a library of patterns. How many such patterns are needed depends on the application. However, our experiments in chapter 4 show that ONISI does tend to start predicting correctly quickly. OFESI's parameters can be adjusted to respond to as much detail as needed, but with little data such detail will likely mostly consist of noise. With parameter settings that filter out the noise, the question becomes whether OFESI's decision not to split a state is due to lack of data, or due to the fact that the state should not be split. One answer lies in examining why OFESI discards candidate states: A state that is discarded due to OFESI's information gain threshold likely should be discarded, whereas a state discarded due to OFESI's data amount threshold likely indicates lack of data.

2.3 Base State Space

The sequence H of interface configuration and action pairs we observe induces a natural basic Markov model that contains one state per interface configuration occurring in the history. Note that this one-to-one mapping between states and interface configurations only holds for the base model. When we refine our models, many state can correspond to one interface configuration. First, we get a state sequence corresponding to the observation sequence by setting $q_t = S_i$ such that $c_t = C(q_t)$. The observation probabilities for state i are given by

$$b_i(k) = \frac{\sum_{t=1}^{T} p(t,k) r(t,i)}{\sum_{t=1}^{T} r(t,i)} \tag{2.1}$$

where

$$p(t,k) = \begin{cases} 1 & \text{if } O_t = v_k \\ 0 & \text{otherwise} \end{cases} \tag{2.2}$$

and

$$r(t,i) = \begin{cases} 1 & \text{if } q_t = S_i \\ 0 & \text{otherwise} \end{cases} \tag{2.3}$$

The transition probabilities for state i, on the other hand, can be calculated using

$$a_{ij} = \frac{\sum_{t=1}^{T-1} r(t,i) r(t+1,j)}{\sum_{t=1}^{T-1} r(t,i)} \tag{2.4}$$

Figure 2.2 shows the base model of the search application. The figure represents the space exactly as recorded, except for that we have given the states meaningful names. Reflecting the division of the application into two modes, problem creation and problem solution mode, the graph exhibits two distinct components. The right hand component corresponds to problem solution mode, whereas the left

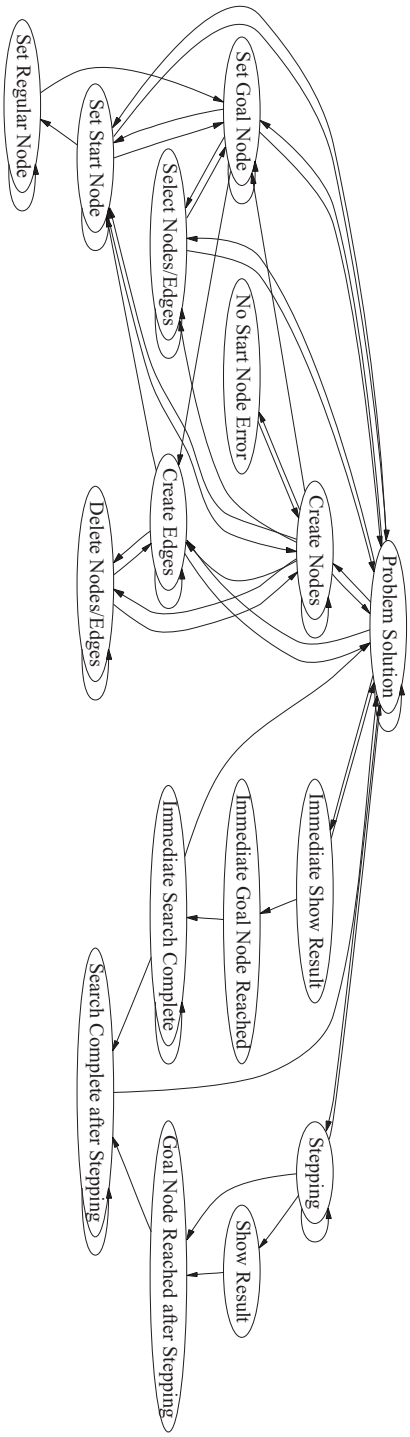

Figure 2.2: The Base State Space

hand one corresponds to problem creation mode. The students were mainly using the application to solve problems that were given to them, so we recorded significantly more data for problem solution mode. The following discussion therefore focusses on the right hand subcomponent of Figure 2.2.

In this component, we see two distinct ways of examining search algorithms using this application. Students can either step through a problem using a search algorithm, or they can show the result of the algorithm given the problem. At most times they can reset the search, which transports them back to the **Problem Solution** start state. While stepping, they can still ask to be shown the result at any time. **Show Result** and **Goal Node Reached** are the states in which dialog boxes are shown. We can distinguish two versions of these states, one in which the student has stepped previously, and one in which the student asked to see the result directly from the **Problem Solution** state. Note that we are unable to distinguish, for example, between a state in which the user is stepping through a problem and a state in which the user is fine-stepping through a problem, because the interface configurations for these states appear identical.

It should be obvious that while this base model captures some aspects of the application, its states are very coarse in nature. For example, knowing that the application is in its **Stepping** interface state does not indicate whether the user will choose to Step, Fine Step or Reset the Search next, and **Problem Solution** state tells us little about whether a student will choose to step or examine the result. However, there are clues in the recorded history that yield far more information that the base model's state. For example, a user that has just stepped is likely to step again, and a user that has been viewing the answers to various algorithm is likely to view the result again after switching to a new search algorithm. In other words,

the states in our base model in many cases do not fulfill the Markov assumption. History can be used to predict what the user will do when he or she enters a state of the base model. In the next chapter, we will discuss how to analyse history to split a state of our base model into more informative substates.

Chapter 3

Explicit State Identification

3.1 Goal and Approach

We now discuss how to explicitly refine the states of the base model presented in section 2.3. Our goal is to produce an accurate stochastic dynamic model of application use that can be used for interface analysis and as a basis for intelligent interface components. In general, all our refinement strategies will rely on checking the Markov assumption for the states of the model. That is, we always attempt to predict what will happen in a state by mining the observed history. If we find that we can make such predictions, we say that the Markov assumption does not hold for this state and replace it with ones that are more Markovian. To produce these new states, we form substates of the original state by attaching the history segments that let us make predictions. In off-line model building there exists no current context for making predictions, thus the question to be answered is rather general: Are there any sequences that yield information about events in a significant portion of a state's occurrences in the observed history?

This state refinement problem can be viewed as a classification task: given

the occurrences of a state in recorded history, the action sequences that precede them, and the actions that follow them, how should we group the sequences such that the groups give us as much information as possible about what action will occur after the state? This problem sounds much akin to the problem of picking an attribute to split on in building a decision tree using ID3 (Quinlan 1986). However, the natural attributes to use in splitting a state are the action sequences preceding it. These attributes are many-valued, producing a split into a large number of substates. Instead, we would like to split the state into as many states 'as make sense', that is, as are useful in capturing possible user intentions when reaching the observed interface state. We need to dynamically construct attributes with fewer values to produce more Markovian k-Markov states.

OFESI performs hierarchical binary splits of a state according to how much information such a split yields about the actions taken from it. Grouping the preceding action sequences to maximize this information supplies us with a new state. This k-Markov state is more Markovian than its parent state in the sense that additional information from history is less useful in predicting events in the state. Such a state captures more detailed knowledge about the current context of user and application.

3.2 The OFESI Algorithm

Figure 3.1 depicts the schema OFESI employs in splitting a state. We initially consider all distinct fixed length history sequences that have preceded an interface configuration in recorded history. For configuration $C(S_i)$ and length l, we call this set

$$H_{S_i}(l) = \{h_{t-l}h_{t-l-1}\ldots h_{t-1}\, s.t.\, c_t = C(S_i)\} \tag{3.1}$$

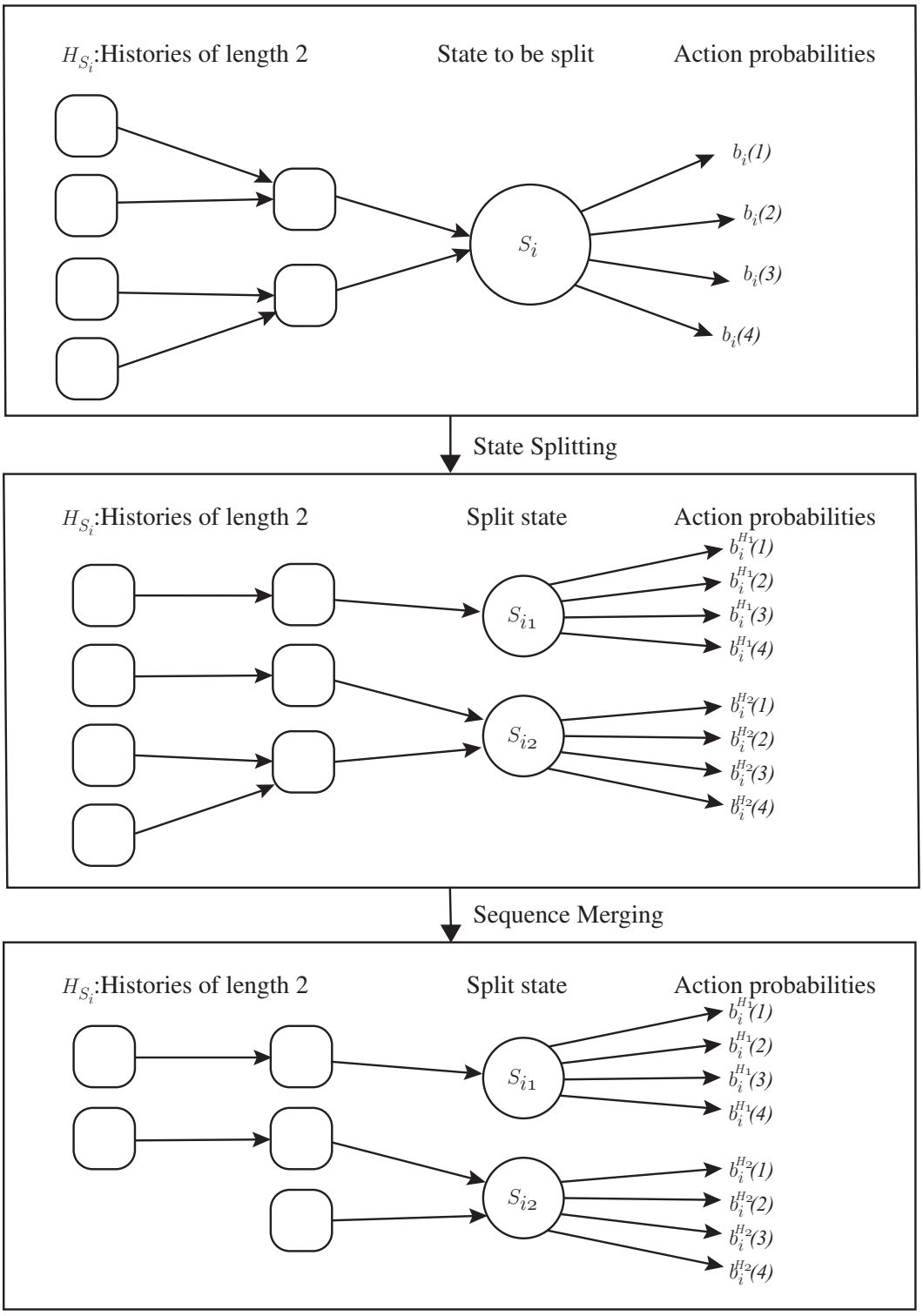

Figure 3.1: OFESI State Splitting Method

The length of these sequences is an input to the algorithm, and the fact the we limit the length of our sequences makes this Markov assumption check approximate. However, it seems likely that more recent events are more likely to yield information about the current state. We discuss the choice of length in Section 3.3. We now split $H_{S_i}(l)$ into two subsets, H_1 and H_2. For the resulting sets, we estimate the distributions associated with a state in the following way:

$$b_i^{H_1}(k) = \frac{\sum_{t=1}^{T-l} \sum_{g \in H_1} u(t, g, C(S_i)) p(t, k)}{\sum_{t=1}^{T-l} \sum_{g \in H_1} u(t, g, C(S_i))} \tag{3.2}$$

where

$$u(t, g, C_m) = \begin{cases} 1 & \text{if } H \text{ equals } g \text{ from } t - |g| \text{ to } t - 1, \text{ and } c_t = C_m \\ 0 & \text{otherwise} \end{cases} \tag{3.3}$$

Along the same lines,

$$a_{ij}^{H_1} = \frac{\sum_{t=1}^{T-l} \sum_{g \in H_1} r(t, j) u(t, g, C(S_i))}{\sum_{t=1}^{T-l} \sum_{g \in H_1} u(t, g, C(S_i))} \tag{3.4}$$

We now associate H_1 with these distributions and the interface configuration $C(S_i)$ to turn it into a candidate state for our model. Similarly, we turn the set of sequences H_2 into a candidate state.

The main question now is: how can we split $H_{S_i(l)}$ into H_1 and H_2, such that the resulting sets convey as much information as possible about the next action distribution $\{a_{ij}\}_{j=1}^{N}$? Just as in building a decision tree, we evaluate possible sets according to their information gain. In decision tree building, information gain lets one pick the attribute that yields the most information about the attribute to be predicted. Here, we are looking the attribute that conveys the most information about the actions occurring in a state. The difference is that in decision trees

the attributes are usually given, whereas we dynamically construct them as sets of history segments. For H_{S_i} the information needed to fully predict the next action is (Shannon and Weaver 1949):

$$I(S_i) = -\sum_{k=1}^{M} b_i(k) \log b_i(k) \tag{3.5}$$

A new state S_{i1} induced by $H_1 \subset H_{S_i(l)}$ leaves us with a remaining information need of

$$R(S_{i1}) = -\sum_{k=1}^{M} b_i^{H_1}(k) \log b_i^{H_1}(k), \tag{3.6}$$

for that part of the original state, so the split of S_i into S_{i1} and S_{i2} yields an information gain of

$$G(S_i, S_{i1}, S_{i2}) = I(S_i) - P(S_{i1})R(S_{i1}) - P(S_{i2})R(S_{i2}), \tag{3.7}$$

where

$$P(S_{i1}) = \frac{\sum_{t=1}^{T} \sum_{g \in H_1} u(t, g, C(S_i))}{\sum_{t=1}^{T} \sum_{g \in H_{S_i}(l)} u(t, g, C(S_i))} \tag{3.8}$$

is the probability with which the predictions grouped into substate S_{i1} occur (similarly for $P(S_{i2})$. We need to split the set of action sequences into two subsets such that this value is maximized.

We know of no global heuristics to guide this maximization and the search space grows quickly with increasing l, so we choose an approximate optimization technique. We use a form of stochastic local search (Hoos 1998), which may not find the global maximum, but is likely to find a good split after a reasonable number of steps. Specifically, we initialize the search by splitting the history sequences leading to the interface state into two random subsets. Each search step moves exactly one sequence from one set to the other. To perform a step we pick the move that

increases information gain the most with probability p and pick a random move with probability $1 - p$. We only move attributes that have not been moved for at least k steps, and reset the search process if information gain has not improved for m search steps. We discuss the parameter settings for this search in Section 3.3.3.

OFESI accepts the best split if it yields an information gain of at least G_{min} and if for each set the sum of the action instances that set predicts is at least A_{min}. G_{min} restricts splits to those that still yield a reasonable amount of information about the actions taken from a state. A_{min} prevents OFESI from creating states that explain very few actual instances of their parent state. That is, if there are two instances of a state, each with a unique history segment before and a unique action after the instance, information gain tells OFESI to split the state. A_{min} prevents this from happening. A better approach may be to weigh the increase in complexity in the model against its descriptive power, but we have not implemented such a measure. Both G_{min} and A_{min} cannot be optimized in any objective sense, but rather depend on one's goals in using OFESI. We discuss their impact in Section 3.3.2.

There is likely to be some redundancy amongst the resulting sets. For example, if all preceding sequences that end in the same action are grouped into one of the substates, that single action is enough to identify all these sequences. Thus, OFESI's last step examines the sequence sets for such redundancies and replaces groups of sequences with a shorter sequence wherever possible. Upon a successful split, OFESI now considers each substate as the new state to be split and continues splitting in this hierarchical fashion as long as each split fulfills the G_{min} and A_{min} restrictions. The history segments considered in each subsequent split are a subset of the original set. Figures 3.2 and 3.3 outline the complete OFESI algorithm.

```
//States are represented by sets of history segments leading to single
//interface states.
Given settings for Amin, Gmin,
      sequenceLength;
OFESI() {
   For each base model state s {
       Let resultingStates = {};
       Find all actionSequences of length sequenceLength
            that precede s;
       Call OFESI-split-state(actionSequences, resultingStates);
       Remove redundancies from segments in resultingStates sets
       Replace s by substates found in resultingStates;
   }
}

OFESI-split-state(actionSequences, resultingStates) {
   Let {gain, substate1, substate2} = OFESI-binary-split(actionSequences);
   If gain > Gmin AND the number of actions predicted by each of
          state1 and state2 is greater than Amin {
      Call OFESI-split-state(state1, resultingStates);
      Call OFESI-split-state(state2, resultingStates);
    } else {
       Append actionSequences to resultingStates;
    }
}
```

Figure 3.2: The OFESI Algorithm

```
Given settings for searchSteps, p, k, m;
OFESI-binary-split(actionSequences) {
   Randomly split actionSequences into state1, state2;
   Let gain = G(actionSequences, state1, state2);
   Let bestGain = gain;
   Let stagnantCount = 0;
   Let bestSplit = {state1, state2}
   For searchSteps number of times {
      With probability p {
         Move action a that maximises gain improvement
                  when moved between sets and hasn't been
                  moved for k steps;
      } else {
         Move random action a;
      }
      Let gain = G(actionSequences, state1, state2);
      If gain > bestGain {
         Let bestGain = gain;
         Let bestSplit = {state1, state2};
         Let stagnantCount = 0;
      } else {
         stagnantCount++;
      }
      if(stagnantCount > m)
         Randomly split actionSequences into
             state1, state2;
   }
   return {bestGain, bestSplit};
}
```

Figure 3.3: Binary State Split Algorithm

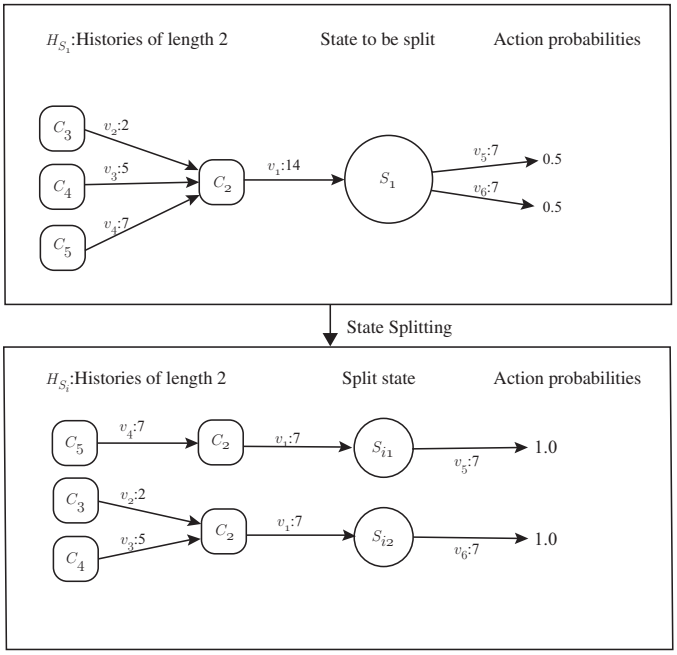

Figure 3.4: Example of OFESI State Split

Consider the example depicted in Figure 3.4. We see a state S_1 with interface configuration $C(S_1) = C_1$ that is always preceded by (C_2, v_1), which in turn is preceded 5 times by (C_3, v_2), 2 times by (C_4, v_3) and 7 times by (C_5, v_4). Whenever the user arrives in C_1 via C_3 or C_4 he or she chooses action v_5, whereas C_5 leads him or her to execute v_6. Assume that v_5 and v_6 are the only actions that occur in C_1. It follows that for the base model state S_1 we have $b_1(5) = 0.5$ and $b_1(6) = 0.5$. Running OFESI with $l = 2$, $G_{min} = 0.5$ and $A_{min} = 3$ and using base 2 logarithms we see that the best top level split creates $H_1 = \{((C_3, v_2), (C_2, v_1)), ((C_4, v_3), (C_2, v_1))\}$ and $H_2 = \{((C_5, v_4), (C_2, v_1))\}$ because the information gain of this split is a maximal

$$G(S_1, S_{11}, S_{12}) = I(S_1) - 0.5R(S_{11}) - 0.5R(S_{12}) =$$

$$-0.5\log(0.5) - 0.5\log(0.5) - 0.5(-0.71\log(0.71) - 0.4\log(0.4)) - 0.5(1\log(1))$$

$= 0.56.$

According to G_{min}, S_{12} can be split further because the only possible split yields $G(S_{12}, \ldots) = 0.86$, but as one of the resulting states explains too few actions (namely 2), the $A_{min} = 3$ setting prevents OFESI from splitting further. To illustrate the merging of redundant history segments, consider a similar situation as above, but with $H_2 = \{((C_5, v_4), (C_6, v_7))\}$ and H_1 unchanged. After splitting as above, OFESI can shorten the sets to $H_1 = \{((C_2, v_1))\}$ and $H_2 = \{((C_6, v_7))\}$ without loss of information.

3.3 Results and Discussion

There exists no obvious user-independent performance measure for OFESI. Its usefulness depends on the goals of the person employing the system, be it to debug an existing application, to design additional application components or to simply perform a study of application usage. We evaluated the implicit version of our state identification approach to predict future user actions in chapter 4 and in chapter 5 we discuss an application of OFESI to another problem that can benefit from explicit state identification and that provides a more objective performance measure, namely that of deriving structure for Hidden Markov Models (Rabiner 1989). In the following sections, we present the model OFESI derives for the search application and argue that it captures significant features of user behaviour.

3.3.1 State Splitting Example

First, let us examine the state **Stepping** and how OFESI splits it. Figure 3.5 show the two splits that occur. We see that there are three main actions users choose

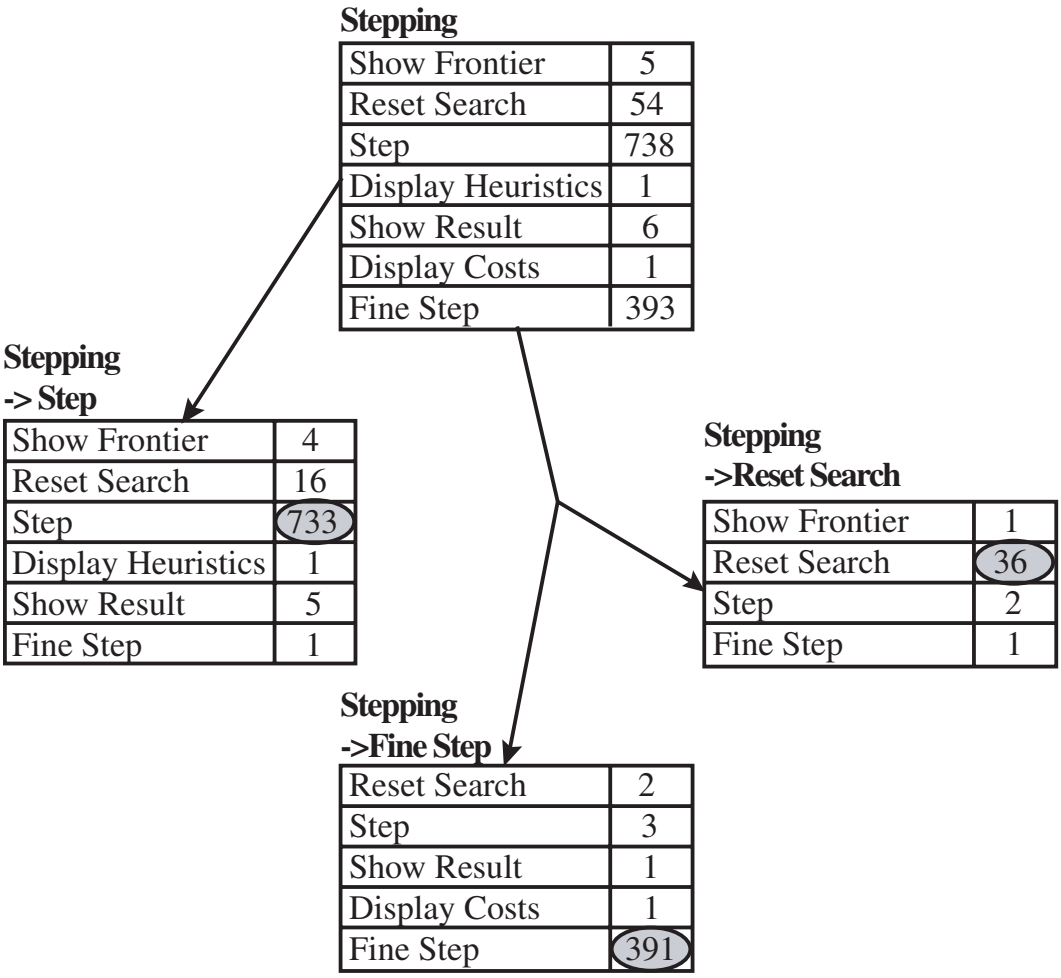

Figure 3.5: Split of the Stepping State with $G_{min} = 0.15$ and $A_{min} = 10$

Table 3.1: Length 1 Sequences Defining **Stepping-Step**

Previous State	**Previous Action**
Problem Solution	Step
Stepping	Display Edge Costs
Stepping	Step

from the original **Stepping** state: They either step, fine step, or reset the search. Intuitively, we would like to split the state into three substates, each predicted by an appropriate set of history sequences leading to it. As visible in Figure 3.5, OFESI suggests exactly three states when considering history sequences of length one. Table 3.1 gives the action sequences predicting the new **Stepping->Step** state. This substate is a result of the initial binary split of **Stepping** OFESI performs, which splits the state into one substate predominantly predicting the Step action and another predicting the Fine Step and Reset Search actions.

This substate makes sense, judging from the intuition behind the action sequences that predict it (a user that stepped before is likely to step again), and from the next action distribution that is dominated by the Step action. This substate will not be split again, but OFESI does split its dual substate in a hierarchical call into substates that predict the Fine Step action and the Reset Search action (it is predicted by the last state having been **Goal Node Reached after Stepping**.) Figure 3.5 shows the next action counts for all three states. OFESI was run with with $G_{min} = 0.15$, $A_{min} = 10$ and $l = 1$.

The choice of action sequence length constitutes a trade off between computational and explanatory complexity on the one hand and explanatory power on the other hand. That is, short action sequences (say, of length one) are easy to read and there are few of them, whereas there tend to be exponentially more sequences

Table 3.2: Sequence length 4: Next Actions Counts

Action	Count
Show Frontier	2
Reset Search	13
Step	736
Display Node Heuristics	1
Show Result	4

with each additional action considered. More longer sequences allow us to split the state at least as well, and usually better, than few shorter sequences, but due to their number the splitting process is computational more expensive and the resulting substates are hard to interpret based on the action sequences (they are still often easily interpretable from the actions they predict.) At the same time, overfitting may occur with longer action sequences in the sense that patterns peculiar to the training history may be used to identify substates.

Table 3.2 shows the action distribution for the state **Stepping->Step** as derived by OFESI run with sequence length four. It is clear that the split is cleaner - there are no more Fine Step actions predicted by this state, and more Step actions predicted. We refrain from including the unwieldy set of action sequences that predicts this state. Upon examining this set we find that in essence OFESI was able to take into account features like that if the last action was Step, but the three before that were Fine Step, the user is likely to choose Fine Step next.

3.3.2 Refined State Space

Figure 3.6 shows the state space after running OFESI on the right hand component of state space shown in Figure 2.2. States are labelled by their original name followed

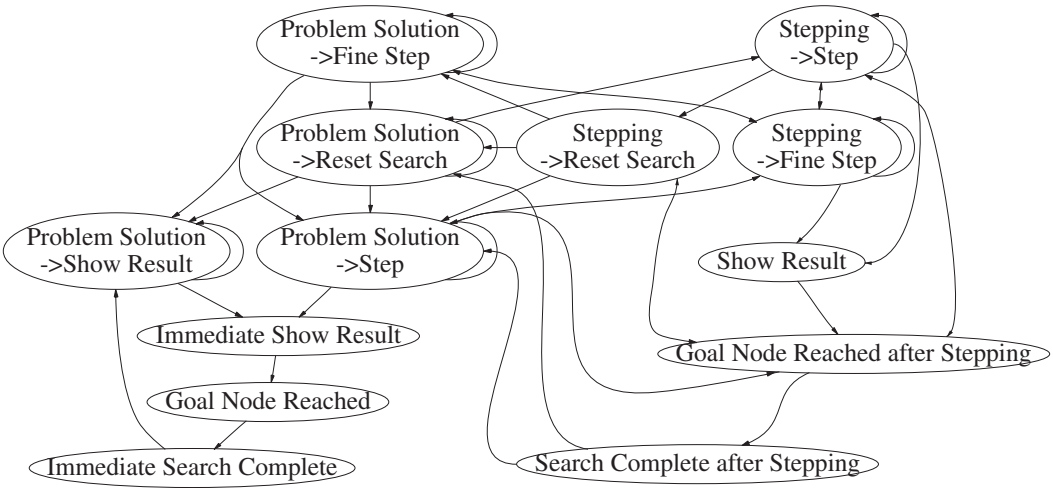

Figure 3.6: Split State Subspace for Search Application

by the most frequently occurring action in their next action distribution if they are substates of the originally observed interface states. OFESI was run with $G_{min} = 0.15$, $A_{min} = 10$ and an action sequence length of 5. These parameters can be set differently according to one's goal in running OFESI. Generally, lower settings of G_{min} will produce more states, but states at lower levels of the hierarchical split will tend to account for fewer actual occurrences of the parent state. Lower settings of A_{min} will also produce more states, but these states will tend to distinguish actions that occur less often. First, notice the three substates of the **Stepping** state as discussed in Section 3.3.1. In the same fashion, OFESI splits **Problem Solution** into four substates, according to whether the user is likely to Step or Fine Step through the problem, to ask to be shown the result or to reset the search. The last of these **Problem Solution/ActionEvent on Reset Search** is an artifact of the user interface design. The problem solution process is in its initial state if the user is in **Problem Solution** state, but users ask for the process to be reset both after just having reset it and after switching search strategies or search options (as obvious

from the action sequences OFESI attaches to the new substate.) The problem is that, for example, in the interface configuration shown in figure 2.1 a click on the Reset Search button will not disable the button, thus leaving the user unclear as to whether the action was performed. Clearly, it is due to a flaw in the interface that users engage in this behaviour. In general, the state space shown in Figure 3.6 presents much more detailed model of application use, and its states yield far more predictive power than the originally observed coarse interface states.

3.3.3 Information Gain Optimization

As mentioned before, we employ stochastic local search to optimize information gain when splitting a state into two substates. Stochastic Local Search promises good results, without needing to know details about the search space. The complexity of this search largely depends on the length of the action sequences considered. With sequences of length one, there are only few items to be grouped and most parameter settings find the optimal solution quickly. With longer sequence the number of items in the groups tends to grow exponentially, and the search quickly becomes more complex. However, we found that with sequences of length five the search rarely find better solutions after more than 500 steps. In addition, the improvements in later steps tend to be much smaller than in earlier ones and we do not necessarily care to find the optimal split as long as we find a very good one, so running the search to 500 steps, with a probability of random steps of 0.1 and restarting after the solution has not improved after 80 steps proved sufficient for our purposes. We do not claim that these are in any way the optimal parameters, but they appear to work well enough in practise.

Chapter 4

Implicit State Identification

4.1 Goal and Approach

So far, we have assumed that computation is performed off-line, and thus could engage in the information gain optimization procedure presented in chapter 3. There are many scenarios, like web page precaching, automation or command completion where one needs to predict which action the user is likely to perform next, but needs make this prediction before the user actually executes the next action. In this setting, several requirements change:

- Information gain optimization is too computationally expensive to perform on-line.
- OFESI's model is static and too expensive to recompute often, and thus cannot quickly adjust to changes in the user's behaviour.
- There now exists a current context as specified by the immediate history and the currently observed interface configuration that should be taken into account in making predictions to achieve greater accuracy. OFESI's approach is

general and ignores such information.

Instead of performing the detailed and general Markov assumption check OFESI engages in for each state, ONISI approximates this check for the current context. Working with the base model from section 2.3 where each state corresponds to an interface configuration, ONISI implicitly identifies the current state by re-ranking the actions in the state of the base model according to how much they are supported by history in the current context. There are several choices we must make as to how to mine the observed history to perform our approximate Markov assumption check. We need to choose

1. A type of pattern to extract from the interaction history that can be matched against recently occurring actions,

2. A method to summarize the occurrence of a pattern in history,

3. A function that ranks the currently possible actions according to the summaries of applicable patterns.

IPAM (Davison and Hirsh 1998) exemplifies one possible set of choices. Davison and Hirsh (1998) choose pairs of actions occurring in sequence as a pattern and summarize them by increasing the probability of those that occur and decreasing the probabilities of all others, achieving are running estimate of $P(O_t|O_{t-1})$. Finally, they rank the currently possible actions by considering all the pairs that start with the action that occurred one timestep ago and selecting the one of highest estimated probability to predict the next action. These choices make an implicit Markov assumption, namely that the last action together with the current summary provided by the probability estimates contain enough information to predict the next state. Looking at the user interaction traces described in chapter 2 we

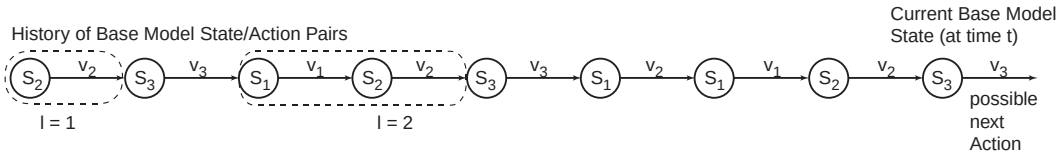

Figure 4.1: Example for On-Line Implicit State Identification

found that often users enter modes that can be easily identified by examining the behavioural patterns they engage in, but that these patterns span more than two actions in a row. For example, in the search algorithm application we investigated users would either step through a problem or simply look at an algorithm's result. Both of these modes can be identified and the actions that occur in them predicted by looking several steps back. IPAM fails to do so, and cannot make a significant fraction of predictions in this case.

4.2 The ONISI Algorithm

To remediate this problem, we decided to automatically vary the length of patterns we identify. Indeed, we deem patterns more important to the state identification process if they are longer, building on the idea that longer patterns capture the current state better. Overall, we choose our patterns to be the longest sequences in history that match the immediate history. Given a small integer k, in state $q_t = S_i$ we summarize the sequences that predict action v_j by computing $l_t(S_i, v_j)$: the average of the lengths of the k longest sequences that end with action v_j in state S_i and match the history sequence immediately prior to time t. We hypothesize that $l_t(S_i, v_j)$ gives us an indication of how much history supports action v_j in state S_i. Thus, we use l_t as our primary ranking to predict actions. We only consider k maximum length sequences due to our decision to view match length

as the important criterion and under the assumption that there will be few long sequence. In a sense, we are voluntarily overfitting our prediction to a few long history matches. However, it is unlikely for such matches to occur at random. In short, rather than estimating the probability of the next action given the previous action like IPAM does, we estimate the probability of the next action given as much as we deem significant of the history up to this timestep. We determine significance by whether this sequence occurs previously in the observed history. This approach leans on a similar one used to identify hidden states during reinforcement learning (McCallum 1996). It is also akin to automation approaches that identify repeated patterns in user history (Ruvini and Dory 2000), but does not rely on an evaluation of these patterns.

However, l_t captures no frequency information and therefore fails to distinguish between actions that have equal length history matches supporting them. To account for these case, we weigh our frequency derived base model probabilities into the ranking. The tradeoff between base model probabilities and re-estimates according to match length uses a parameter $0 \leq \alpha \leq 1$ that must be experimentally determined for each dataset. Our results confirm the expectation that α should assign most of the weight to the length measure l_t as it is the more predictive term. On the whole, ONISI can be viewed as performing an approximate Markov check on the base model states. It re-ranks actions according to history support and weighs in the base model if the history ranking fails to distinguish between actions. Figure 4.2 summarizes the ONISI k-nearest neighbours algorithm.

When normalized across all possible actions from the current state, $R(S_i, v_j)$ can be interpreted as replacing the probability of each action, $b_i(j)$ at time t. Figure 4.1 shows an example step of ONISI on a short history sequence. The application

Given the current base model state S_i (corresponding to the currently observed interface configuration, i.e. $C(S_i) = c_t$) the action v_j under consideration, a small integer k to indicate how many pattern matches to consider and a real value $0 \leq \alpha \leq 1$ to indicate how to weigh off between the match length measure and base model probability, where $\alpha = 1$ uses only the match length measure and $\alpha = 0$ only base model probabilities,

1. Compare the immediate history starting at t with the interface configuration/action pair (c_t, v_j) and running backwards through all the recorded history. Find the k longest sequences in the recorded history that match the immediate history.

2. Average the length of these sequences (excluding the current interface configuration / action pair, so a match is length 0 if only that pair matched or if the pair does not occur at all) and call this average $l_t(S_i, v_j)$.

3. Return ranking

$$R_t(S_i, v_j) = \alpha \frac{l_t(S_i, v_j)}{\sum_{n=1}^{M} l_t(S_i, v_n)} + (1 - \alpha) b_i(j)$$

Figure 4.2: The ONISI k-nearest neighbours algorithm

is currently observed to be in interface configuration c_t which corresponds to base model state S_3, i.e. $C(S_3) = c_t$. We choose to show base model states in this figure, but remember that they are in one-to-one correspondence with interface configurations. The algorithm is ranking the possible action v_3 from that state with $k = 3$. As shown, it finds $\{2, 1, 0\}$ as the set of maximum length sequences matching the immediate history, and thus calculates

$$l_t(S_3, v_3) = \frac{0 + 1 + 2}{3} = 1.$$

Assuming that all actions provide a sum $\sum_j l_t(S_i, v_j) = 5$, that action v_3 has occurred 50 times in state S_3, and that S_3 has been visited 100 times overall, ONISI run with $\alpha = 0.9$ finally assigns a rank of

$$R_t(S_3, v_3) = 0.9\frac{1}{5} + 0.1\frac{50}{100} = 0.18 + 0.05 = 0.23$$

to action v_3 in base model state S_3 at time t.

4.3 Results

4.3.1 Parameter Settings

ONISI depends on two parameters: k and α. Figure 4.3 shows a graph of ONISI's performance over a range of values for these parameters, measured in percentage of actions predicted correctly for the search algorithm application (the application that yielded the largest dataset of about 2200 user actions.) The other two applications show similar trends, but due to the smaller dataset sizes they are less distinct. The graph continues to decrease for larger values of k and performance continues to get worse for smaller values of α.

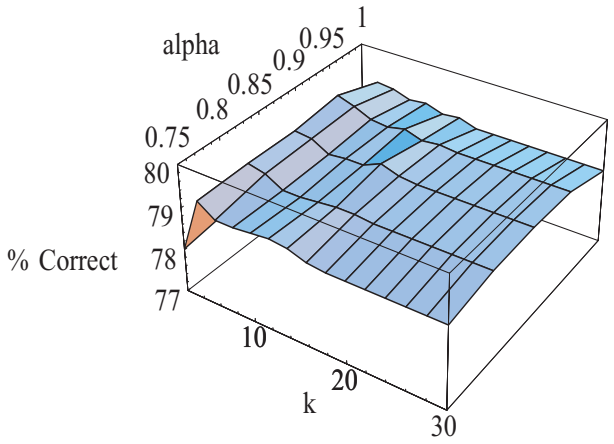

Figure 4.3: ONISI Performance at various parameter settings

The graph shows that small values of k suffice to estimate history support. In fact, performance decreases slightly for increasing k, a phenomenon that becomes even more pronounced at larger values of k. The problem is that shorter history segments are starting to dominate the average forming l_t, and thus actions appear less distinct. We attempted weighing longer matches more than shorted once, but performance decreased in any such attempts. Frequency and match length are not easily combined into one measure, because there are insufficiently many long matches to gain good frequency measures, but taking into account the frequency measures of shorter segments makes it difficult to weigh in long matches appropriately.

As for α, a setting that assigns almost all importance to the match length measure yields the best performance, but if the base model probability is ignored ($\alpha = 1.0$,) performance degrades. Upon inspection, the cases that are successfully predicted at $\alpha = 0.9$, but not at $\alpha = 1.0$ are indeed those where there are k maximal matches of same length in history for several actions and the default random choice between them that is used at $\alpha = 1.0$ performs worse than using the frequency measure to distinguish between them by setting $\alpha = 0.9$. All the following experiments

were run with $k = 5$ and $\alpha = 0.9$.

4.3.2 ONISI vs. IPAM

We compared the implicit version of our state space approximation to IPAM which its authors in turn compare to a number of others (Davison and Hirsh 1998). IPAM estimates $P(O_t|O_{t-1})$, i.e. the probability of the next action given the previous action. To do so, they build a table with all possible actions as rows and as columns and update its entries to be their desired probabilities during runtime. Specifically, they decrease all action pair probabilities in the row of the current action by a factor of $0 < \alpha < 1$ (note that this α is different from the α ONISI uses to trade of between frequency and length measure) and boost the probability of the action pair that just occurred by adding $1 - \alpha$, thus ensuring that the probabilities in a row continue to add up to 1.

We found IPAM to perform best on our data with a value of $\alpha = 0.9$, and all results here occur with that setting. The graphical user interface setting under investigation here differs somewhat from the UNIX command line prediction IPAM was designed for, and so does the state identification framework. To level the playing field, we let IPAM consider observed action-interface configuration pairs of the interface to be individual actions for its predictions. Taking state into consideration in this way produces better action distinctions, and IPAM's performance increases slightly (by ca. 2% for the search application.)

Figure 4.4 shows the percentage of actions predicted correctly by ONISI and IPAM for the search, neural network and constraint satisfaction applications. The numbers in parentheses indicate the number of actions recorded for each application. As baselines, the chart also indicates the percentage of actions predicted correctly

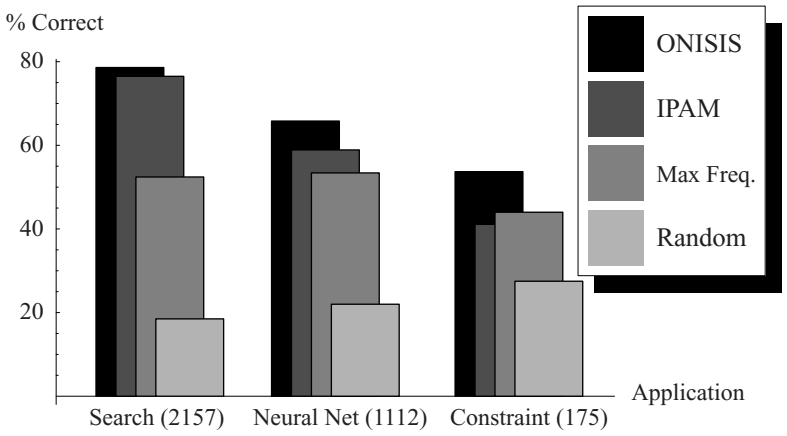

Figure 4.4: Performance of ONISI vs. IPAM

using just the maximum action frequency in a state (ONISI with $\alpha = 0$) and by picking a random action amongst all that have been observed in a state. In each case the algorithms ran in a realistic setting with no access to actions that lie in the future. As stated in section 2.2, there are few users who performed enough actions to allow for meaningful analysis on a per-user basis, so the data were treated as if produced by a single user in successive sessions. We believe that the applications are simple enough and the user's goals and experience levels similar enough to validate this simplification. The differences in the percentages of actions that can be predicted for each application stem partially from the differing number of actions recorded for each, but also from the nature of the application. First, the random prediction baseline shows that as a tendency the number of possible actions for a given interface states increases with the number of total actions we record from users (i.e. users did not explore as much of the application when using the constraint satisfaction application as of the search application.) Second, the frequency measure demonstrates that slightly less than half the actions in any of the applications can simply be predicted from the frequency distribution recorded in an interface state.

Third, It is expected that with less data to predict from all prediction performances decrease, except that of the random predictions. As expected, ONISI degrades more gracefully than IPAM, because IPAM requires some time for its predicted probabilities to converge.

As for the difference in prediction performance of ONISI and IPAM as compared to the frequency measure in the search and neural network applications: the recorded history for the search application includes long runs of the students stepping through a problem. These runs consist entirely of one action, and are easy to predict for either algorithm, making the overall prediction percentages of ONISI and IPAM high and close in value. Neither the neural net nor the constraint application show this type of student behaviour in the histories. To document this fact we recorded the average maximum match length ONISI detects when finding the k nearest neighbours. For the search application, this average lies at a length of 14.2 steps, whereas for the Neural Net application it is 4.6 steps, indicating that indeed long sequence matches are abundant in the search application.

Overall, ONISI performs better for each application with statistical significance at greater than 95% confidence as given by a chi-square test using a directional hypothesis for each of the applications. Why does ONISI perform better? Some clear patterns emerge when looking at the specific actions that it predicts but IPAM (and, we suspect, other approaches that work from a very short, fixed length interaction history) fail to predict:

- While the IPAM probability estimates take some time to converge for a new user, ONISI is able to identify a distinct behavioural pattern if it has only occurred once before.

- ONISI successfully identifies state information hidden in the user's history

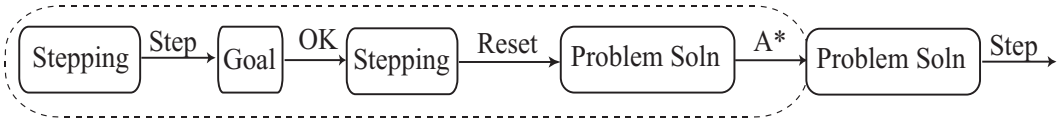

Figure 4.5: History Pattern Detected by ONISI

beyond the last action. For example, there are two modes in which users seem to learn about search algorithms with the search application. In one mode, a user will switch to a new search algorithm and step through the problem by hand to learn how the algorithm works.

In the other mode, as exemplified in Figure 4.5, a user wants to learn about the result of different search algorithms, rather than the process, and immediately displays this result after switching to a new search algorithm. The problem for IPAM consists of the fact that the last action, namely the switch to a different search algorithm, does not encode the mode in which the user is currently using the application. In fact, often even the last few actions are not sufficient to identify this mode, because the user has likely just finished another search problem and clicked through a sequence of result dialogues - a sequence that is the same for either mode. ONISI, however, easily distinguishes these modes, because it finds some nearest neighbours that match the whole history sequence in the Figure 4.5 and further actions. These are of greater length than sequences that only match backwards to where the goal node was reached, but contain the actions to immediately show a search result before that point.

Even more interesting than the fact that ONISI performs better than IPAM is that they predict different actions correctly. For the search application, there are 160 cases (ca. 7.5%) in which one of the algorithms predicts correctly and the

other does not. This means that one does not replace the other, and we belive that there is a systematic reason for this that can be exploited in further research. Also note that one can likely outperform both approaches with an informed modeling approach. For example, if it is known that the goal of a student is to understand a set of search algorithms, it may be possible to maintain estimates of the student's understanding of each algorithm and thus predict which algorithm he or she will likely focus on, and how he or she will explore it.

Chapter 5

SIHMM

While OFESI in chapter 3 appears to build useful user models, that judgement remains somewhat subjective. To prove that OFESI suggests the correct states to model a time series of observations, we now present the results of using OFESI to construct states for Hidden Markov Models (HMMs).

5.1 Structure Learning for Hidden Markov Models

An HMM models the process that produces an observed signal. The only difference between an HMM and the models presented in earlier chapters is that we added the interface configuration mapping $C(S_i)$. Rabiner (1989) explains how HMMs can be used to model sequences. In particular, HMMs can be trained using the Baum-Welch algorithm, which locally maximizes the likelihood of the model producing the training data.

Rabiner (1989) assumes that the number of states for an HMM is known a priori, together with much structural information to let the Baum-Welch algorithm converge to a global optimum. He also presents algorithms to retrieve the most likely

state sequence that explains the observation sequence (Viterbi), and to evaluate how likely it is that a given HMM produced the observation sequence (Forward Algorithm.) However, for many problems the structure of the Hidden Markov Model, including the number of states and their basic connectivity is not known. In light of this, a number of approaches have been proposed that try to learn an HMM's structure in addition to its parameters.

Broadly, most of the proposed structure learning algorithms rely on generate-and-test methodology. They often use very simple means of generating candidate models, for example by starting with a model that fully characterizes the observed data and merging states with similar output characteristics (Stolcke and Omohundro 1994) or "V-merging" two similar states that share transitions from or to a common state (Seymore, McCallum and Rosenfeld 1999). Stolcke and Omohundro (1994) employ a prior that punishes large models and maximize a posteriori likelihood of the data, whereas Seymore et al. (1999) test the classification performance of candidate models on a validation set. Brand (1998) proposes a direct modification of the Baum-Welch algorithm through an entropic reestimator, that takes into account the entropy of the HMM explanation of the data. By forcing weak transitions (as defined by his entropic score) to 0 he performs some structure learning, but an estimate of the number of states involved and some prior structure knowledge are still required.

In the following, we propose an accurate method of scoring the usefulness of a possible new state based on how its parent state is used to explain the observation sequence. This results in an algorithm that only introduces meaningful states and introduces them in the right places, taking care of structure learning and helping Baum-Welch reestimation by providing a starting point closer to a global maximum.

5.2 OFESI for Hidden Markov Models

OFESI already has much of the functionality we are looking for in finding structures for HMMs. It tests the Markov assumption for each state, and replaces the state with more Markovian substates if appropriate. However, there are some differences between the problem OFESI solves and that of HMMs:

- HMMs do not have an equivalent entity to the interface configurations that let us induce a base model for OFESI. Note that HMM observations cannot be used in the same way because one state may explain many observations. We take advantage of being able to assign one interface configuration to each state in building our models.

- OFESI is mostly concerned with the user's actions in a state, and only tests the Markov assumption relative to the action probability distribution. There are two ways in which a state in a Markov model can be identified as non-Markovian: Either history lets us predict the observation we will make in the state better than the state's action distribution $\{b_i(k)\}$ (the check OFESI performs) or history lets us predict the state the model will transition to at the next timestep better than the state's transition distribution $\{a_{ij}\}$.

- an HMM contains only 1-Markov states, whereas OFESI introduced k-Markov states into the model.

We need to replace the essential functions interface configurations perform for OFESI and add the second kind of Markov check to arrive at an algorithm for Structure Identification for Hidden Markov Models. Furthermore, we need to convert the k-Markov states produced by OFESI into 1-Markov states for the HMM.

First, let us deal with the problem of the missing interface configurations. For OFESI, they induced an easily constructed base model because they can be observed at every timestep and they give some indication of the current state of user and application. The base model provides OFESI with a starting point for its modeling effort, but more importantly, it lets us observe a history that accounts both for the base model's state and the user's action at all points in time. This is important because we wish to check the Markov assumption relative to the history of the complete system, and observing user actions alone does not give a good indication of this. There is no such observable indication of state for the general HMM problem because observations in the standard HMM formulation are not a projection of the HMM's state space like we assume interface configurations to be for our Markov models. However, the Viterbi algorithm returns the most likely state sequence for a given observation sequence given a HMM. If we make sure that at any iteration in our state identification algorithm we have a completely specified HMM, we can replace the interface configuration observed at a given point in time with the HMM state the Viterbi algorithm assigns to that point in time. In short, our history now is $H' = \{(q_t, O_t)\}_{t=1}^{T}$. This carries over to the sets of history segments, so H_{S_i} becomes

$$H^b_{S_i}(l) = \{h'_{t-l}h'_{t-l-1}\ldots h'_{t-1} s.t. q_t = S_i\} \tag{5.1}$$

with $h'_i \in H'$. We add the b superfix to distinguish this set from the one defined for transitions below.

Second, we expand OFESI's Markov assumption check to include testing history for information about future state transitions. To do so, we define a history segment set similar to $H^b_{S_i}(l)$ but including the observation at time t, namely

$$H^a_{S_i}(l) = \{h_{t-l}h_{t-l-1}\ldots h_{t-1}, (q_t, O_t) s.t. q_t = S_i\} \tag{5.2}$$

Then, we simply add a second version of the information gain equations

$$I^a(S_i) = -\sum_{j=1}^{N} a_{ij} \log a_{ij} \tag{5.3}$$

and

$$R^a(S_{i1}) = -\sum_{j=1}^{N} a_{ij}^{H_1^a} \log a_{ij}^{H_1^a}, \tag{5.4}$$

where $H_1^a \subset H_{S_i}^a(l)$. The state splitting procedure based on transition probabilities is then the same as that for observation probabilities with these new definitions. For OFESI we did not consider transition probability distributions in our state splits, because in the user modeling setting observed actions largely correspond to state transitions. In other words, button clicks and menu selections in direct manipulation interfaces usually have deterministic enough effects and lead to distinct states, making them roughly equivalent to state transitions. The added overhead in considering state transitions in addition to actions seems not worth the small gain in modeling detail.

Third, we need to convert the resulting k-Markov model to the 1-Markov model that constitutes an HMM. We do so by simply discarding the histories attached to a state. To compensate for this loss of information, we iterate the two state splitting procedures introduced in the last paragraph until neither of them introduces any new states into the model. In this way, history dependencies can propagate backwards through the model structure and finally produce the 1-Markov model that is equivalent (or superior, due to the fixed length of the history segments considered) to the intermediate k-Markov models. We did not convert OFESI's models to 1-Markov models because the k-Markov states are more concise and easily interpreted by hand, which is one of the main goals of OFESI.

```
Input: Observation Sequence
Output: HMM modeling the Observation Sequence

Initialize HMM to have one state (or start with hand-defined
structure)
Do {
    Retrieve state sequence explaining observation sequence
        through Viterbi
    Form a history of (state, observation) pairs
    Run OFESI with this history. Start by optimizing information gain
        on observation probabilities. Alternate each iteration with
        optimizing information gain on transition probabilities.
    Use the history segments attached to states to reestimate
        observation and transition probabilities for the state
    Run Baum-Welch to finalize parameters
} Until neither the transition nor the observation split introduce any
    new states.
```

Figure 5.1: The SIHMM Algorithm

To summarize, SIHMM can start with an HMM containing only a single state, or with any pre-defined structure. At each stage, it produces a history by running the Viterbi algorithm to produce a state-observation pair for each point in time. Using this history, SIHMM alternates between checking the Markov assumption relative to the observation distribution and relative to the transition distribution. Once states are split, SIHMM reestimates the HMM parameters using equations 3.2 and 3.4 and runs the Baum-Welch algorithm to converge to maximum likelihood parameters. Figure 5.1 gives the SIHMM algorithm as pseudo-code.

5.3 Results

We tested SIHMM on a regular language used to evaluate HMM structure generation in (Stolcke and Omohundro 1994). For such noiseless problems OFESI can run with $A_{min} = 1$ as no spurious state splits will occur. Throughout, SIHMM runs OFESI

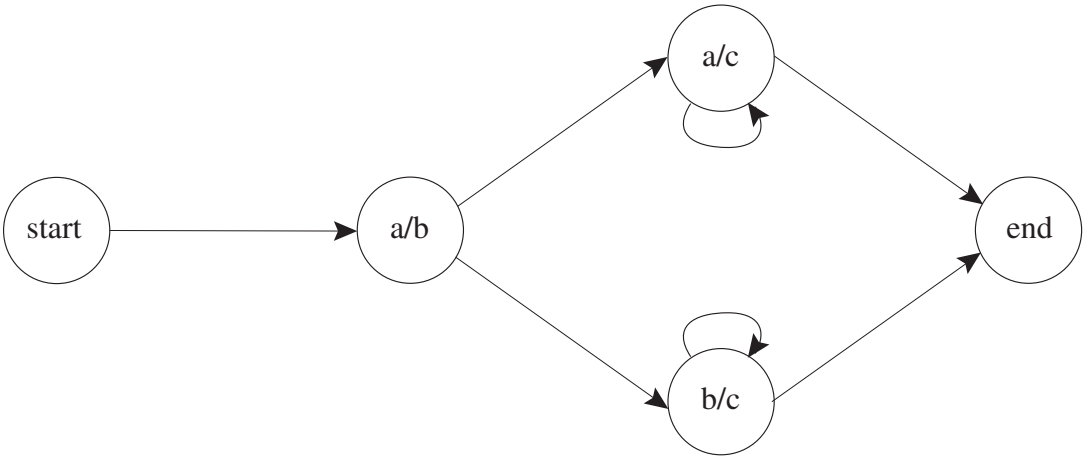

Figure 5.2: Initial Iteration SIHMM Splits by Observation for $ac^*a \cup bc^*b$

with $G_{min} = 0.05$ and the parameter settings for OFESI's search algorithm as discussed in section 3.3.3.

We discuss retrieval of an HMM producing the regular language $ac^*a \cup bc^*b$ from its output alone. We had the hand-crafted production HMM emit a string of 500 observations, where individual strings were separated by a start symbol emitted by a start state. For clarity, we have separated this start state into a start and an end state in the figures in this section. The HMM starts out as a single state model and thus all observations are associated with that single state. SIHMM's first split is thus based on how much sequences of observations of length three tell us about the observation that follows them.

Figure 5.2 shows SIHMM's first iteration splits with OFESI run on observation probabilities. SIHMM has split the initial HMM state into 4:

1. One state accounting for the start symbol (depicted by start and end state in figure 5.3). This state is preceded by either "cb", "ca", "aa" or "bb".

2. One state accounting for the initial "a" or "b", always preceded by the start

symbol.

3. One state accounting for any number of "c"s and a concluding "a", always preceded by a beginning "a".

4. Correspondingly, one state accounting for any number of "c"s and a concluding "b", always preceded by a beginning "b".

SIHMM has no notion of there being 'any number of "c"s,' but rather notices that after an initial "a" another "a" or a "c" follow with same likelihood, so it introduces one state capturing these histories. SIHMM cannot split the new state further based only one observations, because their symbol emissions, as dictated by the regular language, do not depend on the immediate history in any way. Thus, SIHMM attempts to split state by transition probabilities next.

Figure 5.3 shows that on the second iteration splits on transition probabilities, SIHMM manages to retrieve the source HMM. Whereas no prediction of observation symbols was possible on the initial state emitting "a" or "b", it is possible to predict which state the system will transition to based on whether the initial state emitted an "a" or a "b". Similarly, SIHMM splits the other two states because they transition to the end state if they emit a "c" and to themselves if they emit an "a" or a "b". It takes SIHMM only two iterations to arrive at the HMM producing this language, and the Baum-Welch runs SIHMM initiates to finalize the HMM parameters converge in a few steps without significant changes to the model structure.

We intend this example to show the power of SIHMM's informed state splitting algorithm. Due to time constraints and lack of suitable discrete data we have not yet evaluated SIHMM's performance on real data. The data used in the evalu-

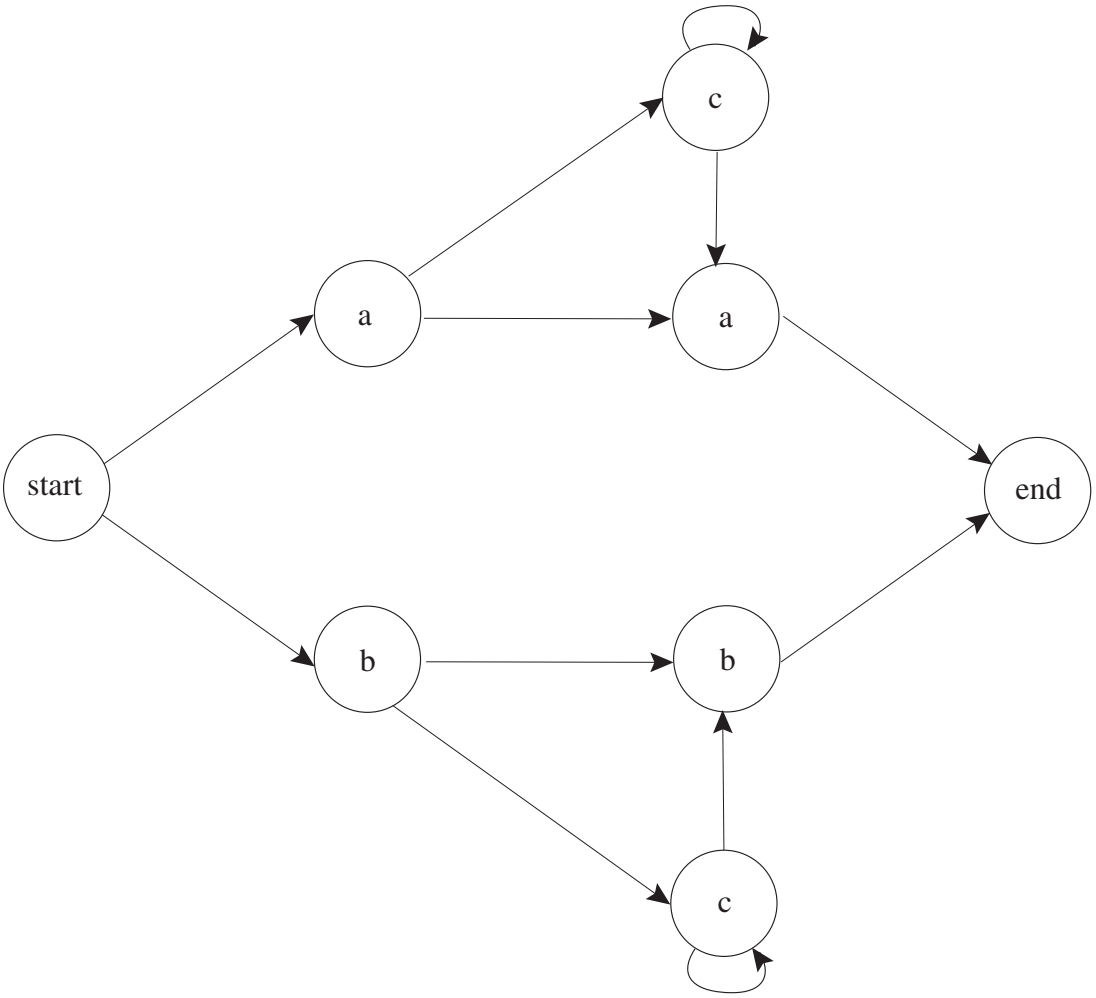

Figure 5.3: Second Iteration SIHMM Splits by State Transitions for $ac^*a \cup bc^*b$

ations in chapters 3 and 4 does not contain enough data points per user to be used for user identification, which would be the primary task for an HMM in modeling observed user interactions.

Chapter 6

Conclusion

In this work, we have concerned ourselves with using recorded observation histories to identify states that violate the Markov assumption and with using this identification to infer better states in the Markovian sense.

For user modeling purposes, we worked from a base model that can be easily induced from the observed user actions. We presented an explicit version of our state identification method, tailored for building humanly understandable models of application use that form the basis for further intelligent interface components. OFESI groups history sequences preceding states according to how much the resulting substates reveal about actions occurring in the parent state. In this way, the final state structure contains states that are as Markovian as possible without overfitting to noisy history traces.

To predict the next action of someone using an application, we implicitly refined the base model states by weighing possible actions by the lengths of matches to the history preceding the current state (ONISI). This on-line method captures users' behavioural patterns well (subject to sufficient data being available) and outperforms other next action prediction methods. As all methods presented here, it

can be implemented without modifying the application to be modeled. ONISI reveals that neither frequency nor match length is the ideal predictor for user actions.

Finally, we applied our explicit state identification scheme to Hidden Markov Models by taking into account state transition distributions in addition to observation distributions. We showed that our methods retrieve the source model from recorded observations.

Overall, we conclude that we were able to gain much valuable structure information from analysing a model for violations of the Markov assumption. Such structure information lets us automatically construct better uninformed models for prediction and modeling tasks. We point out again that this approach in now way replaces the detailed and meaningful knowledge that can be gained from an informed approach. We cannot determine the user's task or state of mind, but only capture a structured model of the application and the user's behaviour. Our claim is that this model has some direct uses, like next action prediction, but more importantly provides a solid foundation for designing informed models. Furthermore, the insights gained in analysing our models transfer to other uninformed modeling strategies, such as Hidden Markov Models.

In future, we hope to use both ONISI and OFESI to build intelligent user interface components like help systems and automation engines. Especially in OFESI we see the promise of building a general application analysis tool that makes model building and application extension easier. Other plans include building adaptive web sites based on OFESI's models (Perkowitz and Etzioni 2000), and assisting with building user plan libraries for interfaces (Lesh, Rich and Sidner 1999). There remains some analysis left to do in the comparison of ONISI and OFESI, as well as ONISI and IPAM. Both comparisons must investigate the trade-off between match

length and frequency of matches and their information content with respect to the current state. While the algorithms presented here exploit both, the trade-off can probably still be captured more precisely and then evaluated for different goals. Finally, we hope to apply SIHMM to more Hidden Markov Model problems, especially with real data.

Bibliography

Albrecht, D. W., Zukerman, I., Nicholson, A. E. and Bud, A.: 1997, Towards a bayesian model for keyhole plan recognition in large domains, *User Modeling: Proceedings of the Sixth International Conference, UM97*.

Brand, M.: 1998, An entropic estimator for structure discovery, *Proceedings of NIPS98*.

Cadez, I., Heckerman, D., Meek, C., Smyth, P. and White, S.: 2000, Visualization of navigation patterns on a web site using model based clustering, *Technical Report MSR-TR-00-18*, Unversity of California, Irvine.

Davison, B. D. and Hirsh, H.: 1998, Predicting sequences of user actions, *Technical report*, Rutgers, The State University of New York.

Encarnacao, L.: 1997, *Concept and Realization of intelligent user support in interactive graphics applications*, PhD thesis, Eberhard-Karls-Universität Tübingen, Fakultät für Informatik.

Gorniak, P. J.: 1998, Sorting email messages by topic. Project Report.

Gorniak, P. J. and Poole, D.: 2000a, Predicting future user actions by observing unmodified applications, *Proceedings of the 17th National Conference on Artificial Intelligence, AAAI-2000*.

Gorniak, P. J. and Poole, D. L.: 2000b, Building a stochastic dynamic model of application use, *Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence, UAI-2000*.

Hoos, H. H.: 1998, *Stochastic Local Search – Method, Models and Applications*, PhD thesis, Technische Universität Darmstadt.

Horvitz, E., Breese, J., Heckerman, D., Hovel, D. and Rommelse, K.: 1998, The lumiere project: Bayesian user modeling for inferring the goals and needs of software users, *Uncertainty in Artifical Intelligence, Proceedings of the Fourteenth Conference*.

JDK: 1998, *Java Development Kit Documentation*.
URL: *http://java.sun.com/products/jdk/1.1/docs/*

Lesh, N., Rich, C. and Sidner, C. L.: 1999, Using plan recognition in human-computer collaboration, *User Modeling: Proceedings of the 7th International Conference, UM99*.

Lieberman, H.: 1998, Integrating user interface agents with conventional applications, *Proceedings of the International Conference on Intelligent User Interfaces, San Francisco*.

McCallum, A. R.: 1996, Instance-based state identification for reinforcement learning, *Technical report*, University of Rochester.

Perkowitz, M. and Etzioni, O.: 2000, Towards adatptive web sites: Conceptual framework and case study, *Artificial Intelligence* **1–2**.

Quinlan, J.: 1986, Induction of decision trees, *Machine Learning* **1**, 81–106.

Rabiner, L.: 1989, A tutorial on hidden markov models and selected applications in speech recognition, *Proceedings of the IEEE*, Vol. 77(2).

Ruvini, J.-D. and Dory, C.: 2000, Ape: Learning user's habits to automate repetitive tasks, *Proceedings of the International Conference on Intelligent User Interfaces*.

Seymore, K., McCallum, A. R. and Rosenfeld, R.: 1999, Learning hidden markov model structure for information extraction, *AAAI-99 Workshop on Machine Learning for Information Extraction*.

Shannon, C. and Weaver, W.: 1949, *The Mathematical Theory of Communication*, University of Illionois Press, Urbana.

Stolcke, A. and Omohundro, S. M.: 1994, Best-first model merging for hidden markov model induction, *Technical report*, International Computer Science Institute, University of California at Berkeley.

Zukerman, I., Albrecht, D. and Nicholson, A.: 1999, Predicting users' requests on the www, *User Modeling: Proceedings of the 7th International Conference, UM99*.