An HMM Applied to Semi-Online Program Phase Analysis

Michael Otte and Scott Richardson

University of Colorado at Boulder
Technical Report CU-CS-1034-07

November 8, 2007

An HMM Applied to Semi-Online Program Phase Analysis

Michael Otte
Department of Computer Science
University of Colorado at Boulder
Michael.Otte@Colorado.edu

Scott Richardson
Department of Computer Science
University of Colorado at Boulder
Scott.Richardson@Colorado.edu

Abstract

Phase detection aims to identify the principle movements of a program by discovering the sections of a program's execution that are similar. Phase prediction actively anticipates program operation. This information provides knowledge of a program's behavioral patterns, which can then be used for analysis and optimization. We present two novel approaches to the problem of phase detection and prediction: a Vector Quantized hidden Markov model (VQ-HMM) and a Continuous Density hidden Markov model (CD-HMM). The performance of the models on the datasets varied widely. At their best, some models (defined by a set of model parameters) fit the data almost perfectly (> 90%), but when re-trained, could not consistently duplicate these results. However, certain parameter sets consistently produced models that reduced error by one fifth relative to a naïve model. The VQ-HMM performed better than the CD-HMM, but comparable to simple K-means clustering. In general, model performance is highly dependent on the disparity between the granularity at which phase behavior is detected and that at which it most naturally exists.

1 Introduction

1.1 Description of Domain

A program is said to display phase behavior when a program metric exhibits some degree of spatial and/or temporal stability/predictability over a section of program execution, punctuated by abrupt, regular variations; likely the product of the looping structure of the program. This metric could be one for resource use, such as IPC, cache miss rate, branch miss rate, etc, or one indicative of predictable code traces. Research has shown that phase behavior can be detected across a number of different metrics and that phase boundaries coincide across the metrics [5,11].

Phase detection aims to identify the principle movements of a program by discovering the sections of a program's execution that are similar – the phases. After classifying the program's behavior into phases, further analysis or optimization of the program's execution is possible; essentially automating the adage, "Identify the most frequently executed sections of a program, and make them fast".

Phase detection can be applied to both *offline* and *online* systems. An offline algorithm is not time sensitive; it has access to the complete instruction trace and potentially the code that generated the instruction trace. Simulation points, drawn from the set of phases, provide a representative cross-section of the entire program execution. Programmers or the program compiler can then use the simulated program execution profile to focus code optimization.

Online algorithms run in real time, in parallel to program execution, and therefore, only have access to information culled from the system, such as dispatched instructions and/or certain architectural metrics (e.g. cache misses), as they occur. Thus, in order to be effective, a phase detection algorithm must amortize the impact of its own

execution on the system (in lost instruction cycles). Most online phase detection algorithms can be described as interval-based classifiers aimed at delineating either phase boundaries or known phases. A roughly homogenous mixture of many similar intervals constitutes a phase.

1.2 Related Work

Attempts to characterize phases center around two approaches: interval-based and graph-based. As our approach is interval based, we will forgo a discussion of graph-based algorithms in the interest of space. An interval-based approach divides program execution into time slices called intervals or *windows*. A single phase usually consists of multiple intervals that exhibit similar structure. A phase typically lasts more than one interval, but the same phase may occur at multiple non-contiguous points during a program's execution. Phase information is gathered by examining the content of these intervals. This technique is typically online.

Several implementations of the interval-based approach have characterized an interval with a fingerprint. Nagpurkar et al. compares successive instruction sequences to determine if the current interval is part of the existing phase or a new/transitional phase[1] [13]. Another implementation that has met with success is Basic Block Vectors (BBV), as introduced by Sherwood et al. [16,17]. A basic block is a section of instructions that are executed from start to finish with one entry and one exit point. The distribution of basic blocks over an interval provides that interval with a fingerprint, known as the *basic block vector*. "If two fingerprints are similar, then the two intervals spend about the same amount of time in the same code, and the performance of the two intervals should be similar" (i.e. they are in phase) [17]. These BBVs are then compared via the Manhattan distance and either clustered to find representative intervals that are characteristic of the entire program execution trace or evaluated against a threshold to identify phase boundaries. In both cases, a phase is characterized by a succession of similar intervals.

A method for phase prediction typically accompanies an online phase detection algorithm. Phase prediction includes both when the next transition will occur, and what the new phase will be after the transition. It is worth noting that given only part of this information, optimizations may not be efficiently introduced. In conjunction, phase detection and prediction are used to guide dynamic system resource optimization. Phase prediction is valuable enough that offline profiling models have been used for online phase prediction [15].

A simple prediction scheme is a last value predictor. In this case, a process exhibiting stable behavior (i.e. in phase) is assumed to remain stable. This predictor provides an interesting baseline to compare more sophisticated prediction techniques against [4]. A table-based predictor, as implemented and tested by Duesterwald et al. [4], and similarly a Markov predictor, as implemented and tested by Sherwood et al. [17], are both improvements over the last value predictor. These techniques relate the next state of a system to the previous set of states by using "an encoding of past behavior history as an index into a prediction table. The prediction stored in the table is a value that was observed to immediately follow the behavior in the past" [4]. The Run Length Encoding Markov Predictor [17] uses a compressed (run-length encoded) version of the phase history to index into a prediction table. Specifically, this index is a hash of the phase id that was just seen, and the number of times prior to now that it has been seen in a row. Thus, in this way, the encoding contains information relating to phase switching behavior and not simply interval switching. Again, the prediction is just the last phase id that was previously observed to follow the current state.

1.3 Generative Models

We believe that there is room for improvement among the current incarnation of phase prediction algorithms. Known approaches, including the Markov predictor used in conjunction with Basic Block Vectors and the heuristic table-based approach, do not incorporate information about previous phase transitions in a sophisticated way. Generative machine learning algorithms, on the other hand, are built on such a probabilistic framework and have not been investigated.

A hidden Markov model (HMM) is one such model. An HMM is a graphical model that assumes both: observable states are probabilistically dependent on hidden – latent – states; the system being modeled is a Markov process. In this problem domain, states correspond to program phases. A Markov process is defined as a process in which the state at time t is independent of all but the state at time $t-1$. The task of the model is to infer the relationships

[1] Although this can be thought of as boundary detection, it is also a form of classification.

between the hidden states and a sequence of observations generated from a known language. As such, an HMM is well rooted in the theory of computation. It is analogous to a deterministic finite automaton with a memory of one. This empowers the HMM to infer relationships regarding the structure of the data.

HMMs diverge from previous algorithms that have used heuristics to identify and predict phases. By building a model that infers hidden states from observations, the nature of the system is more explicitly modeled. The separate problems of phase detection and phase prediction are combined into a single model and solved simultaneously in a principled way. Also, by modeling the structure of the program, an HMM can reveal inter-phase relationships, as well as trends across data sets. Ultimately, this information is used to facilitate informed optimization decisions.

2 Algorithms

2.1 A Typical Hidden Markov Model

An HMM is a graphical model defined by: the state-to-state transition probability matrix A; the language-state probability matrix B; and a prior probability π over the states. The state transition probability matrix supplies the likelihood of transitioning between any two states; the state graph is reflexive, thus self-transitions are possible. The language probability distribution matrix supplies the likelihood of an observation from the language given a state. The prior over the states supplies the probability that the model is initialized in a state. For our purposes, a phase of the program corresponds more or less to a state in the HMM, and the language is some set of program metric values.

There are three questions that an HMM can answer (for an through description of HMMs see [14]).

Given the observation sequence $O = O_1\ O_2 \bullet \bullet \bullet O_T$, and a model $\lambda = (A,B,\pi)$:

1) What is $P(O|\lambda)$, the probability of the observation sequence given the model? This provides an analysis of how well the model fits the observation sequence. Equivalently, this provides a measure of the predictive power of the model.

2) What is the corresponding state sequence $Q = q_1\ q_2 \bullet \bullet \bullet q_T$ that best explains the observation sequence? Solved via the Viterbi algorithm, the most likely state sequence given the observation sequence corresponds to the expected phase behavior of the program. An HMM assumes the process being modeled is Markovian. Therefore, the probability of state q_t at time t, conditioned on the previous state sequence $(q_{t-1} - q_1)$ and the set of observations at time t, is equivalent to the probability of state q_t conditioned on only the state q_{t-1} and the set of observations at time t:

$$P(q(t) \mid q(t-1), q(t-2)...q(1), O(t)) = P(q(t) \mid q(t-1), O(t)).$$

3) What are the model parameters (A,B,π) that maximize the likelihood of the observation sequence, *argmax* $P(O \mid \lambda)$? This inference is guaranteed to converge to a solution via the Baum-Welch algorithm—a variation of the EM algorithm that inherits its susceptibility to local optima. Thus, this is an iterative process, and although very efficient, is not fast enough to be performed online. As a compromise, our implementation generates the model offline (e.g. bootstrap from an initial program execution) and uses the model in subsequent executions of the program for online phase prediction. Given that program execution is dependent on the underlying code structure, it is possible to detect and predict phases in future runs of the program.

2.2 Variations of the HMM

A simple HMM is designed for use with a discrete set of observations. Typically, this is a set of unique symbols corresponding to the natural numbers 1 through N, where N is the number of distinct observations in the language. For our purposes, we focus on detecting phase behavior via a branch-instruction trace that is generated during the program's execution. The naive approach would be to train the HMM using the raw sequence of symbols generated by the execution trace (i.e. the language recognized by the HMM is the set of unique branch instructions in a given trace). However this presents a few problems. The raw data of an execution trace may contain hundreds or thousands of unique observations. Moreover, a single symbol contains relatively little information about the state of the program. In order to glean any useful information from the execution trace, many such observations must be combined. It is possible to increase the memory of the HMM (i.e. have the next state be dependent on a finite number of previous states), however this extension is exponential in complexity.

We explored two different HMMs that accept a set of observations, a Vector Quantized HMM (VQ-HMM) [14] and a Continuous Density HMM (CD-HMM) [12,14]. The first approach involves forming a codebook by clustering a set of observations into one of a finite number of sub-sets, or *clusters*, and then labeling each sub-set with a unique *code word*. The second approach involves modeling the set of observations as a vector in a high dimensional space. Both of these approaches, however, require that the set of observations are not intervals (or *windows*) of discrete symbols, but rather vectors in a continuous space. A straightforward solution to this problem is to process the branch-instruction traces into symbol count vectors. We call this *vectorization*.

Vectorization is accomplished by binning together the discrete observations from windows of the execution trace into a vector **W**, where ||W|| = N - the number of unique symbols in the execution trace. The value of each bin (or component) of the vector corresponds to the number of times that a particular symbol was observed in the observation window. This effectively turns a window's worth of discrete observations into a single continuous observation in N space. A skip size must also be considered. For example, a window size of 1000 branch-instructions with a skip size of 500 branch-instructions implies that each symbol count-vector (after the first) will contain exactly 500 unseen instructions. The distance between vectors in N space is a measure of their similarity, and therefore the similarity between the contents of the observation windows from which they were created. In practice, this has the effect of reducing the sequence length by several orders of magnitude. The main drawback to this approach is that temporal information provided by the ordering of symbols is lost at granularities of less than the window size. However, inter-window dependencies are still maintained.

In our approach, the count-vectors are then further processed via either vector quantization, for use in the VQ-HMM, or principle component analysis (PCA) [6], for use in the CD-HMM.

2.2.1 Vector Quantized HMM

Vector quantization is achieved by applying the K-means clustering algorithm [10] to the sequence of vectors **W** in N space. K-means is related to the expectation maximization (EM) algorithm. It assumes that the data set is generated from a Gaussian distribution and, by repeatedly applying the *E* and *M* steps, tries to minimize the total intra-cluster variance. These steps are respectively: assign each point in the data to one of *k* centroids (means), and recalculate the centroids of the clusters. The algorithm has converged once the centroids no longer move (or move tolerably). There is no upper bound for the number of iterations that this may take, and it is subject to local optima, however, K-means is one of the quicker clustering algorithms available. The centroid of a cluster is a vector that acts as the representative for the other vectors in the cluster. Thus, the sequence of continuous vectors is translated into a sequence of codes, or observations, used to train the VQ-HMM.

2.2.2 Continuous Density HMM

A CD-HMM is another variation of the traditional HMM that can handle more than a single discrete observation. In general, an HMM requires a probability estimate for an observation given a state. Thus, the standard language probability matrix *B* can be replaced with a probability density function, such as a normal distribution. In our case, we used a mixture of multivariate normal distributions to represent the probability of an observation vector in a state. Each dimension of the multivariate distribution is fitted to a component of the observation vector (i.e. a multivariate normal distribution is defined by the mean and standard deviation of the corresponding component in the set of vectors). A mixture of the multivariate normal distributions were used in order to more accurately fit the underlying probability density functions. This, however, can lead to a very high dimensional multivariate normal distribution, in which the distribution of points becomes unacceptably sparse. To combat this, we used principle component analysis (PCA) to reduce the dimensionality of the set of observation vectors. One assumption that we made by using PCA is that the data can be represented on a low dimensional linear manifold. In practice, we found that the first 1% of principle components accounted for ~40% of the variability of the data; hence this assumption does not seem unreasonable. Aside from these changes, the CD-HMM is trained much like the standard HMM.

3 Methodology

Our datasets consist of five sequences of symbols corresponding to branch path identifiers (representing method id, byte code index into the method that contains the branch, and whether or not a given path was taken) extracted from the programs 202_jess, 205_raytrace, 209_db, 213_javac, and 228_jack. The complete sequence of symbols trace the program's execution, while sub-sequences of symbols are identified as phases. The language size of the model is on the order of 100 to 1,000 discrete symbols. There is, however, no elapsed time or instruction count

information associated with the symbols. Note, one particularly important domain constraint lies in the non-portability of generative models between different programs. This is because the language of symbols between programs is different. Thus, a HMM must be trained and utilized with reference to a particular program, a consideration that makes a purely online technique impractical. On the other hand, many implementations of both online and offline profiling have required specialized hardware or software [2,15,17]; by using data gathered by Nagpurkar et al., we inherit their dependencies. Thus, we are not constrained to specialized hardware, architecture specific metrics, or a specific optimization client.

Given an observation sequence, (again, a model observation consists of a processed window of symbols) the HMM can optimize the model parameters via the Baum-Welch algorithm. As stated, the number of hidden states in an HMM must be specified. From Sherwood, we know that 32 phases (states) can account for 90% of the program trace [17]. However, we opted to search over a wide range of states (two to 64, incremented on a log scale). Beyond this parameter, the K-means algorithm requires that the number of clusters must be specified, (again, this translates to the number of code words in the language recognized by the VQ-HMM). We selected the number of clusters out of the range two to 128 (incremented on a log scale). In the case of the CD-HMM, the number of principle components was varied from seven to 25 and the number of mixtures was varied from one to ten. In order to detect a phase, the phase must be larger than the window size; thus, two further parameters exists, the window size, and the skip size. Again, a window corresponds to the granularity at which the instruction trace is vectorized. We adhered to the recommendation made in [13] and considered window sizes from 500 up to 100,000 with skip sizes of half the window size or equal to the window size.

Results for the HMMs were gathered on two variations of the datasets defined above. Both HMMs were trained on the full instruction traces from the datasets and on a training/testing set generated from the full instruction trace. Results for the full instruction trace were gathered to evaluate how well the HMM was able to fit the observation sequence; in essence, how much variance can be accounted for by the language and state probability transition matrices. Results for the HMMs were also gathered from "separate" training and testing sequences. Although choosing model parameters via N Fold cross validation was not possible (due to the limited nature of our datasets), a measure of each model's performance on future data was obtained. This gauges the HMMs susceptibility to both over-fitting and local optimum.

One problem with generating a training/testing set lies in the fact that there is no clear way to divide the full dataset in half; this is a consequence of the signal nature of the data. Random partitioning of the dataset is insufficient to ensure that the cues picked out of the training set are represented in the test set. Instead, we settled for a relatively straightforward method of random sampling. This technique randomly ushers each symbol into either the training or the testing set, while maintaining relative order.

Given that an observation is defined by a distribution over symbols, then a random sampling from the distribution will maintain the relative distribution (assuming a lot of samples are taken), and is thus sufficient to maintain the integrity of observations made in a phase. In the case of a transition, the only property that need be maintained is that any information remain unrelated to the preceding phase, or be beneath the observable granularity of the model. Thus, the information in an observation is reasonably well maintained across the distinct training and testing sequences.

An ideal baseline does not exist to determine the accuracy of our phase detection algorithm; however, we use the baseline defined by Nagpurkar et al. [13] as a metric for comparison. This baseline distinguishes between the sections of an execution trace that are in-phase (i.e. a period with expected behavior) and in-transition (a period with unexpected behavior). An HMM, however, generates the most likely state sequence given an observation sequence. As such, there is not a characteristic 'transitional state'. Instead, we compared where our algorithm identified phase boundaries to those identified by the baseline. This simply involved labeling a state as in-phase ($q(t) = q(t-1)$) or in-transition ($q(t) \neq q(t-1)$), depending on whether or not the current state is identical to the previous one, respectively. Total accuracy is computed from this correspondence[2]. Note: when gauging the accuracy of a model, it must be analyzed with respect to the prior distribution of in-phase or in-transition labels in the execution trace. This is analogous to comparing the accuracy of one of our models to that of a naïve model defined simply to return only the most frequent mode (in phase or in transition) of an execution trace. We will refer to this naïve model of comparison as the *prior-model*.

[2] Note: we may also consider other baseline metrics, such as the metric proposed by Sweeney et al., or a metric that treats transitions and regular regions as *events*, not on a slice-by-slice basis.

The likelihood of the state sequence given the observation sequence and the model ($P(Q \mid O, \lambda)$) is also used as a measure of how well the model predicts the data (i.e. how well λ approximates f(x)). As a generative model, the HMM can then be used to predict future trace behavior.

For further assessment, we also compare the results of the HMM with those of simple K-means. K-means does not capture information about phase transitions, i.e., a static mode that classifies each segment independent of the others. A comparison of these models reveals what, if anything, is gained from the HMM's phase transition information.

4 Results

4.1 The Prior-Model

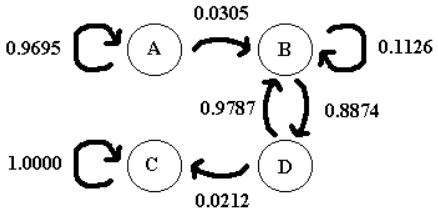
The prior distribution of observations labeled as in-phase or in-transition is highly dependent on the dataset and minimum phase length. This is of immediate importance for the analysis of our models because it affects the accuracy of the prior-model. The percent of a program trace that is labeled as in-phase (equivalently, the accuracy of the prior-model) varies by as much as 70% across the datasets for each of the given minimum phase lengths. Perhaps more salient than that phase behavior varies across programs is that given a dataset, the choice of minimum phase length significantly affects the percentage of a program trace that is considered in-phase (or in-transition accordingly). For instance, the dataset 202_jess with natural phase lengths of 100,000 and 1,000 give corresponding prior-model accuracies of 43% and 91%. The accuracy of the prior-model provides the context in which to gauge how much information our models are discovering (see Discussion for possible problems with this metric). All subsequent references to *relative error* are made with respect to the error of the corresponding prior-model.

4.2 The VQ-HMM

Analyzing the model's proficiency at distinguishing one phase from another reveals that the model space is highly non-convex. For instance, a large subset of model parameters has the ability to produce models that out-perform the prior-model significantly; however, models retrained from this subset do not consistently perform well (due to local optimum). Furthermore, while a few sets of VQ-HMM parameters consistently reduce the relative error between 5% and 30%, these parameters are highly dependent on dataset and minimum phase length, and only represent a small fraction of the total model space.

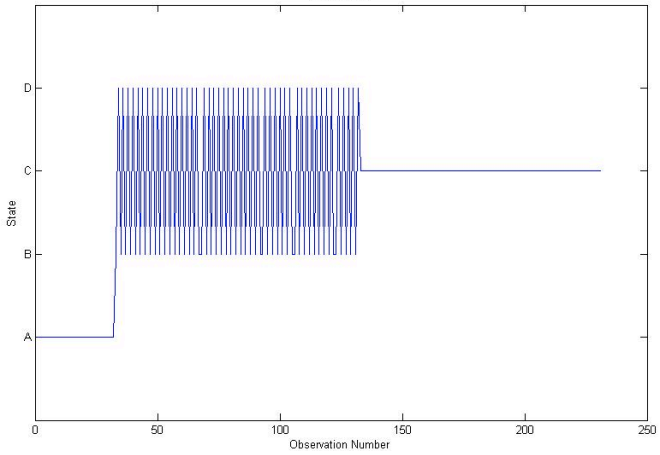
More fruitful is an analysis of global trends found by examining the *best* models (those with the greatest improvement in accuracy) for a given set of model parameters. A comparison between best models is effectively a survey of the upper bound on the expected performance increase for a given subset of model parameters. The best VQ-HMM models are generally those with two to 16 states. Of the best models, the number of states and the number of clusters are highly correlated, i.e. the number of clusters is generally within a factor of four of the number of states. The best VQ-HMM models reduce relative error by roughly 20%. For a dataset that has a well performing prior model, such as 209_db, this corresponds to an increase in accuracy from 91% to 99%. For a data set that has a poor performing prior, such as 205_raytrace, this corresponds to an increase in accuracy from 53% to 85%. 204_jess, 213_javac, and 209_db have the greatest reduction in relative error when the window size and minimum phase length are each either 10,000 or 25,000. 205_raytrace has similar optima when the window size and minimum phase lengths are each either 50,000 or 100,000.

Figure 1: State Transition Diagram



Figures 3 and 4 graphically represent the state transition matrix and most probable state sequence for a model trained with 205_raytrace. Probabilities not shown are 0. The model was created using 8 clusters, 4 states, a window size of 50000, and a skip size of 25000. The minimum phase length was 50000.

Figure 2: State Graph



The estimated state of a program (Y-axis) at the observation number specified along the X-axis

Table 1		Probability of observing (emitting) cluster codewords in a state							
		#s	#t	#u	#v	#w	#x	#y	#z
State	A	0	0	0	0	0.4716	0	0.0943	0.4341
	B	0	0.4027	0.0607	0.2888	0	0.147	0	0.1008
	C	1	0	0	0	0	0	0	0
	D	0	0.2254	0.1651	0.0778	0	0.4287	0	0.1029

Table 1: The emission matrix corresponding to the model used in figures 3 and 4.

Figures 1 and 2 graphically display the information contained in the state transition matrix and the most probable state sequence for a model trained over the dataset 205_raytrace. Table 1 displays the information in the model's language emission matrix. The model was created using 8 clusters, 4 states, a window size of 50,000, a skip size of 25,000, and an assumed minimum phase length of 50,000. The model has a relative error reduction of 77%, an overall accuracy of 89%. States A and C have a high probability of reflexive transition, while state B and D have a high probability of transitioning back and fourth between each other.

4.3 The CD-HMM

Although a few sets of CD-HMM model parameters are able to reduce the relative error, these are highly dataset specific. Also, the variance of total accuracy over the model parameters is extreme, and a majority of the models have a total accuracy of within a few percentage points of the prior-models'. Again however, trends can be observed if focus is turned to the best models. The CD-HMM tends to create better models when more principal components are used. As the number of principle components increases (i.e. as the amount of variability included in the observations increases), a larger number of mixtures is needed to fit the observations well. The numbers of states tends to contribute significantly to model performance. The best models built with a low number of states (typically 2 - 8) reduce the relative error by about four times as much as the best models built using a larger number of states (greater than 8). However, the variance of the models built with fewer states is about four times greater than that of the models built with a higher number of states. This effect is consistent across all data sets. The highest upper bound on relative error reduction is 36% for the dataset 205_raytrace. The average relative error reduction across all datasets is 21%. Again, these results are generated from the *best* models and represent an upper bound for possible error reduction attainable by this method.

4.4 The HMMs as Predictors

How closely a model fits the data, or equivalently, the predictive power of the generative model, can be summarized by the likelihood of a testing observation sequence given a model ($\log\_PSQ = P(O_T \mid \lambda)$). Theoretically, the model with the log_PSQ closest to zero most closely approximates the process by which the sequence was

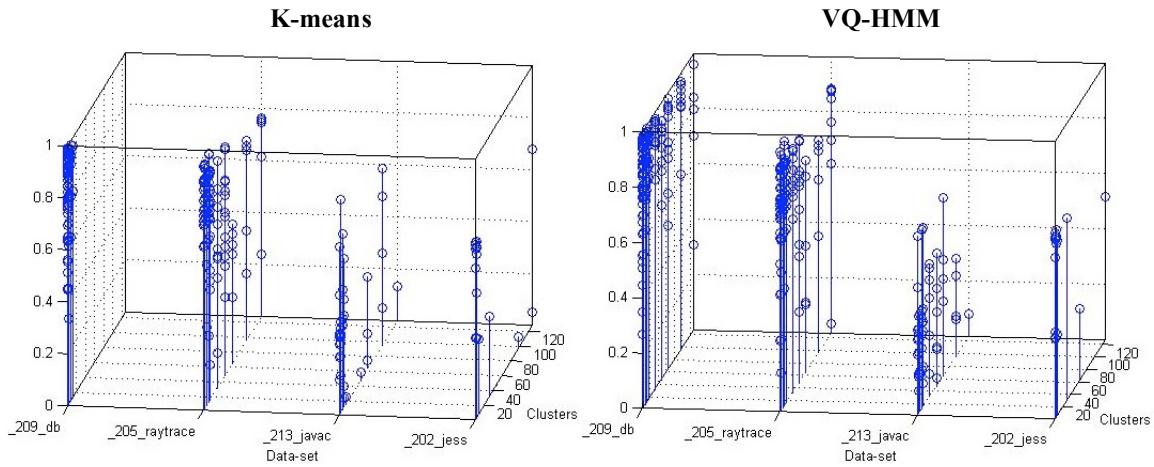


Figure 3: Reduction in error relative to the prior model. Y-axis: Data sets. X-axis: Clusters. Note that the performance between K-means and the VQ-HMM is nearly identical for clusters two through 12.

generated. This involves striking a balance between number of states, number of clusters/principle components, number of Gaussians, etc. in order to avoid over fitting while maintaining generality. The variability between log_PSQ of identically trained models is small (less than 20%), even when the reduction in relative error varied significantly (by more than 50%). Whether or not predictions made by models that have small log_PSQ are accurate remains unknown; this could be determined by testing in an embedded system where an appropriate cost function is defined.

4.5 The K-means Algorithm

The K-means clustering algorithm was also applied to the datasets. Unexpectedly, the accuracy attained by the simple K-means algorithm is almost identical to that of the VQ-HMM. This is illustrated by the similarity of the plots for K-means and the VQ-HMM in Figure 3. Further, K-means is actually far more likely than the CD-HMM to find a consistently good model. For instance, 205_raytrace was run over 25 independently trained K-means models (each containing 16 clusters, a skip size of 12,500, and a window size of 25,000) and found to decrease the amount of relative error by a mean 27%, with a standard deviation of 12%. An HMM trained with identical parameters and 16 states had nearly identical results. However, when a similar HMM was trained using 8 states, the mean decrease in relative error dropped to 8% and had a standard deviation of 30%.

5 Discussion and Conclusions

One of the most striking results is that the K-means algorithm performs comparably to the VQ-HMM. This begs the question: are the few *consistently* good models created by the VQ-HMM attaining high performance simply because K-means is used as a preprocessing step? The data suggests that this may be the case. In a pure transition

1	C	C	C	C	C	C	C	C	C	B	B	B	B	B	B	B	B	B	D
2	D	D	D	D	D	D	D	D	D	C	C	C	C	C	C	C	C	C	C
3	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	A
4	B	D	B	D	B	D	B	D	B	C	C	C	C	C	C	C	C	C	A
5	D	B	D	B	D	B	D	B	D	C	C	C	C	C	C	C	C	C	C
6	C	C	C	C	C	C	C	C	C	D	D	D	D	D	D	D	D	D	D

Figure 4: A slice of the predicted state seqence of six independently trained identical VQ-HMM models for the dataset 205_raytrace. Note that the character labels are not consistent between models (i.e. state A in model 1 is not necessarily the same as state A in model 2). Sequence 1, 2, and 6 increased the relative error by about 25% and model 3 by 27%. Models 4 and 5 decrease the relative error by about 55%.

detection scheme it may be advisable to run K-means by itself, while an HMM may be more useful as a profiling tool. This also suggests that K-means could be used to quickly search the parameter space for sets of parameters that consistently build well performing models. These could then be used to constrain the HMM's parameter search to the number of states. In any case, the HMM is capable of phase prediction (i.e. given the current state, what is the next most likely state), while the K-means algorithm is not.

It is worth mentioning that any interval-based algorithm is highly dependent on the disparity between the granularity at which phase behavior is detected (the minimum phase length or window size) and that at which it most naturally exists. In the model space, the most appropriate window size can be quantitatively selected based on the correspondence between the predicted state sequence and a given baseline. However, our baseline also risks missing the structure of the data because it must assume a particular minimum phase length. However, choosing the correct baseline for any program is dubious because it is impossible to distinguish between a baseline that was extracted from a program containing little structure and one that was analyzed with a poorly fitted minimum phase length. The consequence of this is that the accuracy of the prior-model may be artificially high[3]. Hence, bad relative accuracy could be attributed to either a bad model or a poor choice of baseline.

It follows that the predictive ability of the model is also contingent on the choice of the baseline.

Not only does the correspondence between natural phase length and minimum phase length affect accuracy, but also, as noted above, the granularity at which our models detect phases can affect accuracy as well. This is a consequence of the relationship between window size and natural phase length. As illustrated by Figure 4, the basic structure of good models (4 and 5) and bad models (1, 2, and 6), i.e. those that decrease or increase relative error, respectively, is close for the given set of model parameters. Note, however, that the main difference between the good and bad models is that the good models pick up on a granularity of structure that the bad models do not. Specifically, the good models identify two alternating phases, while the bad models lump them into a single phase. Our algorithms obviously perform best when the natural phase length of the dataset has the same order of magnitude as the chosen window size.

The VQ-HMM consistently performs better than the CD-HMM in this problem domain, given their respective search spaces. Although, neither model has a consistent set of learning parameters that perform well across all datasets, the VQ-HMM is able to repeatedly perform well as long as the learning parameters are specifically tailored to a given dataset.

A major limitation of our implementation is that our models are highly sensitive to the minimum phase length and the window size. Without some prior knowledge, their selection is essentially reduced to a search problem. Good initial estimates for the transition probability matrices are also essential in order to converge on a good solution. Ultimately, the expected performance of the models will approach the upper bound as prior knowledge about the problem domain alleviates the model's propensity for local optimum.

The advantage of an HMM over simple K-means, or even the baseline algorithm, is that it can be used in a Semi-Online sense for phase prediction and as a learning tool. For instance, given the information contained in Figures 1 and 2 and Table 1, it is apparent that 205_raytrace experiences three distinct operational periods at the granularity defined by {window size = 50,000, skip size = 25000, minimum phase length = 50,000}. Further, it is also apparent that having 4 states (vs. 3) or having a minimum natural phase length of 50,000 may introduce undesired transitional behavior between observation numbers 40 and 140, for this particular dataset. During this period there is high transition between states B and D. Note, however, that the rows corresponding states B and D in the emission matrix are very similar. The subset of codewords that they emit (i.e. those with a probability greater than 0) is identical. The only difference between the two states is slightly different codeword emission probabilities. It is possible that reducing the number of states to 3 would cause B and D to recombine into a single state. Also, because the program transitions between the two states on nearly every observation, increasing the minimum phase length may eliminate the transition behavior.

---

[3] In cases when minimum phase length is much larger than the natural phase length, much of the program structure is discarded by the baseline (only structure larger than the minimum phase length can be observed). In this case, a prior-model that assumes the entire program is in-phase may have a high degree of accuracy because transitional features of the program are discarded. Conversely, if the minimum phase length is much smaller than the natural phase length, a prior-model that assumes constant transitions could have a high degree of accuracy if the baseline is unable to distinguish the structure from the noise.

Future Work

Any future work in this area should have multiple runs of the datasets. Although this is obviously necessary for validating and testing the models, it is also particularly important to training the HMM because model parameters can be more accurately initialized given more training sequences.

Due to the success of the simple K-means clustering algorithm, an obvious question is: Are there other clustering algorithms that could perform even better?

The information provided by the state transition and language emission probability matrices could also be used as tool for learning about the relationship between program phases, and the different granularities of program structure (although, an accurate model is a necessary prerequisite of this goal).

In order to not pick up on phase transitions, i.e. sections of the execution trace in which no structure can be picked out, a noisy state could be incorporated into the HMM. Outlier detection might also be applied with the same idea in mind.

The standard HMM can also be extended into a hierarchical model in which information from multiple granularities of the program are combined. This may help address the lack of prior knowledge regarding the minimum phase length and window size.

Acknowledgements

We would like to thank Mike Mozer and the Advanced Machine Learning class at the University of Colorado at Boulder for their input on this project. We would like to thank Peter Sweeney for providing us with program-trace data. We would also like to thank the University of Colorado at Boulder Computer Science Center through which computer time was provided by NSF ARI Grant #CDA-9601817, NSF MRI Grant #CNS-0420873, NASA AIST grant #NAG2-1646, DOE SciDAC grant #DE-FG02-04ER63870, NSF sponsorship of the National Center for Atmospheric Research, and a grant from the IBM Shared University Research (SUR) program.

References

[1] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5):1–12, May 2000. Published as part of the proceedings of PLDI'00.

[2] A. S. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In 29th Annual International Symposium on Computer Architecture, pages 233–244, May 2002.

[3] A. S. Dhodapkar and J. E. Smith. Comparing program phase detection techniques. In *the 36th International Symposium on Microarchitecture*, pages 217–227, Dec. 2003.

[4] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In International Conference on Parallel Achitecture and Compilation Techniques, Sept. 2003.

[5] M. Hind, V. Rajan, and P. F. Sweeney. Phase detection: A problem classification. Technical Report 22887, IBM Research, Aug. 2003.

[6] I. Jollife. *Principle Component Analysis*. Springer-Varleg. New York, 1986.

[7] T. Kistler and M. Franz. Continuous program optimization: A case study. *ACM Transactions on Programming Languages and Systems*, 25(4):500–548, July 2003.

[8] A Krogh. An introduction to hidden Markov models for biological sequences. In *Computational Methods in Molecular Biology*, edited by S. L. Salzberg, D. B Searls and S. Kasif; 45-63. Elsevier, 1998.

[9] J. Lau, E. Perelman, and B. Calder. Selecting software phase markers with code structure analysis. In *ACM Conference on Code Generation and Optimization*, Mar. 2006. 36(11):180–195, Nov. 2001.

[10] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1:281-297.

[11] A. Madison and A. P. Batson. Characteristics of program localities. Communications of the ACM, 19(5):285–294, May 1976.

[12] K. Murphy. HMM Toolbox for MATLAB. http://www.cs.ubc.ca/~murphyk/Software/HMM/hmm.html

[13] P. Nagpurkar, M. Hind, C Krintz, P. Sweeney, V.T. Rajan, Online Phase Detection Algorithms. In *International Symposium on Code Generation and Optimization,* Mar. 2006.

[14] L. R. Rabiner. A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition. In *IEEE*, vol. 77, no. 2, February 1989.

[15] X. Shen, Y. Zhong, and C. Ding. *Locality phase prediction*. In *Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2004.

[16] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[17] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *30th Annual International Symposium on Computer Architecture*, pages 336–349, June 2003.