Tools for Detecting Dependencies in AI Systems

Matthew D. Schmill, Tim Oates and Paul R. Cohen
Computer Science Dept., LGRC
University of Massachusetts
Box 34610
Amherst, MA 01003-4610
{schmill, oates, cohen}@cs.umass.edu
Contact Author: Matthew D. Schmill
(413) 545-3638, 413-545-1249 (fax)

Abstract

We present a methodology for learning complex dependencies in data based on streams of categorical, time series data. The streams representation is applicable in a variety of situations: a program's execution trace may be thought of as a stream. The various monitor readings of an intensive care unit may be thought of as concurrent streams. Our learning methodology, called ***dependency detection***, examines a stream or multiple streams to characterize recurring structure with a set of ***dependency rules***. These dependency rules are useful not only as a description of how the data is structured, but as a means for predicting future stream states from those of the present. Further, we describe a set of tools for program analysis that use dependency detection.

To appear in *Proceedings of the Seventh International IEEE Conference on Tools with Artificial Intelligence.*

1 Introduction

Many dynamic situations can be represented as *streams* or time series of tokens. In particular, the behavior of a computer program over time might be encoded as an execution trace: a single stream containing the program's current state. Similarly, an intensive care unit might be modelled as many streams, each containing a particular life sign or monitor reading. Given a system that can be represented by one or more such token streams, our aim is to learn rules that predict the state of the system – the token values in all streams at a particular time – from the stream values at earlier states. Such predictive rules can serve a variety of purposes, from providing insight into program behavior to forming the basis of an automated intensive care monitor.

In this paper, we present a set of tools called *dependency detection* algorithms that address the problem of predicting future states from prior ones. The first in the series of dependency detection (DD) algorithms, is a technique for identifying situations that lead to failure in AI planning systems. The procedure, termed *failure recovery analysis* (FRA), models a program as a single stream (the execution trace), and uses a statistical test [1] to locate contributors to plan failure. [4] The benefit of using FRA as a debugging tool for planning systems is due largely to its generality. Basing its diagnosis solely on observed patterns in execution traces allows the dependency detection algorithm to operate with little domain knowledge and only a weak model of how the actual system may cause its own failure.

Multi-stream Dependency Detection, or MSDD, extends dependency detection to characterize structure across multiple, concurrent streams. [6] Like the DD component to FRA, MSDD uses statistical testing [2] to identify patterns of stream values that predict patterns occurring in the future. MSDD examines batches of time series data, searching in a focused manner for recurring structure that it summarizes in the form of *dependency rules*. These dependency rules characterize the regularity in the data they refer to and are general in the sense that stream values which play no role in prediction are ignored. These rules can be used to make predictions in future data, as well as describe the nature of the system which produces the data. We return to the discussion of MSDD in section 3.1.

The latest development in the series of dependency detection algorithms is called IMSDD. IMSDD extends MSDD to work in an incremental fashion. That is, IMSDD builds its set of multi-stream dependency rules *as the data is produced*, without requiring off-line batch processing time. The result is not necessarily a faster or more accurate algorithm, but an algorithm that has the capability to learn as data becomes available, making it viable as a concurrent, embedded learning component in a much larger system.

The remainder of this paper is devoted to discussion of the multi-stream dependency detection technique, its application, and the current state of its implementation as a research tool. After a glossary of terms associated with dependency detection, we offer a short description of MSDD and an example application. Next, we consider IMSDD in detail, describing its search for dependencies and its performance on that task. We follow with a discussion of the current integration of the algorithms into CLIP/CLASP, [1] an environment for empirical studies, then conclude with a brief summary of the value of dependency detection.

---

[1] A statistic that quantifies statistical significance from observed frequencies called the *G Test* was used.

[2] MSDD uses a simple contingency table statistic to determine significance.

2 Definitions

Operating in the world of dependency detection requires a few definitions and representations to keep the concepts clear. We will adopt the following terminology for the remainder of this paper.

- A *stream* is a source of categorical data values that change over time, such as a program's execution trace. We denote a stream by its values over time, for example *aabac*.
- A *token* t is a datum in a stream. We assume the tokens in streams are from a relatively small (discrete) alphabet, instead of real numbers. [3]For example, a is the first token in the stream *aabac*. The set of all possible tokens is our *alphabet*, which is of size k.
- A *multitoken* is a vector of all streams' values at a particular point in time. We represent a multitoken as a list of tokens $(t_1 t_2 \ldots t_n)$ and denote the length of a multitoken by n. Suppose we have streams *adbc* and *caac*. The second multitoken is (da).
- A *rule* is a pair of multitokens, one of which predicts the other. We denote a rule by *precursor* $\rightarrow$ *successor*. For the purposes of this paper, all predictions are of *lag 1*. That is, the successor occurs in the timestep directly following the precursor.
- A *word* is a rule presented as a single entity. We represent a word with the form $\langle t_1 t_2 \ldots t_w \rangle$, and denote the length of a word by n_w.
- A *wildcard* is a token that indicates that a particular stream value is to be ignored. A wildcard is denoted $*$.

3 Detecting Dependencies in Multiple Streams

Characterizing the structure present in multi-stream data can be thought of as a search problem. Given a system that generates stream data and a space of possible rules to describe the dynamics of the system, a dependency detection algorithm searches for rules which are supported in the stream data. MSDD and IMSDD offer two procedures for guiding this search.

3.1 The MSDD Approach

The MSDD algorithm takes a top-down approach to building a base of dependency rules. Starting with the completely general rule $(* * \ldots *) \rightarrow (* * \ldots *)$ as a root of a *generalization hierarchy*, MSDD iteratively expands that hierarchy by instantiating a single wildcard of a leaf rule at a time. Figure 1 shows an example MSDD generalization hierarchy. Each iteration of the search for predictive rules extends the frontier of the tree a single level until a predetermined size limit has been reached. At that point, MSDD concludes its training session.

MSDD accrues several benefits from this approach. Because it considers a large batch of data simultaneously, any rule that MSDD proposes can be evaluated and either rejected or accepted immediately. This allows MSDD to explore only fruitful paths in its search space, and results in a succinct, powerful model for learning rules. In section 3.3, we will see why these benefits are important and why an incremental approach lacks them.

[3]Continuous valued time-series data is appropriate to many domains and is discussed in [3, 2, 7].

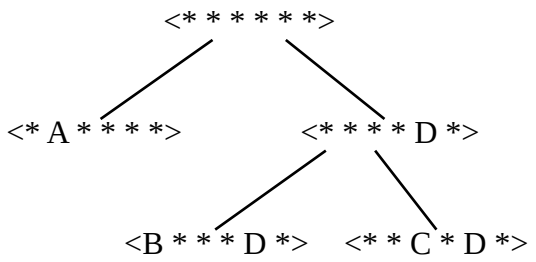

Figure 1: A simple MSDD generalization hierarchy.

3.2 MSDD as a Learning Component

The usefulness of the multi-stream representation was first tested in a simulated shipping network called TransSim. TransSim is a system designed for the task of schedule maintenance in complex, unpredictable environments. [5] The system comprises two main components: a *pathology demon* and a *schedule maintenance agent*. The pathology demon monitors the simulation environment, as it runs, in an effort to detect states that may lead to degradation of plan performance, or even plan failure. When such a state is detected, the schedule maintenance agent devises a schedule modification to alleviate or avoid the possible pathology.

Our goal was to replace the pathology demon, along with its use of detailed domain knowledge, with a set of rules generated by MSDD after examining batches of data produced in previous trials of TransSim. We modelled the execution as a set of status streams for the TransSim ship ports: the number of incoming ships, length of the docking queue, and so on. Our hope was that MSDD could do automatically what a programmer or domain expert often takes many hours to do: devise a set of rules that accurately describe situations of interest in a system.

The results, summarized in [6], indicate that MSDD's rules were capable of performance comparable to that of the hand crafted pathology demon using only high-level domain knowledge. In fact, the MSDD system predicted less pathologies and made fewer schedule changes, while achieving performance almost indistinguishable from the pathology demon's in all but one of the cost metrics associated with the shipping simulation. [4]

3.3 The IMSDD Approach

Developing an incremental version of the MSDD algorithm serves two purposes. First, an incremental MSDD algorithm need do no off-line processing and is not dependent on the existence or quality of training data. Rather, the incremental algorithm does all processing as the data becomes available, making it suited for real-time use as an embedded learning component. For example, an incremental MSDD module could serve as a learning component in the pathology demon of TransSim. Second, an incremental algorithm gains the ability to adapt its rule base to account for structure that changes over time. Should the overall dynamics of how TransSim pathologies come about change, for example, an incremental algorithm would be able to gradually adjust its rule base to reflect the change.

The move to incremental processing is not without consequence to the algorithm. First and foremost, IMSDD is forced to abandon the generalization hierarchy exploited by MSDD. The loss of the training batch (and thus the ability to recount) makes it impossible to start with general

[4] The metric was SD, simulated days. The MSDD version took more simulated days to complete its shipping quota.

rules and work towards more specific rules; doing so would result in a disastrous, contextless search in IMSDD's $O(k^n)$ [5] generalization space. As a result, IMSDD must take a data-driven, bottom-up approach to forming rules. As IMSDD receives input, it stores multitoken pairs as fully instantiated words. Tokens in streams that exhibit no contingent structure are *generalized* as a move towards representing the data's true structure. Such *noisy* streams in generalized rules are given wildcards for their values to indicate that they should be ignored.

The basic operation of IMSDD follows a $predict \rightarrow verify \rightarrow generalize \rightarrow update$ loop. Based on the current input (multitoken), IMSDD predicts what the next multitoken will contain, evaluates that prediction, uses the input to form new generalizations, and then updates its internal data structures.

At this point it is worth considering the worst-case complexities of the predict, verify, generalize, update loop. For a given word w of length n_w, there are 2^{n_w} possible generalizations. Without intelligent control, we can expect a worst case of $O(2^{n_w})$ for each of the phases except verify[6]. In i timesteps, a naive algorithm may approach $O(i2^{n_w})$ and generate the entire search space of $O((k+1)^{n_w})$.[7] The focal point of the IMSDD algorithm is reducing the expected complexity of this search.

4 IMSDD in Detail

The actual process of guiding a dependency detection algorithm through its exponential search space is worth examining in detail. We next describe the individual components to the youngest of the algorithms, IMSDD, and it's predict, verify, generalize, update cycle. Further, we add a *pruning* step to the cycle and evaluate the algorithm's performance in initial experiments.

4.1 The IMSDD Memory Structure

The first step to combating the combinatorics of the dependency detection algorithm is defining a memory structure that facilitates efficient storage and retrieval of rules. With an $O((k+1)^{n_w})$ search space, a dependency detection algorithm needs a data structure that inherently limits focus to at most the 2^{n_w} generalizations that are actually relevant to any given input (and ideally, far fewer). IMSDD makes use of a structure that resembles a parse tree for the precursor of a rule, aptly named the *precursor tree*. Figure 2 shows an illustrative parsing of the precursor portion of $\langle abac \rangle$, (ab). Starting with the token a at the root, IMSDD parses the precursor by moving down the branch that corresponds to the current token, and moves on to the next token, which in the example is b. By the time IMSDD reaches a leaf in the structure, IMSDD has fully parsed the precursor multitoken of a rule. Information about what the precursor predicts is stored in the leaf as a *successor table*. Each row of a successor table represents a stream in the successor, and keeps a history of token frequencies in each of the successor streams. For example, in figure 2, row one of the successor table has recorded that an a in stream one has followed (ab) twelve times.

This structure narrows IMSDD's focus for a given precursor to 2^n rules by storing precursor information as paths in a tree and recording the successors as tables that require a simple $O(n)$ operation to query. It also provides a simple means for keeping successor counts current with

[5] A multitoken may take on k values over each of n streams.

[6] Which is a simple $O(n)$ comparison.

[7] The complexity derivations are quite simple. The 2^{n_w} cases involve all possible wildcard combinations with a fully instantiated word, and $(k+1)^{n_w}$ is the full set of n_w length words with k possible tokens plus the wildcard token.

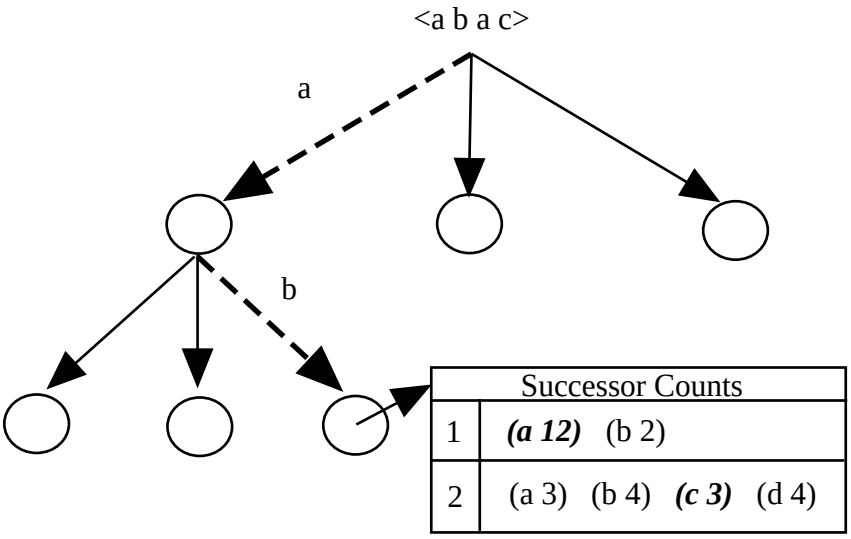

Figure 2: An IMSDD precursor tree and the parsing of $\langle abac \rangle$.

respect to a particular precursor, which will become important in the prediction phase. We now turn to the procedures of the algorithm to see how the focus can be tightened further.

4.2 Predicting and Verifying

Predict and verify are perhaps the simplest phases of the IMSDD cycle. Given a multitoken, IMSDD's goal is to predict the next multitoken, based on the contents of the relevant successor tables in the precursor tree. Once a prediction has been made, its prediction is verified against the observed successor. Since the verification process is a trivial token-by-token comparison, we turn to a closer examination of the prediction scheme.

Suppose IMSDD observes the multitoken p (e.g. $p = (ab)$ as in figure 2). To predict its successor, IMSDD parses p through the parse tree to generate the set of all leaves whose path matches p.[8] We call this set $\mathcal{S}$, and observe that it contains all of the possible successor tables for p supported by data we have already seen. Each successor table in $\mathcal{S}$ *suggests* a unique successor s based on the frequency information it contains. This is accomplished by choosing the "best" token t from each of the rows in the successor table as rated by the function S:

$$S(t) = \alpha(N_{Hits}(t)) - (1 - \alpha)(N_{FP}(t))$$

where α is the *aggressiveness coefficient* of the algorithm, or the extent to which it values hits versus false positives, N_{Hits} is the number of times t has occurred in the successor table row, and N_{FP} is the number of times t did not occur in the successor table row. For each row j of the successor table, IMSDD suggests a token t_j which maximizes S. The result is the complete successor multitoken s. As an example, given the precursor (ab) in figure 2, IMSDD would suggest either (ab) or (ad), since a clearly dominates row one, while b and d have matching counts in row two.

It is now possible to think of $\mathcal{S}$ as a set of suggested successors to p. Next IMSDD will need to rate each complete successor in $\mathcal{S}$ to decide which is the best to apply. For this purpose, we define S_{total} on successor multitoken s :

[8] At first, it may seem that this set would contain a single leaf. This would be the case were it not for precursor generalizations, which are considered in section 4.3.2.

$$S_{total} = \alpha \sum_{i=0}^{|s|} (N_{Hits}(t_i)) - (1-\alpha) \sum_{i=0}^{|s|} (N_{FP}(t_i))$$

All that remains is to choose the successor in $\mathcal{S}$ that maximizes S_{total}.

4.3 Making Generalizations

Intelligent control when forming generalizations (deciding that a stream should be ignored in a rule) is perhaps the single most important aspect of the IMSDD problem. Because the generalization space is exponential, the brunt of our effort in developing IMSDD was devoted to addressing this problem. As in the counting scheme, precursors and successors are treated differently in the generalization scheme. We will look first at the simpler case of forming generalizations in the successor portion of a rule.

4.3.1 Successor Generalizations

The impetus for successor generalization is successor data that provides little predictive power. IMSDD uses the layout of the successor table to decide when generalization is beneficial. Based solely on the distribution of token frequencies in the rows of a successor table, IMSDD can decide whether it is best to make a prediction or abstain from making one (i.e., predict a wildcard).

Upon startup, IMSDD assigns a constant S value, τ, to the wildcard token. This value is based on the aggressiveness of the algorithm, α, and the probability of correctly guessing a token by chance. Each time IMSDD attempts to predict a stream value, it considers a wildcard token along with all those tokens that were actually observed to occur. Should no token's S value be high enough (greater than τ) for IMSDD to believe it did not occur by chance, the algorithm will suggest a wildcard for that stream.

The result of this simple scheme is a substantial saving in time and space requirements. However, it relies on an important assumption. By storing only token frequencies in the rows of a table rather than complete multitokens, IMSDD assumes that streams within a successor are independent, or because they are dependent, the dependent stream values will always occur together, and thus either all of them will dominate the successor table, or none will. The benefits of this assumption are a succinct means of recording successor frequency, and a generalization mechanism that requires little extra time and space of the algorithm.

4.3.2 Precursor Generalizations

Generalizations in the precursor of a rule come at a somewhat higher cost. The reason is that successor tables are only valid for a single precursor, and so when IMSDD generalizes, it must have a unique successor table for the new, generalized precursor. The result is a procedure that generates precursor generalizations opportunistically, and creates a new path in the precursor tree to represent the generalized precursor. Figure 3 shows the path $(*b)$ in an illustrative precursor tree, along with a successor table that is the composition of two successor tables whose precursors match the new generalization.

The greatest challenge in generating precursor generalizations is deciding when to make a generalization, because there are $O(2^n)$ possibilities and no easy answers as there are with successor generalizations. To control the explosiveness of the generalization routine, IMSDD constrains precursor generalizations in the following ways :

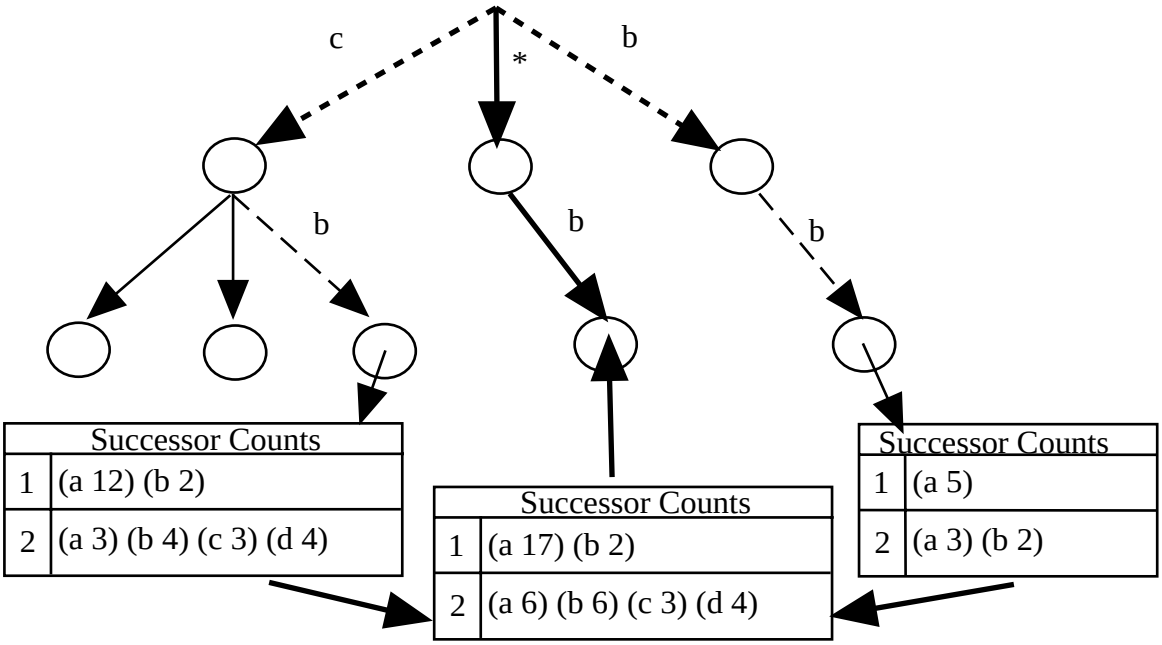

Figure 3: Forming a precursor generalization ($*b$) from (cb) and (bb).

- A precursor path must exist in the precursor tree which differs from the current precursor in exactly one stream (any token matches a wildcard).
- The successors to the nearly-matching precursors must also match or nearly match.

The first of these constraints is implemented in the search for matching precursors. A mechanism called the *wrong turn procedure* locates all matches to a precursor that are off by one token. The wrong turn procedure parses the new precursor into the precursor tree, and at each level of the parse, considers what would result if the token at that level was parsed *incorrectly*. When the algorithm is left to run its course, it will return precisely those leaves in the tree whose paths differ from the new precursor by exactly one stream value.

In addition, IMSDD offers another speedup option. Instead of considering all wrong turns at each level of the parse, IMSDD samples randomly some proportion of the possible wrong turns. Here, IMSDD operates under the assumption that a uniform distribution of stream values (which indicate that a generalization is going to be useful) will ensure that all important matches will be manifest at least once through random sampling.

Once the matching precursors have been found, IMSDD has a set of plausible generalizations and the actual precursors which support them. A generalization will be accepted as good if all those precursors which match it suggest similar successors. The new generalization will then be recorded as a path in the precursor tree. At the end of the path, as in figure 3, a new successor table is formed, and the supporting precursors' successor tables are combined to fill in the token frequencies.

As the number of streams presented to IMSDD increases, it becomes increasingly unlikely that two successor tables will be similar, especially if some of the streams are noisy. This causes precursor generalizations to fail on the matching-successor constraint. Consequently, IMSDD could run for many timesteps before having the opportunity to make a single good generalization. IMSDD offers as a satisficing solution a mechanism that resembles *simulated annealing*. When IMSDD is filtering

the possible generalizations, it will with some probability γ, which diminishes with time, *trivially accept* each one whether it meets the constraints or not. In effect, this property overcomes the bootstrapping problem by agitating the rule base early on until there are some good generalizations. These good generalizations will have higher counts in the successor that can enable noisy streams to be identified, and the bad generalizations can be removed by the pruning mechanism, discussed in section 4.5.

4.4 Updating

Counting successor frequencies is vital to correctly quantifying the predictive accuracy of a rule. When IMSDD observes a word, it parses the precursor portion in its precursor tree, finding all paths that the precursor matches. For example, the precursor (bb) would match $(*b)$ and (bb) in the tree of figure 3. In each path's successor table, the token count for each row is incremented according its corresponding stream value. Using the same example, if precursor (bb) was followed by successor (ad), the counts in the $(*b)$ successor table frequencies would change to $(a\ 18)$ in row one and $(d\ 5)$ in row two.

4.5 Pruning

The final component to the IMSDD algorithm is the pruning component. By pruning, IMSDD attempts to bound the search space to contain only those rules that have occurred recently or have proven useful. Rules are selected for pruning during the prediction phase. During IMSDD's selection process to find the best successor to predict, the pruning component selects a fixed number of the worst rated rules. These rules, unless used under different circumstances within a certain period of time, will be pruned.

4.6 Empirical Evaluation

Because of the potential complexity of dependency detection, IMSDD was under constant evaluation to determine the impact of design decisions on its performance. For the purposes of testing, a system for creating artificially structured datasets was developed. For a given number of streams, ASG, the artificial data generator, produces a set of general rules, and generates a series of multitokens, some containing noise, and some containing actual structure. We define the metrics *adjusted hit rate (ahr)* to be the number of correct token predictions divided by the number of tokens seeded in the dataset,[9] and *fp-rate (fpr)* to be the number of incorrectly predicted tokens divided by the total number of tokens.

Figure 4 shows IMSDD's learning curves for *ahr* and *fpr* on IMSDD experiments with varying parameters. The curves were produced by recording IMSDD's performance on a fixed test batch every 100 timesteps during the learning process.

The first mechanisms we examined were the pruning strategy and the sampling policy for precursor generalization. We recorded the learning curves generated by IMSDD as we varied the sampling rate and turned pruning on and off. Figure 4 a suggests that both pruning and sampling have small effects. The effects of pruning and sampling on false positive rate (not pictured) were somewhat surprising; both sampling and pruning led to slower increase rates and lower peaks in the curves[10], while the steady state of each curve was similar.

[9] Since IMSDD may correctly predict tokens that occur by chance, the *ahr* metric often exceeds 1.0.

[10] Some of the fpr curves rose sharply and then dropped.

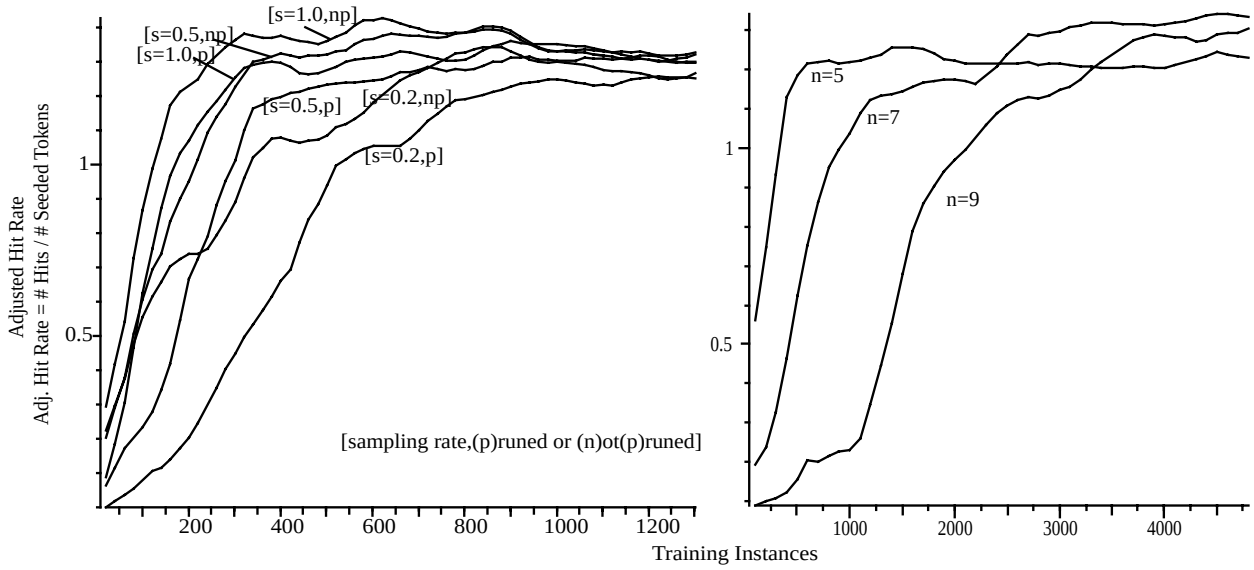

Figure 4: **(a)** Effects of sampling and pruning on learning curves. **(b)** Learning curves for $n = (5, 7, 9)$.

We next explored the effects of increasing n, the number of streams. Figure 4b shows the effect of increasing n from 5 to 9. First note that the overall slope of the learning curve is similar for all three values of n. Second, the onset of the learning curve and the point at which IMSDD can account for 100% of the structure differ are delayed as n increases. The first result suggests that the learning algorithm, when scaled up might exhibit the same facility for learning rules and accounting for structure. The second result implies that due to the larger stream size, there is some degree of difficulty learning good initial generalizations given the added dimensions in search space size.

While these results were encouraging, it remained to test IMSDD against its tried and true predecessor MSDD. Using identical data, we ran the 5 stream dataset through MSDD, varying the size of its training batch, to see how IMSDD compared. Examination of the *ahr* curve showed that MSDD was quicker to account for 100% of the present structure, but that IMSDD was not far behind. The *fpr* curve suggested that MSDD's false positive rate was somewhat lower at 0.23, and constant, while IMSDD's *fpr* curve appeared similar to its learning curve, scaled down to peak at 0.3.

5 Implementation

Both MSDD and IMSDD have been integrated into a single system for the empirical analysis of AI programs called CLIP/CLASP. CLASP, the Common Lisp Analytical Statistics Package, offers an environment suited for the analysis of experiment data. CLIP, the Common Lisp Instrumentation Package, provides a means for designing and instrumenting experiments. With the addition of a "layer" of dependency detection tools, the CLIP/CLASP environment offers tools to 1) design and instrument an experiment, 2) run the experiment, collecting data as the program executes, 3) analyze the data, and 4) apply the dependency detection algorithms to form predictive rules about the program's behavior. The interface to CLASP and the dependency detection tools is pictured in figure 5.

The graphical interface to the dependency detection offers an easy facility to generate and

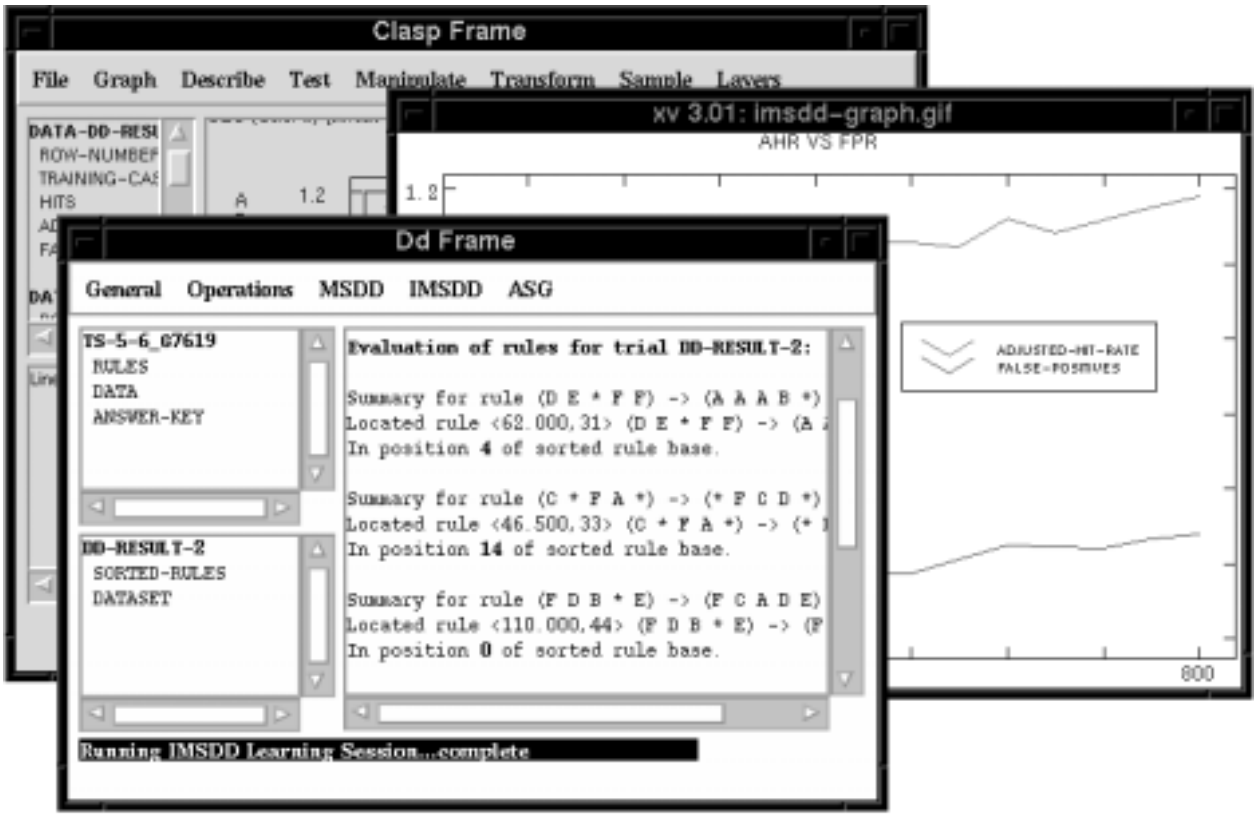

Figure 5: An excerpt from a dependency detection session in the CLIP/CLASP environment. At the front is the DD frame, currently evaluating the rules generated by an IMSDD session on artificially generated streams. Behind the DD frame is the CLASP interface, along with a graph of hit accuracy and false positive rate during the IMSDD session. At the top of the DD frame is a menu of commands for accessing the dependency detection algorithms. The top, left pane of the DD frame displays the currently loaded time series and the bottom left pane displays various results of the current session. The large pane in the DD frame displays output from the dependency detection commands.

analyze dependency rules for datasets. For example, using CLIP, a trial of TransSim is recorded into a CLASP dataset with measurements of port status, docking queue lengths, and the like. This data is then submitted to the DD frame, pictured as TS-5-6_G7619 in the figure. After specifying parameters such as aggressiveness and search-length, MSDD is then run by choosing the "Rule Only Session" command from the MSDD menu. The resulting rules, shown as "DD-RESULT-2" in figure 5, are then viewed or exported to a file and used to supplant TransSim's pathology demon.

Likewise, the entire TransSim system could be loaded in alongside CLASP, and its output relayed directly to IMSDD using the "Operate on System" command in the IMSDD menu. This command allows the user to specify where IMSDD will get its data (i.e. from a TransSim function) and how to communicate its predictions (i.e. via a global variable). The result of these actions would essentially be the integration of IMSDD into the pathology demon of TransSim as alluded to in section 3.3.

Finally, because the CLASP environment facilitates the free exchange of information, the results of the TransSim trials could be analyzed using the statistical and analytical functionality of CLASP.

6 Conclusions

We have presented a methodology for detecting and characterizing complex dependencies in time series data. Based on single and multi-stream representations, the DD, MSDD, and IMSDD algorithms are implementations of this methodology. The strength of the algorithms lie in their generality of representation. The structure they derive is independent of domain, but reflects the observed behavior of the data they monitor. We described one such application of these general methods as a pathology demon in the shipping simulation TransSim, and noted that MSDD, using only high level information gathered from execution traces, was able to produce results comparable to the pathology demon that was built from domain specific knowledge and control strategies.

IMSDD, the newest of the dependency detection algorithms, was described in detail, as a representative of the dependency detection process. We showed how the transition from batch processing, in MSDD, to incremental processing, in IMSDD, was handled, and how the complexity of the search for predictive dependency rules was managed. Our discussion of IMSDD concluded with an empirical evaluation of IMSDD, showing that IMSDD was capable of quickly accounting for structure in data and could produce accuracy comparable to MSDD while offering the benefit of incremental operation.

Finally, we described the current implementation of the dependency detection algorithms as a module in the CLIP/CLASP empirical analysis package, and revisited the TransSim experiment in the context of this environment. Taken as a whole, CLIP/CLASP and the dependency detection layer offer tools for analyzing the behavior of computer programs, and in particular, provides a general method for building predictive rules about the way our programs behave.

Acknowledgments

This work was supported by ARPA/Rome Laboratory under contract number F30602-93-C-0010 and NTT Data Communication Systems Corp. The U.S. government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon.

References

[1] Scott D. Anderson, Adam Carlson, David L. Westbrook, David M. Hart, and Paul R. Cohen. Tools for experiments in planning. In *Proceedings of the Sixth International Conference on Tools with Artificial Intelligence.*, pages 615–623, 1994.

[2] Donald J. Berndt and James Clifford. Using dynamic time warping to find patterns in time series. In *Proceedings of the AAAI-94 Workshop on Knowledge Discovery in Databases*, pages 359–370, 1994.

[3] John Fox and J. Scott Long. *Modern Methods of Data Analysis.* Sage, 1990.

[4] Adele E. Howe and Paul R. Cohen. Understanding planner behavior. *To appear in AI Journal*, Winter 1995.

[5] Tim Oates and Paul R. Cohen. Toward a plan steering agent: experiments with schedule maintenance. In *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems*, pages 134–139, 1994.

[6] Tim Oates, Dawn E. Gregory, and Paul R. Cohen. Detecting complex dependencies in categorical data. In *Preliminary Papers of the Fifth International Workshop on Artificial Intelligence and Statistics.*, pages 417–423, 1995.

[7] John W. Tukey. *Exploratory Data Analysis.* Addison-Wesley, 1977.